Operations concurrentes et files de priorite

B. Le Cun, Bernard Mans, Catherine Roucairol

▶ To cite this version:

B. Le Cun, Bernard Mans, Catherine Roucairol. Operations concurrentes et files de priorite. [Rapport de recherche] RR-1548, INRIA. 1991. inria-00075013

HAL Id: inria-00075013
https://inria.hal.science/inria-00075013

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

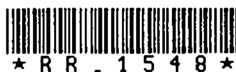
N° 1548

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

OPERATIONS CONCURRENTES ET FILES DE PRIORITE

Bertrand LE CUN
Bernard MANS
Catherine ROUCAIROL

Novembre 1991



Opérations Concurrentes et Files de Priorité

Concurrent Operations and Priority Queues

Bertrand LE CUN, Bernard MANS et Catherine ROUCAIROL

Octobre 1991

INRIA - Action Paradis
Domaine de Voluceau - BP 105 Rocquencourt
78153 Le Chesnay

Laboratoire MASI
Université P. et M. Curie - PARIS 6
4, place Jussieu
75252 PARIS Cedex 05, FRANCE

e-mail: Mans@seti.inria.fr

Résumé

Nous étudions les structures de données de type files de priorité utilisées dans les algorithmes séquentiels et parallèles de recherche arborescente, tels le Branch and Bound.

Après avoir défini les caractéristiques des files de priorité employées par ces algorithmes, nous présentons une comparaison empirique des résultats obtenus sur plusieurs structures de données séquentielles, exécutées lors d'une simulation de Branch and Bound, telles que le D-heap, le Pairing-Heap, le Leftist-Heap, le Skew-Heap, le Funnel-Tree, le Funnel-Table, et différents Splay-Tree.

Nous décrivons la méthode dite de "verrouillage partiel" (partial locking) utilisée dans nos précédents travaux pour paralléliser les structures de données arborescentes dans un contexte d'implémentation sur machines multiprocesseurs à mémoire partagée. Différentes formalisations et preuves des schémas de verrouillage sont données. Une modification de cette méthode permet de diminuer le nombre de primitives de synchronisation et de réduire ainsi le surcoût d'implémentation qu'elles entraînent. Elle est introduite et testée pour l'ensemble des structures de données concurrentes évoquées précédemment.

Dans le cadre des algorithmes Branch and Bound, l'intérêt de ces structures de données tant en séquentiel qu'en parallèle est dégagé. De plus, nous montrons comment la méthodologie décrite peut également s'appliquer à de nombreuses autres structures de données arborescentes dans des contextes différents.

Abstract

In this paper, we study some priority queues well-suited to the context of sequential and parallel Branch and Bound algorithms.

First, we describe the required properties of the priority queues for Branch and Bound algorithms. Second, we present an empirical comparison of the sequential results obtained, during the simulation of a Branch and Bound implementation, on data structures such as: D-heap, Pairing-Heap, Leftist-Heap, Skew-Heap, Funnel-Tree, Funnel-Table, and some Splay-Trees.

Then, we describe the Partial-Locking methodology, used in our previous works, in order to allow parallelization of tree data structures on multiprocessors with shared memory. Several schemes and proofs are given. We also present an improvement of this method which reduces the number of synchronizations primitives and therefore reduces the implementation overcost. Considering this modification, a set of comparatives results for the concurrent data structures is presented.

The interest of several data structures, sequential and parallel, is clearly detected in the Branch and Bound context. However, we show how the methodology presented can be also applied for other tree data structures in different contexts.

Key words.

Branch and Bound Algorithm, Combinatorial Optimization, Concurrency, Data Structures, Parallel Processing, Priority Queues.

Table des matières

1 Introduction **1**

2 Opérations concurrentes sur des structures de données **4**

2.1 Définitions 4

2.2 Structures parallèles passives 5

2.2.1 Sans Primitive de synchronisation 5

2.2.2 Avec Primitive de synchronisation 6

2.3 Structures Parallèles actives 8

2.3.1 File d'attente de Cheng 8

2.3.2 File de priorité de Fan et Cheng 10

2.4 Conclusion 10

3 Résolution des problèmes combinatoires difficiles: algorithme général de Branch and Bound (BB) **11**

3.1 Propriétés de l'algorithme 12

3.2 Caractéristiques de la file de priorité d'un BB 13

3.3 Algorithme et simulation du BB 15

4 Opérations séquentielles sur des files de priorité **17**

4.1 Description des files de priorité et des algorithmes des opérations de base 18

4.1.1 D-Heap 19

4.1.2 Leftist-Heap 21

4.1.3 Skew-Heap 22

4.1.4 Pairing-Heap 23

4.1.5 Semi-Splay-Tree 24

4.1.6 Single-Semi-Splay-Tree 27

4.1.7 Funnel-Tree 29

4.1.8 Funnel-Table 30

4.2 Résultats expérimentaux 31

5 Méthodologie des accès aux files de priorité 37

5.1 Méthodologie avec verrou 38

5.2 Méthodologie avec marquage 46

5.3 Comparaison des méthodologies pour deux files de priorité . . 47

5.3.1 Skew-Heap 47

5.3.2 Funnel-Tree 48

6 Nouveaux algorithmes concurrents pour l'accès à d'autres files de priorité 50

6.1 Funnel-Table 50

6.2 Splay-Tree 51

6.3 Expérimentations 53

6.3.1 Splay-Tree 54

6.3.2 Comparaison avec les files de priorité concurrentes existantes 55

7 Conclusion 57

Table des figures

2.1 Exemple d'exécution pour une File d'attente parallèle. 9

3.1 Exemple d'arbre de GBB 13

4.1 D-Heap 2 Fils 19

4.2 D-Heap 3 Fils 20

4.3 D-Heap 4 Fils 20

4.4 Leftist Heap 21

4.5 Skew Heap 22

4.6 Pairing Heap 23

4.7 Les différentes rotations (a) ZIG, (b) ZIG-ZIG, (c) ZIG-ZAG. Les cas symétriques ont été omis. 24

4.8 La construction finale de l'arbre dans l'opération de Splay. Le nœud x contient la valeur recherchée. 25

4.9 ZIG-ZIG modifié du Semi-Splay. Le cas symétrique est omis. . 25

4.10 Semi-Splay Tree 26

4.11 Simple-Semi-Splay Tree 27

4.12 Single-Semi-Splay Tree 28

4.13 Single-Simple-Semi-Splay Tree 29

4.14 Nombre maximal d'éléments présents dans la file 32

4.15 Nombre total d'opérations. 32

4.16 Test des D-Heaps: 2, 3 et 4 Fils. 33

4.17 Test des 4 Versions de Splay Tree. 34

4.18 Test de toutes les FP. 35

5.1 Structure pourcourue en Partial Locking. 38

5.2 Exemple de fenêtre pour le schéma 1. 40

5.3 Exemple de fenêtre pour le schéma 2 44

5.4 Test sur les différents Skew-Heap 47

5.5 Test sur les différents Funnel-Tree 49

6.1 Test sur les différents Splay-Tree 54

6.2 Test de toutes les FP implémentées 56

Chapitre 1

Introduction

Les méthodes de recherche arborescentes, tel le Branch and Bound, utilisées pour résoudre les problèmes NP-difficiles utilisent dans leur implémentation une file de sous-problèmes obtenus par décomposition du problème originel. Suivant une stratégie définie préalablement, un sous-problème partiel (un élément de la file) est sélectionné et est partagé à nouveau, sauf si on peut prouver que les sous-problèmes résultants ne peuvent conduire à une solution optimale, ou qu'ils ne peuvent plus être décomposés.

La structure de données utilisée est donc généralement une file de priorité. Les opérations de bases sont :

- *deletemin*, lorsque le sous-problème avec la plus grande priorité est choisi pour les décompositions suivantes.
- *insert*, lorsque les nouveaux sous-problèmes ont été générés et doivent être inclus dans la file pour être pris en compte ultérieurement.
- *deletegreater than x*, cette opération sera utile à chaque mise à jour de la borne supérieure. Une borne inférieure[1] des valeurs des solutions pour chaque sous-ensemble créé par partitionnement étant calculé, les sous-problèmes dont la borne dépasse la valeur de la meilleur solution connue peuvent être écartées.

Lorsque les fonctions d'évaluations et la stratégie du Branch and Bound sont fixées, il est intéressant d'étudier les améliorations possibles de la structure de file de priorité pour accélérer la recherche arborescente. De plus, on

[1]dans le contexte d'un problème de minimisation

peut espérer accroître la vitesse de résolution en introduisant, en particulier, le parallélisme asynchrone pour des machines multiprocesseurs à mémoire partagée.

De nombreuses structures de files de priorité ayant été proposées ces dernières années, il nous semble intéressant de caractériser celles qui semblent convenir le plus à l'implémentation des algorithmes Branch and Bound en séquentiel et en parallèle (sur des machines multiprocesseurs à mémoire partagée). Ce travail nous amène, tout en rappelant l'état de l'art des structures de données, à décrire une comparaison exhaustive des files à priorité, des structures auto-ajustables aux structures intrinsèquement parallèles.

De plus, la parallélisation de l'accès aux structures de données pose certains problèmes:

- il faut résoudre les problèmes de **Famines**:
  une opération réalisee par un processus sur une structure de données, doit se faire dans un temps fini,
- il faut éviter tout **Interblocage**:
  des processus ne doivent pas détenir des ressources non préemptives devant être acquises par certains d'entres eux et risquant de les bloquer mutuellement,
- il faut assurer la **Consistance** de la structure:
  à chaque instant la structure doit garder toutes ses propriétés de validité.

C'est dans ce contexte que nous étudierons les possibilités de rendre parallèle l'accès aux files de priorité pour améliorer les résultats des algorithmes parallèles de Branch and Bound.

Pour cela, nous commencerons par exposer dans le deuxième chapitre, plusieurs travaux sur la parallélisation des structures de données. Nous justifierons l'emploi de la méthode du "verrouillage partiel" (partial locking) par rapport aux autres techniques existantes.
Nous exposerons dans un troisième chapitre, les propriétés des algorithmes de Branch and Bound afin de définir les caractéristiques nécessaires aux files de priorité.
Une comparaison empirique de plusieurs files de priorité séquentielles exécutées lors de simulation d'un algorithme de Branch and Bound, sera exposée au chapitre 4.
Dans le chapitre 5, nous présenterons plus en détail la méthodologie de "partial locking" dans le contexte de la parallélisation de files de priorité. Puis

nous apporterons une modification à cette méthode, nous permettant de réduire le surcoût dû au parallélisme et donc d'en augmenter l'efficacité. Nous illustrerons cette amélioration par une comparaison des résultats, obtenus sur une machine Sequent, des files de priorité telles le Skew-Heap et le Funnel-Tree.

Parmi les files de priorité séquentielles décrites auparavant, nous présenterons des parallélisations originales des files les mieux adaptées aux besoins des algorithmes de Branch and Bound (chapitre 6).

Une conclusion sera donnée dans le chapitre 7.

Chapitre 2

Opérations concurrentes sur des structures de données

2.1 Définitions

Nous allons présenter différents travaux qui ont été réalisés quant à l'accès aux structures de données en parallèle.

Définition 1 Structure de données parallèle: *nous appelons structures de données parallèle, des structures permettant la réalisation de plusieurs opérations en parallèle.*

Les structures de données parallèles offrent la possibilité à plusieurs processus de réaliser des opérations en même temps. Si un processus P_i débute une opération sur une structure parallèle S, un autre processus P_j peut commencer une opération sur S sans attendre que P_i ait terminé la sienne.

Dans la littérature, nous pouvons trouver deux grands types de structures de données "parallèles". Les structures que nous appellerons structures parallèles "passives" (SPP) et les structures parallèles "actives" (SPA).

Définition 2 SPP: *Les structures parallèles passives, SPP, désignent les structures de données parallèles où les processus voulant réaliser une opération, effectuent eux-mêmes les modifications dans la structure.*

Ce type de structures est conçu pour des machines parallèles à mémoire partagée, ayant peu de processeurs. Les SPP sont, en général, des structures

séquentielles dont les algorithmes d'accès ont été parallélisés. L'accès à la structure et l'opération sont effectués par le meme processus.

Définition 3 SPA: *les structures parallèles actives, SPA, désignent les structures de données parallèles où les processus voulant réaliser une opération, demande à des processus spécialisés d'effectuer l'opération pour eux.*

Une structure de ce type, n'est gérée que par des processus spécialisés. Lorsqu'un processus veut effectuer une opération, il fera une demande.
Le fonctionnement des SPA leur permet de pouvoir être appliquées au contexte distribué.

2.2 Structures parallèles passives

La modification d'une structure de données est en en général une succession de modifications élémentaires. Le problème de la parallélisation est de rendre atomiques, ces modifications élémentaires effectuées par un processeur, vis-à-vis des autres, afin de garder les propriétés de la structure considérée. Dans la littérature, on rencontre plusieurs méthodes pour résoudre ce problème.

- sans l'utilisation de primitives de synchronisation.
  Compare&Swap [Her90].
- avec l'utilisation de primitives de synchronisation.
  à plusieurs niveaux [Ell80, Ell81, KW83, KL80, LY81].
  à un niveau [BB87, RK88, Jon89, MR90].

Ces méthodes sont décrites dans les deux sous-sections suivantes.

2.2.1 Sans Primitive de synchronisation

Herlihy, [Her90], propose une méthodologie permettant de rendre concurrente les structures de données sans utilisation de primitives de synchronisation en introduisant le concept de Compare&Swap. Le principe de cette méthode est très simple.
Une structure de données séquentielle est souvent référencée par un unique

point d'entrée. En parallèle, lorsqu'un processus P voudra modifier la structure: il en fera une copie locale, effectuera son opération sur cette copie, puis essayera de la faire référencer dans l'espace global. En terme de programmation, il va essayer de changer le pointeur de référence global.

Dans la cas du Compare&Swap, un processus P_i voulant réaliser une opération, sauvegardera le pointeur de référence global en local, puis lors de l'essai de recopie de la structure modifié, il testera l'égalité entre l'ancien et l'actuel pointeur de référence. S'ils sont différents, alors un processus $P_j (i \neq j)$ a modifié la structure pendant que P_i effectuait son opération sur sa copie locale. Dans ce cas, le Compare&Swap a échoué. Afin de garder la structure consistente, P_i doit recommencer son opération depuis le début. Si les deux pointeurs sont égaux, P_i peut faire référencer sa nouvelle structure en global. Il est à noter que l'opération test plus recopie formant la procédure de Compare&Swap est atomique.

2.2.2 Avec Primitive de synchronisation

Les primitives de synchronisation sont utilisées pour assurer les sections critiques lors des modifications dans une structure. Elles sont généralement représentées par des verrous ou des sémaphores. Par exemple, pour une insertion ou une suppression dans un arbre AVL [El181], il faut effectuer un équilibrage de l'arbre pour assurer la validité de la propriété des arbres AVL. Si on utilise des nœuds avec trois champs: une valeur, un pointeur sur le fils gauche et un sur le fils droit, des modifications de pointeurs sont à effectuer. Pour éviter une utilisation par un autre processus des nœuds que l'on modifie, on utilise des verrous. Si un processus veut modifier un nœud, il doit d'abord le verrouiller.

Un processus qui réalise une opération avec cette technique, parcourt la structure et la modifie lorsqu'il en a besoin. Il n'a pas besoin de verrouiller complètement la structure. Il verrouille les nœuds dont il a besoin et déverrouille les nœuds qu'il a déjà modifiés. Cette technique est appelée verrouillage partiel (Partial Locking) [Jon89, Deo89, MR90]. Toutes les techniques de parallélisation décrites ici, utilisant des primitives de synchronisation accèdent à la structure en "Partial Locking". Cette méthodologie sera expliquée plus en détail au chapitre 5.

Cette méthode a été utilisée pour plusieurs structures de donnés. La parallélisation des files de priorité comme le Skew-Heap [Jon89, MR90], le D-Heap [RK88, BB87], le Funnel-Tree [MR90], nécessite un seul niveau de verrou car chaque opération modifie la structure. Mais cette méthode a aussi

été utilisée pour paralléliser les arbres de recherche en base de données (par exemple: B-Arbre [LY81, KL80, KW83, Ell80], B+Arbre) ou les arbres binaires de recherche (arbre AVL [Ell81]). Plusieurs types de verrous de niveaux différents sont utilisés, dans le cas des arbres de recherche, les besoin des processus peuvent être différents. Un processus peut n'avoir besoin que de lire. Il est inutile d'assurer une section critique entre ces lecteurs puisqu'il ne modifie pas la structure. Sont ainsi utilisés:

- un verrou en lecture pour la recherche de valeur dans l'arbre,
- un verrou en écriture pour la phase de recherche dans les opérations de suppressions ou d'insertions,
- un verrou exclusif pour la phase de modification de la structure lors de ces mêmes opérations,

Les dépendances de verrous sont les suivantes:

- compatibilités de verrous
  - N Lecteurs $\Longleftrightarrow$ N Lecteurs
  - 1 Ecrivain $\Longleftrightarrow$ N Lecteurs
- incompatibilités de verrous
  - Ecrivain $\not\Leftrightarrow$ Ecrivain
  - Exclusif $\not\Leftrightarrow$ Exclusif
  - Ecrivain $\not\Leftrightarrow$ Exclusif
  - Lecteur $\not\Leftrightarrow$ Exclusif
- transition possible
  - Ecrivain $\Longrightarrow$ Exclusif

Lorsqu'un écrivain veut réaliser une opération, il verrouille la partie de l'arbre qu'il veut modifier avec des verrous en écriture, puis il doit les changer en verrou exclusif une fois le nœud trouvé. Ceci produit des parcours redondants de la structure, et donc un surcoût dû au parallélisme.

Le problème est ici d'avoir le minimum de nœuds verrouillés à un instant donné, afin d'avoir le maximum de processus travaillant en même temps dans la structure, et donc d'avoir un maximum de concurrence.
D'autre part, si deux opérations sur une même structure s'effectuent dans

deux sens différents, il y a des risques d'interblocage.
Rao et Kumar [RK88] résolvent le problème dans le cas d'un tas, en modifiant l'opération d'insertion pour qu'elle se réalise dans le sens Racine-Feuille, et non plus dans le sens Feuilles-Racine.
Bitwas et Browne, [BB87], résolvent ce même problème en colorant les nœuds par lesquels il passe, afin de connaître si le nœud qu'un processus veut verrouiller, est déjà détenu par un processus qui parcourt la structure dans un sens différent du sien.
Dans une deuxième version, Bitwas and Browne utilisent des processus spécialisés pour effectuer la phase de restructuration. Les processus voulant réaliser une opération, demandent aux processus spécialisés de l'effectuer pour eux. Cela permet aux processus demandeur de se décharger d'une partie du travail. Mais le nombre de processus spécialisés est indépendant de la taille de la structure. Dans la section suivante, nous allons étudier les structures parallèles qui ne sont gérées que par des processus spécialisés.

2.3 Structures Parallèles actives

Dans ce type de structure, les processus qui désirent réaliser une opération font une demande à des processus spécialisés. Chaque processus susceptible de vouloir accéder à la structure a sa propre entrée. Si on a P processus, on a P points d'accès à la structure, et donc P accès simultanés. Ceci permet un accroissement de la concurrence. De plus ces structures peuvent aisément être utilisées dans un contexte distribué car leurs algorithmes consistent à échanger des valeurs entre processus. Il existe plusieurs structures conçues sur ce modèle. Nous avons choisi de présenter, dans la suite, deux exemples de structures de ce type: la file d'attente de Cheng [Che90, FC'90], et la file de priorité de Fan et Cheng [FC'89, FC'90].

2.3.1 File d'attente de Cheng

Un nombre P de processus peuvent accéder à P files d'attentes séquentielles simultanément, par l'utilisation d'un réseau multicouche d'interconnexion. Le réseau est composé de $log(P)$ couches, chaque couche étant gérée par P processus spécialisés. Les processus spécialisés sont donc au nombre de $P\ log(P)$.
L'algorithme consiste à trouver un chemin dans le réseau d'un point d'accès d'un processus vers une des files séquentielles, en prenant en compte: le nombre de valeurs restantes dans les files, le nombre d'insertion et le nombre

de suppression pour un ensemble donné de demandes.
Le calcul est effectué par vagues successives. Les processus d'une couche travaillent sur un ensemble d'opérations de façon synchrone, ils aiguilleront leurs demandes vers un processus particulier du niveau supérieur en fonction des résultats du niveau inférieur.

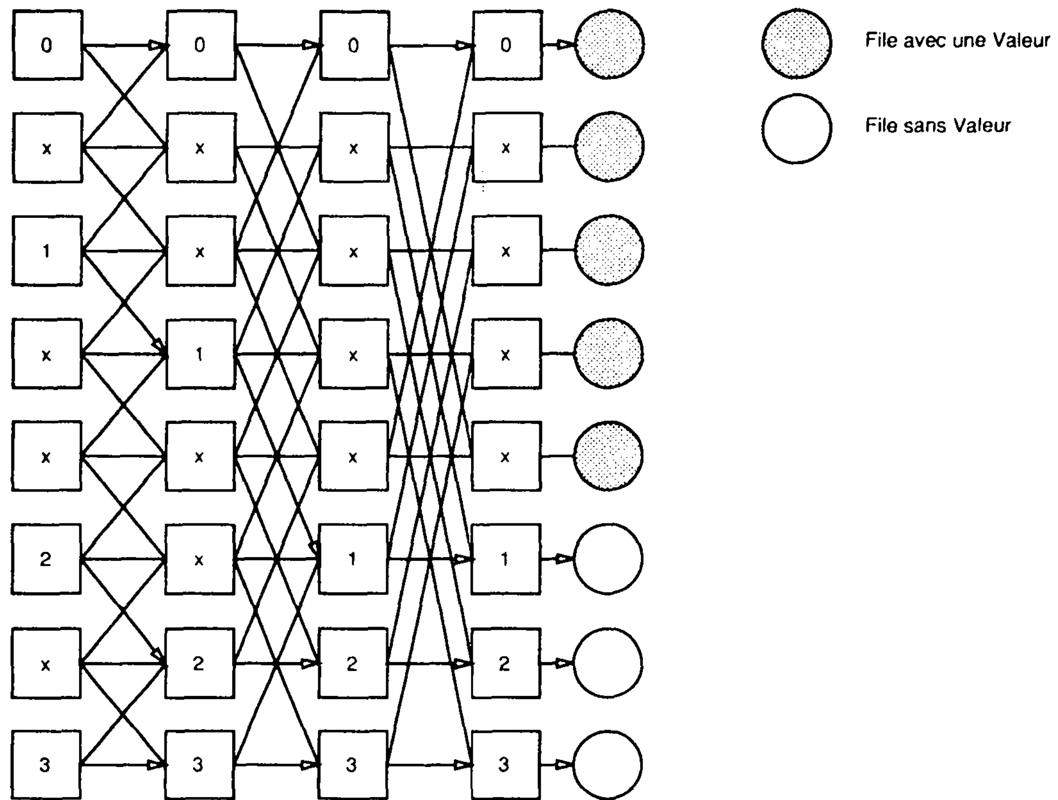


Figure 2.1 : Exemple d'exécution pour une File d'attente parallèle.

Afin de garder la propriété des files d'attentes, les files seront remplies et vidées à tour de rôle.
En effet, sur l'exemple de la figure 2.1, il ne reste qu'une valeur dans chacune des cinq premières files. Nous avons ici 4 valeurs à insérer: 3 d'entre elles seront insérées dans les 3 files vides du bas, la quatrième sera insérée dans la première file.

La complexité de cette structure pour l'insertion et la suppression est $O(logP)$ avec P $log(P)$ processus spécialisés.

2.3.2 File de priorité de Fan et Cheng

Cette file de priorité parallèle reprend les principes des structures parallèles énoncés précédemment. Les processus accèdent à la file par l'intermédiaire d'un vecteur de demande de taille P. La structure est aussi composée de couches. Chaque couche est de taille P. Si on a N valeurs, on aura $(N/P)+1$ couches. Plus la couche est basse, plus les valeurs contenues sont de "basse" priorité. Le vecteur de demande est considéré comme la couche 0.
L'algorithme consiste à échanger les valeurs entre les couches en fonction de celles demandées ou données à la structure. Alternativement les couches impaires puis les couches paires travaillent.
Comme la *structure précédente* tous les *processus d'une couche travaillent* de façon synchrone.

2.4 Conclusion

L'intérêt des structures parallèles est qu'il n'y a aucune interaction entre les processus y accédant et donc aucune attente dû au parallélisme. Le résultat en terme de speed-up est très intéressant.
Mais le nombre de processeurs utilisés est pénalisant dans un contexte de machines MIMD à *mémoire partagée*. Elles les rendent même inutilisables sur des machines existantes. Elles ne peuvent donc convenir pour le modèle dans lequel nous nous sommes placés.
En revanche les structures à accès parallèles sont mieux adaptées à notre but, car elles peuvent aisément résider en mémoire partagée. Dans toute la suite, les structures dont nous parlerons seront donc de ce type.

Chapitre 3

Résolution des problèmes combinatoires difficiles: algorithme général de Branch and Bound (BB)

Le but des algorithmes d'évaluation et séparation (General Branch and Bound BB) est de résoudre des problèmes d'optimisation combinatoire sous contraintes:

$$min\ f(x), x \in X$$

où X représente le domaine d'optimisation,
x est une solution réalisable si elle appartient au domaine X,
$f(x)$ est la valeur d'une solution (critère d'optimisation ou fonction economique).

La méthode du BB est basée sur l'idée d'une énumération intelligente de toutes les solutions réalisables d'un problème d'optimisation combinatoire. Le principe de l'algorithme de BB est de décomposer un problème donné en deux ou plusieurs sous-problèmes de tailles inférieures.
Suivant une stratégie définie préalablement, un sous-problème partiel est sélectionné puis est partagé, sauf si on peut prouver que les sous-problèmes résultant ne peuvent conduire à une solution optimale, ou qu'ils ne peuvent plus être décomposés.

Afin de choisir un sous-problème parmi ceux qui n'ont pas été examinés, une évaluations (borne inférieure) des solutions dans chaque sous-problème

est introduite. Les sous-problèmes dont l'évaluation est supérieure à la valeur de la meilleure solution connue peuvent être supprimés.

Nous pouvons représenter l'état du partitionnement d'un problème, comme un arbre partiel. Un sous-problème est représenté par un nœud de l'arbre, les problèmes en lesquelles il se subdivise sont ses fils.

Plus précisément, un algorithme de BB repose sur les trois notions clés:

- le schéma de partition qui associe à tout sous-problème l'ensemble des sous-problèmes en lesquels il se décompose,
- une fonction d'évaluation qui associe à tout sous-ensemble de solutions, un nombre représentant le coût minimal de toutes ces solutions,
- une stratégie de recherche qui est utilisée pour sélectionner parmi les sous-problèmes restant, celui qui sera séparé. Le nœud sélectionné sera celui de plus grande priorité.

L'implémentation d'un algorithme de BB consiste donc à réaliser des opérations sur une file de sous-problèmes de priorités différentes ou égales. La structure de données utilisée est une file de priorité dans laquelle les sous-problèmes (éléments) sont insérés ou supprimés.

3.1 Propriétés de l'algorithme

L'évaluation d'un sous-problème ne peut être inférieure à celle du problème dont il est issu. Donc, la fonction d'évaluation qui affecte des valeurs aux sous-problèmes fils par rapport au sous-problème père, est monotone. Nous verrons au chapitre 4, que la propriété de monotonie de l'algorithme de BB, nous sera très utile pour l'implémentation de certaine files de priorité.

La particularité commune à tous les algorithmes de BB est qu'ils évitent l'examen complet d'un arbre pour trouver une ou la solution, par acquisition de connaissances leur permettant d'effectuer des élagages tout au long des parcours.
Lorsqu'une solution S de valeur e_s est trouvée, les sous-problèmes dont l'évaluation est supérieur à e_s ne pourront pas générer de solution optimale. Générer les fils des sous-problèmes dans ce cas est donc inutile. Il faut donc pouvoir éliminer ces sous-problémes de la file.

D'autre part, plusieurs sous-problèmes pourront avoir des évaluations identiques. De nombreux travaux ([MR91, LS84, LW86]) ont montré qu'un

choix indéterministe parmi ces sous-problèmes, pouvait conduire à des anomalies de performances, lors de l'implémentation parallèle (sous-accélération: speed-up inférieur à 1, sur-accélération: speed-up supérieur au nombre de processeurs).

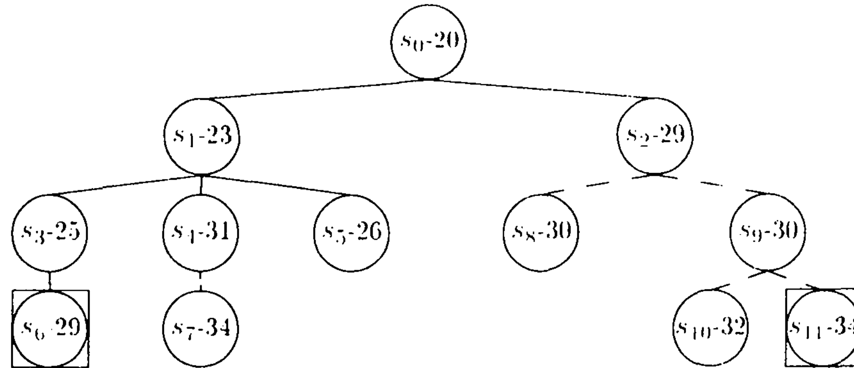


Figure 3.1 : Exemple d'arbre de GBB

La figure 3.1 montre un exemple d'arbre de solution d'un BB. Chaque nœud contient la référence et l'évaluation du sous-problème associé.
Les nœuds s_6 et s_{11} sont des solutions de valeurs 31 et 34. Les sous-problèmes s_7, s_{10} et s_{11} ont des évaluations supérieures à la valeur de s_6, ils doivent donc être élagués.

Afin de choisir les structures de données les plus appropriées, il nous faut d'abord définir leurs caractéristiques dans le contexte d'algorithmes de BB.

3.2 Caractéristiques de la file de priorité d'un BB

Les files de priorité (FP) offrent deux opérations de base:

- DeleteMin: suppression de l'élément de plus grande priorité.
- Insert: insertion d'un élément de priorité quelconque.

Lorsque des élagages doivent être effectués, nous avons besoin d'une opération qui détruise tous les éléments présents dans la file dont les priorités

sont inférieures à une valeur donnée. Cette opération est appelée DeleteGreater.

Les files de priorité généralement utilisées n'offrent pas la possibilité de réaliser cette opération avec un coût acceptable (cf chapitre 4).

La méthode alors utilisée, consiste à tester, lors de la selection d'un sous-problème, si son évaluation ne dépasse pas la valeur de la meilleure solution connue.

Le défaut de cette méthode est que les performances de la structure sont altérées par la présence d'éléments inutiles. Nous pouvons voir l'opération de DeleteGreater comme un ramasse-miettes (garbage-collector). La file de priorité d'un BB ne doit pas avoir obligatoirement cette opération, mais sa présence peut améliorer les peformances du BB.

Afin de répondre au problème posé par les éléments de même priorité, nous avons besoin de FP stable. Si un élément A de priorité pr est inséré avant un autre élément B de même priorité, l'algorithme de la structure doit nous assurer l'ordre dans lequel ils seront supprimés. De plus il serait même intéressant de pouvoir choisir cet ordre en modifiant que très peu l'algorithme, avec un surcoût le plus faible possible.

3.3 Algorithme et simulation du BB

L'algorithme BB est donné par le programme suivant:

Variable
  x : Element
  pr,
  prfils :priorite

procedure *BB*
debut
  si (une bonne solution réalisable x^0 est connue) **alors**
    $BS{:=}f(x^0)$
  sinon
    $BS{:=}\infty$
  finsi
  h:=*MakeHeap*(root)
  Insert(BI)
  tant que (h $\neq \emptyset$)
    debut
    x:=*DeleteMin*(h)
    pour tout fils y de x **faire**
      debut
      si (y est une solution réalisable **et** ($f(y) < BS$)) **alors**
        BS:=f(y)
        DeleteGreater(*BS*,h)
      finsi
      si (y n'est pas une feuile **et** (ν(y)< *BS*)) **alors**
        Insert(y,h)
      finsi
    finpour
  fintant
fin *BB*

Pour les expérimentations, plutôt que de travailler avec la résolution d'un problème réel d'optimisation combinatoire, nous avons simulé une exécution d'un BB de la façon suivante:
Les entrées du programme sont: borne inférieure (BI), borne supérieure (BS), pas d'incrémentation maximale (IM).

Nous commençons par insérer un élément de priorité égale à BI. Chaque élément génère deux fils. Les priorités des fils d'un élément i sont calculées en choisissant aléatoirement une valeur entre la priorité pr de i et $pr + IM$. Si la priorité d'un nœud est supérieure à BS, il n'est pas inséré dans la file. L'algorithme s'arrête lorsque la file est vide.

L'expérimentation séquentielle d'une file de priorité utilise cette simulation. Le temps de calcul de la priorité d'un élément étant très rapide, le temps pris par une simulation dépendra essentiellement du temps pris par les opérations réalisées sur la file. Donc pour les mêmes entrées, la différence de deux simulations avec deux files de priorité distinctes, reflètera la différence d'efficacité des opérations des deux files.

En parallèle, nous nous plaçons dans un contexte de machine multiprocesseur à mémoire partagée. Dans ce contexte, tous les processeurs exécuteront cet algorithme mais en accédant à la même FP qui se trouvera en mémoire partagée. La répartition du travail n'est donc pas statique, les processeurs iront le chercher d'eux même dans la FP, jouant ici le rôle d'un serveur passif. Ce mode de travail est appelé "auto-ordonancement" (Self-Scheduling). De même qu'en séquentiel, les résultats des expérimentations, reflèteront l'efficacité des opérations concurrentes de la file.

Si les tests montrent que la file est plus efficace avec N processus qu'avec $N + 1$, l'algorithme ne peut accepter que N opérations concurrentes. Nous dirons que la structure de données sature avec N processus.

Ce modèle sera utilisé dans toutes les expérimentations présentées dans ce rapport.

Chapitre 4

Opérations séquentielles sur des files de priorité

Une comparaison empirique de plusieurs files de priorité (FP), exécutées dans le "Hold Model" a été réalisé par Jones ([Jon86]). Le "Hold Model" consiste à réaliser un grand nombre d'opérations de base sur une FP, tout en gardant un nombre d'éléments constant dans la file. Les valeurs des priorités sont choisies aléatoirement. Ce modèle ne correspond pas à celui du BB car ce dernier produit un nombre variable d'éléments au cours du temps. Il nous a donc paru intéressant de procéder à une comparaison empirique de plusieurs FP séquentielles afin de voir lesquelles correspondaient le mieux à notre modèle.

Parmi les FP expérimentées par Jones, nous avons choisi d'en expérimenter cinq (les D-Heap, la Leftist-Heap, la Pairing-Heap, la Skew-Heap et le Splay-Tree). Ce choix s'explique par les qualités théoriques de performances et/ou la possibilité d'être rendu parallèle.

Les quatre premières sont des FP représentées sous forme de tas (l'invariant du tas est : tout nœud a une priorité inférieure à son père). L'implémentation de l'opération de DeleteGreater est un problème sur des FP en tas. En effet, il faut parcourir la totalité de la structure pour trouver les nœuds de valeur supérieure à une valeur donnée. La complexité de l'opération est donc en $O(n)$ (n est le nombre de nœuds dans l'arbre) [Rou87].

Le Splay-Tree est un arbre binaire de recherche, qui s'est avéré performant utilisé en FP. De plus il est aisé d'y implémenter l'opération de DeleteGreater.

D'autre part, le Funnel-Tree (arbre à entonnoir) et le Funnel-Table introduites par B. Mans et C. Roucairol, [MR90], sont des FP spécifiques

au modèle de BB dans le sens où elles tirent parti du petit intervalle de priorité possible et de la redondance de priorités dans lesquelles les BB sont exécutés. Les opérations DeleteGreater sont dans leur contexte très simples à implémenter.

Dans la suite de ce chapitre, n désigne le nombre d'éléments présents dans la structure.

4.1 Description des files de priorité et des algorithmes des opérations de base

Pour certaines d'entre elles nous avons données un exemple de Delete-Min et un d'Insert. Ils sont tirés de l'exécution d'une simulation du BB dont les paramètres sont les suivants: Borne Inférieure 1, Incrémentation Maximum 20, Borne Supérieure 40. Pour chacune des figures, les trois exemples (a,b,c) représentent l'état des structures dans le même contexte d'exécution.

(a) → (b) (DeleteMin: 30).
(b) → (c) (Insert 39).

4.1.1 D-Heap

Le D-heap est une structure de tas sous la forme d'un arbre complet. Chaque nœud de l'arbre a D fils (excepté les feuilles et occasionnellement le père du dernier sommet).
Sa représentation habituelle est sous forme de tableau.
L'indice du père d'un nœud d'indice i est $\lceil i/D \rceil$. D étant le nombre de fils.
Les indices des fils d'un nœud d'indice D sont :
$(i-1)*D+2, (i-1)*D+3, \ldots, Min(i*D+1, n)$.
L'indice de la racine est 1.

Lors d'un Insert d'un élément x, x est placé à la fin du tableau (dans le nœud d'indice $n+1$).
si la valeur du père de x est supérieur à la valeur de x, l'invariant n'est plus satisfait, donc les deux valeurs sont echangées. On remonte la valeur de x par des échanges successifs jusqu'à satisfaction de l'invariant.
Pour le DeleteMin, le sommet à extraire se trouvant à la racine, la racine est remplacée par le dernier nœud du tableau (nœud d'indice n), puis est redescendue par ces mêmes échanges.
La complexité des opérations d'Insert et de DeleteMin est en $O(log_D\ n)$.

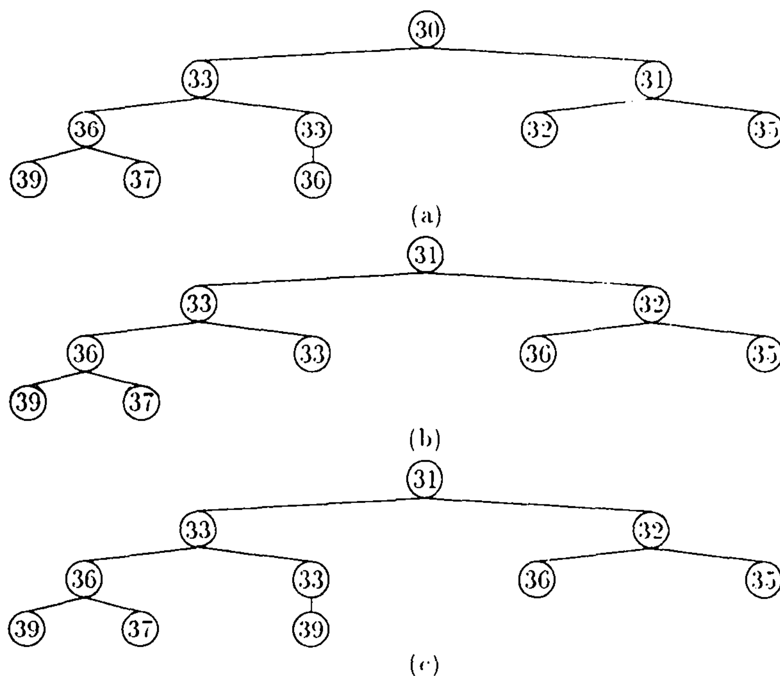


Figure 4.1 : D-Heap 2 Fils

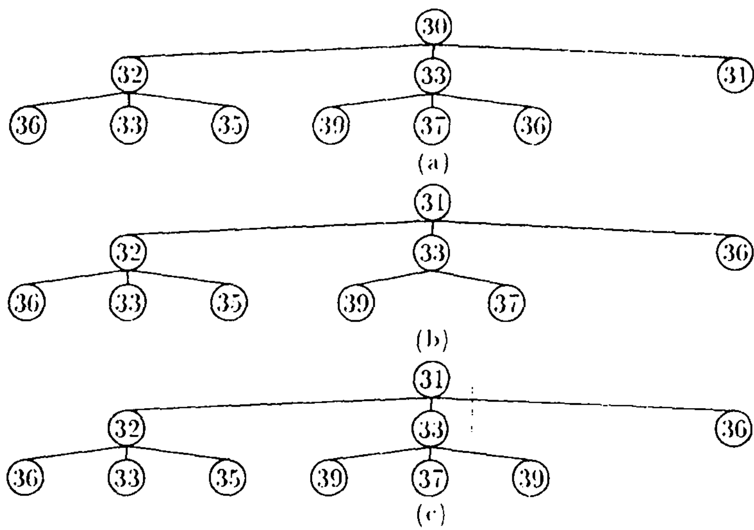


Figure 4.2 : D-Heap 3 Fils

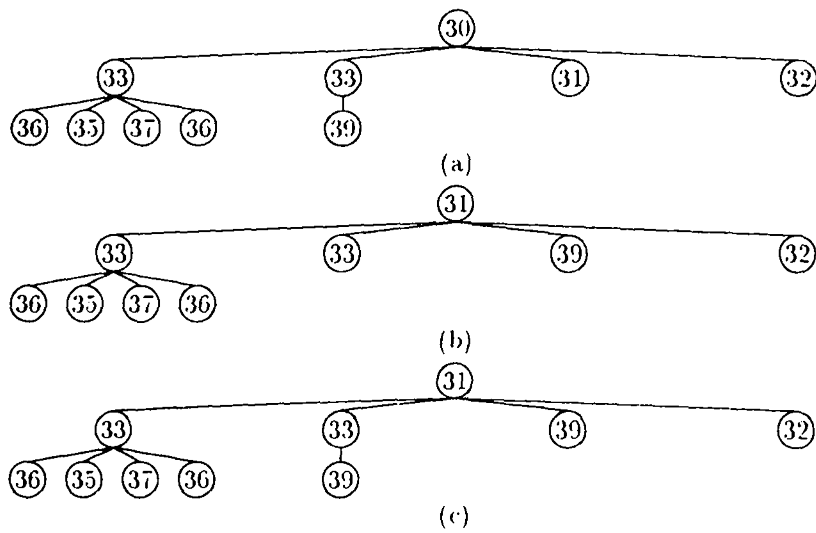


Figure 4.3 : D-Heap 4 Fils

4.1.2 Leftist-Heap

Le Leftist Heap, [Cra72], peut être représenté par un arbre binaire où chaque élément est un nœud.
Chaque nœud contient la valeur de la priorité, les pointeurs sur le fils droit et le fils gauche et une valeur représentant la différence de hauteur entre le sous-arbre droit et le sous-arbre gauche.
L'opération de base du Leftist-Heap est la fusion (melding). Elle consiste à construire un seul tas à partir de deux, en parcourant les branches droites de chacun des deux arbres récursivement. En remontant par la récursivité, à chaque nœud, on procède au calcul de taille du sous-arbre.
Afin d'assurer un équilibrage du tas et de minimiser l'opération de fusion, on effectue des rotations pour que le sous-arbre droit soit plus petit que le gauche.
L'Insert est réalisé en fusionnant le nœud avec le tas. Lors d'un DeleteMin, les deux sous-arbres de la racine sont fusionnés.

La complexité de l'opération de Melding est dans le pire des cas $O(log\ n)$.

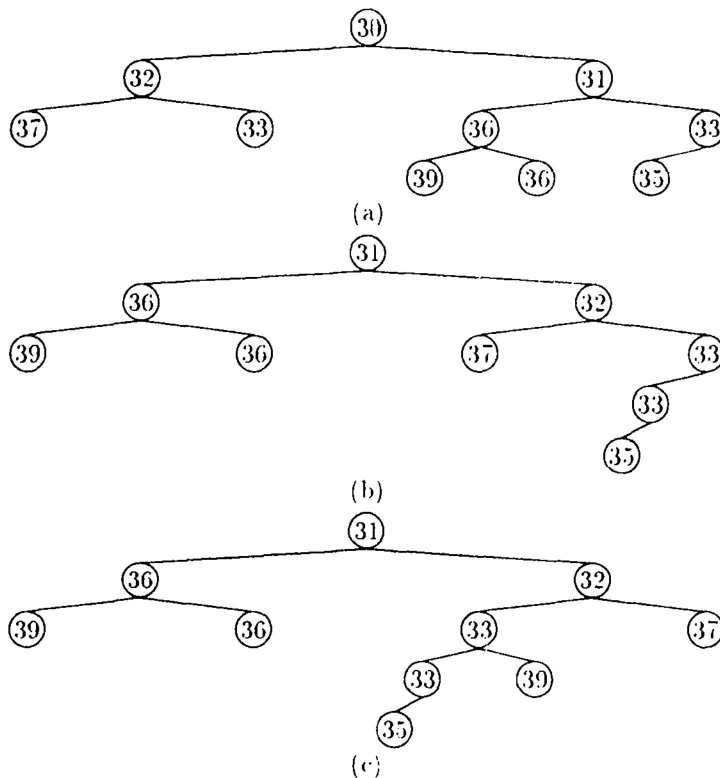


Figure 4.4 : Leftist Heap

4.1.3 Skew-Heap

Le Skew-Heap est un tas créé par Sleator et Tarjan [ST86] dans le cadre de leurs recherches sur la complexité amortie [Tar85]. C'est une version auto-ajustable du Leftist-Heap. Les tests explicites pour garder l'équilibrage de l'arbre sont remplacés par une heuristique. Sleator et Tarjan assurent que pour la pire séquence de m opérations, la complexité est $m\ O(log\ n)$, donnant une complexité amortie en $O(log\ n)$ par opération.

Le principe de l'algorithme de fusion (melding) est de parcourir les branches droites de chacun de deux tas pour n'en construire qu'un, mais à chaque pas de l'opération, afin de garder un équilibrage de l'arbre, on échange le fils droit et le fils gauche du nœud considéré.
Il existe deux implémentations différentes des Skew-Heap: Top-Down et Bottom-Up. Nous avons implémenté la version Top-Down pour les expérimentations présentées dans la section suivante.

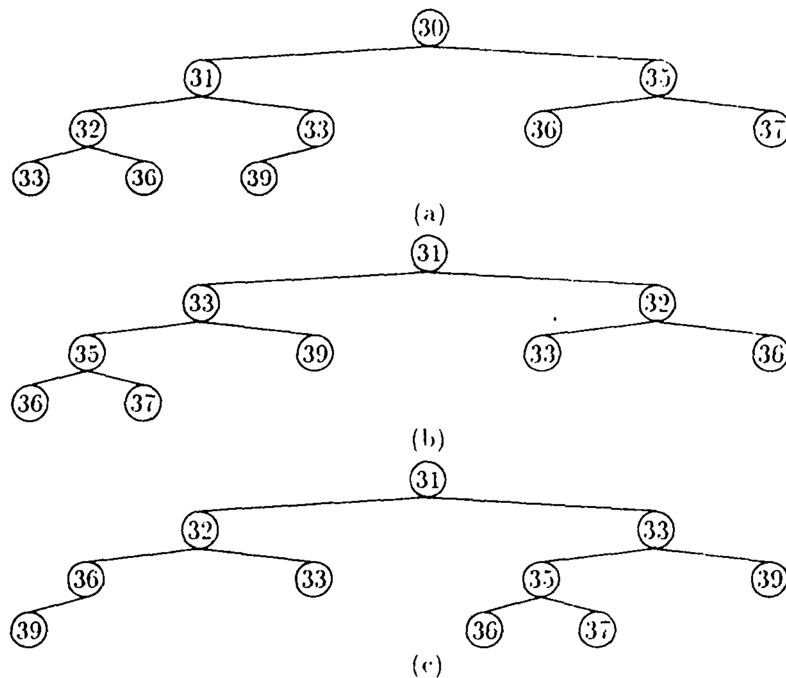


Figure 4.5 : Skew Heap

4.1.4 Pairing-Heap

Le Pairing-Heap est la version auto-ajustable des Binomial-Queue [Vui78, Bro78]. Tarjan avait créer une première version auto-ajustable des Binomial-Queue, appelé Fibonnacci-Heap [FT84], mais les mauvaises performances de cette structure l'ont poussé à créer le Pairing-Heap.
La structure de Pairing est représenté par un arbre K-aire.
L'Insert consiste à insérer le nouveau nœud i comme premier fils de la racine, puis d'échanger les valeurs entre i et la racine afin de satisfaire l'invariant. Cette opération se réalise en $O(1)$.
Pour effectuer un DeleteMin, la racine est retirée, puis on procède à l'opération de "pairing". Elle consiste à parcourir les fils de la racine deux à deux, construire un sous-arbre avec les deux fils considérés, afin de réduire de moitié le nombre de fils de la racine. Puis parmi ce nouvel ensemble de fils, l'élément de plus grande priorité est choisi comme racine. La complexité amortie du DeleteMin est $O(log\ n)$.

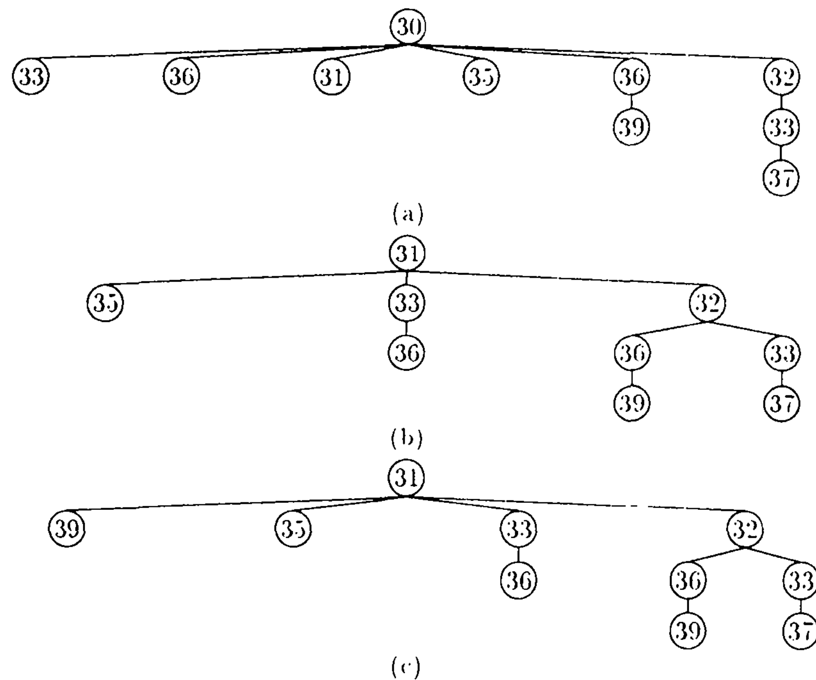


Figure 4.6 : Pairing Heap

4.1.5 Semi-Splay-Tree

Le Splay-Tree est un arbre binaire de recherche créé par Sleator et Tarjan [ST83]. Il est la version auto-ajustable des arbres Red-Black ou AVL. Nous rappelons l'invariant des arbres binaires de recherche:
Pour tout nœud x de valeur i, les nœud du sous-arbre droit (resp:gauche) de x ont des valeurs supérieures (resp:inférieures) à i.
L'opération de base de cette structure est appelée le Splay. Elle consiste lors du parcours de l'arbre pour la recherche d'un élément de valeur i, à construire deux sous-arbres R et L où R (resp: L) contient les valeurs supérieures (resp: inférieures) à i.
L'heuristique utilisée pour garder l'arbre équilibré est la combinaison de trois types de rotations: le ZIG, le ZIG-ZAG et le ZIG-ZIG. Elles sont représentés sur la figure 4.7.

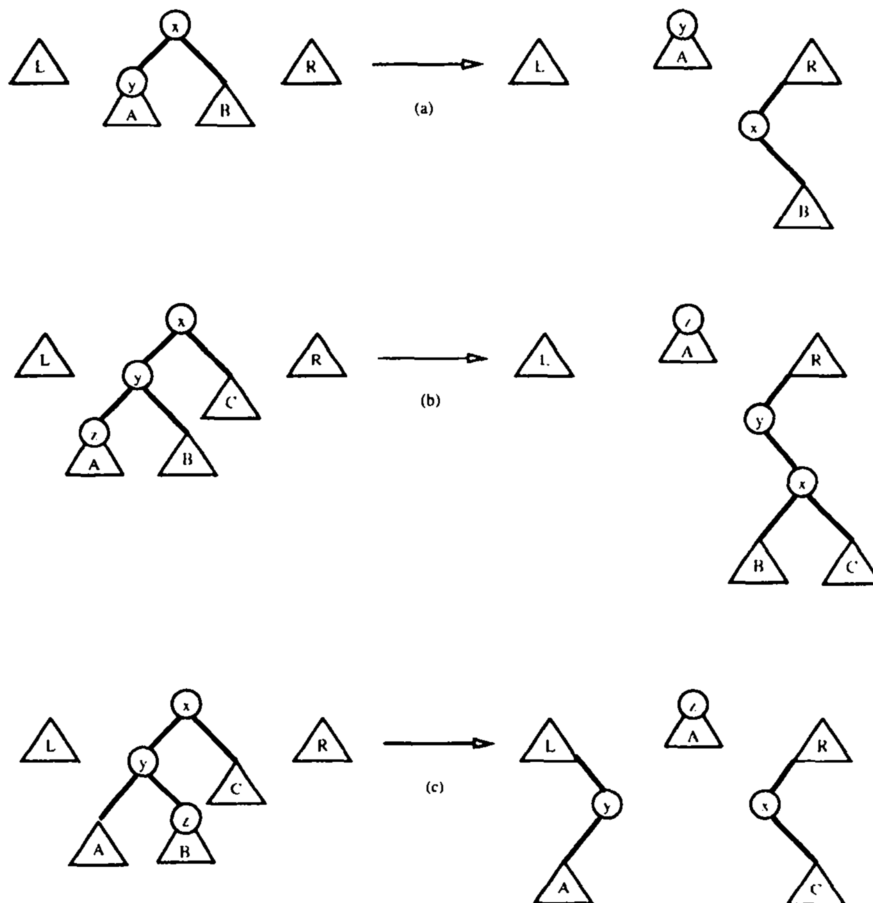


Figure 4.7 : Les différentes rotations (a) ZIG, (b) ZIG-ZIG, (c) ZIG-ZAG. Les cas symétriques ont été omis.

L'arbre est finalement assemblé comme le montre la figure 4.8 lorsque le nœud cherché a été trouvé. Nous pouvons remarquer que dans l'algorithme de Splay, l'élément cherché devient la racine de l'arbre. La complexité amortie de cette opération est $O(log\ n)$.

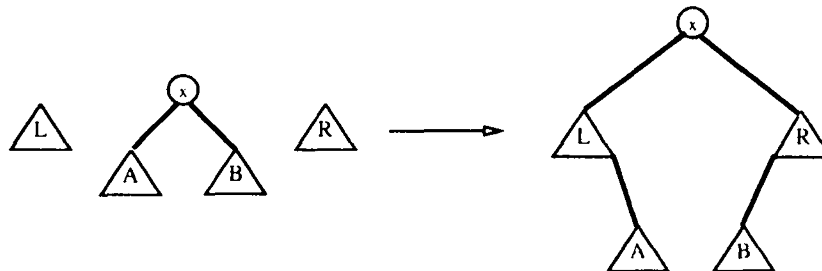


Figure 4.8 : La construction finale de l'arbre dans l'opération de Splay. Le nœud x contient la valeur recherchée.

Il existe plusieurs versions de l'algorithme de Splay. Parmi celles-ci, La version appelée Semi-Splay est donnée par Tarjan comme étant une des meilleures.
Le Semi-Splay diffère du Splay par la rotation ZIG-ZIG. Dans la version de base, l'arbre est obtenu à la fin de l'algorithme.
Dans le Semi-Splay, L'arbre est partiellement reconstruit à chaque ZIG-ZIG. Les arbres L et R sont conservés, mais nous avons en plus un arbre Top qui se construit au fur et à mesure. La version modifiée du ZIG-ZIG est présentée par la figure 4.9. Nous pouvons remarquer que dans cette version du Splay, l'élément concerné par l'opération ne deviendra pas forcément la racine de l'arbre.

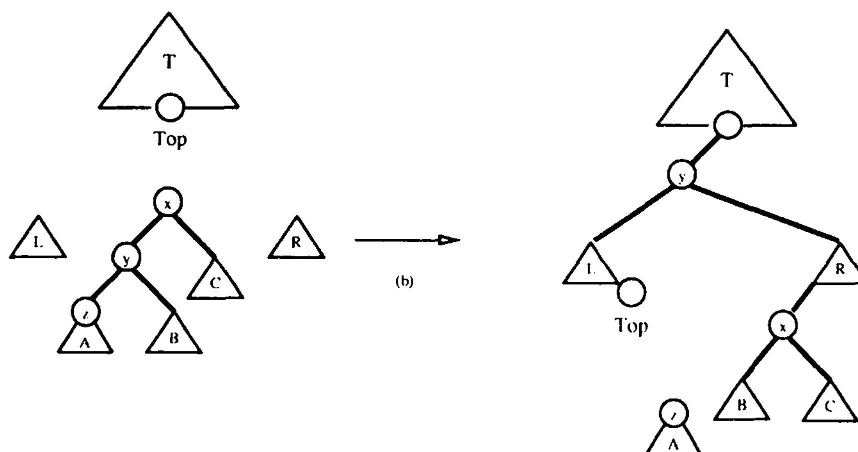


Figure 4.9 : ZIG-ZIG modifié du Semi-Splay. Le cas symétrique est omis.

L'algorithme de Semi-Splay étant assez difficile à implémenter, Tarjan

propose une modification qui ne change pas la complexité de l'algorithme. La simplification consiste à remplacer la rotation ZIG-ZAG par deux rotations ZIG. Le nouvel Algorithme est appelé Simple-Semi-Splay.

Dans le cas d'une utilisation des Semi-Splay-Tree et des Simple-Semi-Splay-Tree comme FP, nous considérons les nœuds de plus grande priorité comme étant ceux de plus petite valeur. L'algorithme du DeleteMin doit donc retirer le nœud de plus petite valeur. Pour cela, il doit donc parcourir la branche la plus à gauche de l'arbre tout en effectuant des rotations (ZIG-ZIG ou ZIG). Le dernier nœud trouvé est celui que l'on doit supprimer. L'opération de DeleteMin n'utilisant pas de rotation ZIG-ZAG, nous pouvons remarquer que dans le Semi-Splay et le Simple-Semi-Splay présentées ici, utilisent le même algorithme de DeleteMin.

L'algorithme de l'opération d'Insert doit ici parcourir l'arbre de façon à trouver l'emplacement du nœud à insérer tout en faisant les rotations.
Le Semi-Splay utilisera les rotations ZIG, ZIG-ZAG et ZIG-ZIG modifié.
Le Simple-Semi-Splay utilisera les rotations ZIG et ZIG-ZIG modifié.

Les figures 4.10 et 4.11 montrent des exemples de DeleteMin et d'Insert.

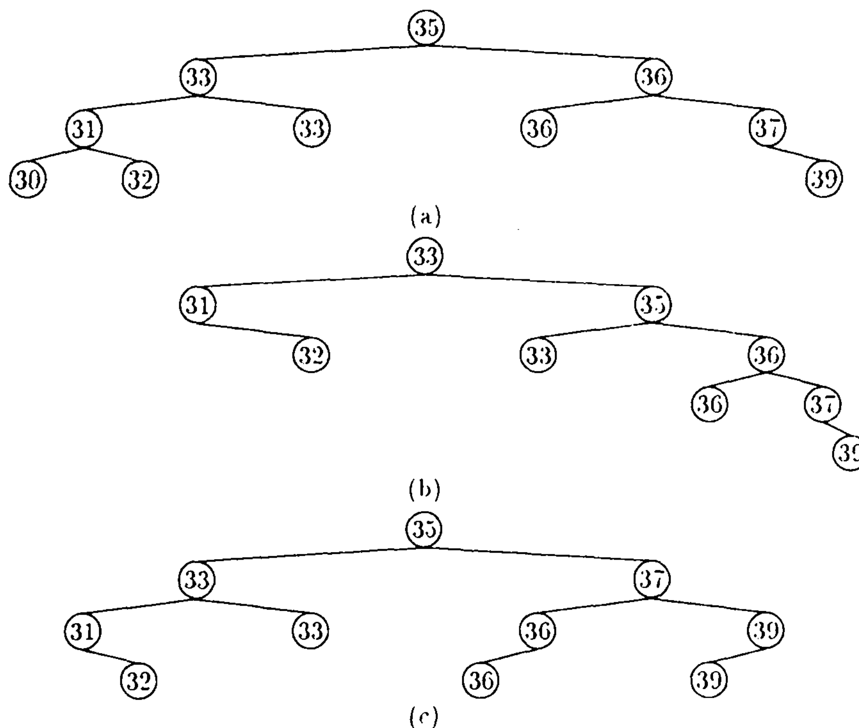


Figure 4.10 : Semi-Splay Tree

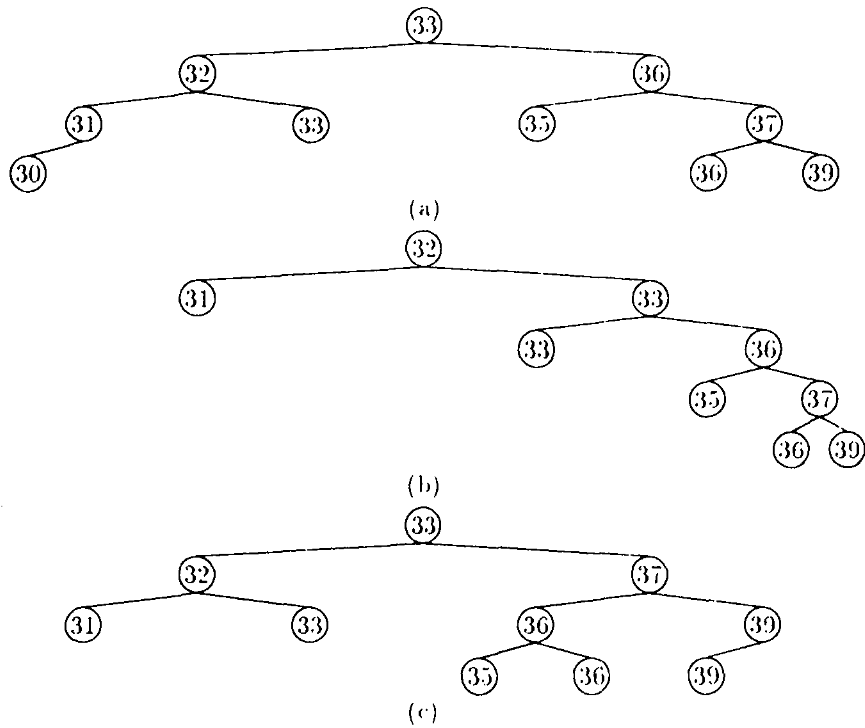


Figure 4.11 : Simple-Semi-Splay Tree

4.1.6 Single-Semi-Splay-Tree

Dans notre modèle, nous pouvons rencontrer plusieurs éléments de mêmes valeurs. Nous pouvons tirer parti de cette particularité pour réduire la taille de l'arbre. Et donc, de réduire la complexité des opérations de Semi-Splay.
En effet, le Semi-Splay-Tree ou le Simple-Semi-Splay-Tree sont des arbres binaires de recherche, donc l'emplacement dans l'arbre d'un nœud de valeur i est unique. Nous pouvons donc modifié les algorithmes de Semi-Splay et Simple-Semi-Splay pour que dans l'arbre, un nœud ne corresponde qu'à une valeur. Ces nouvels algorithmes sont appelés Single-Semi-Splay-Tree et Single-Simple-Semi-Splay-Tree.

Pour cela, une file d'attente est rattachée à chaque nœud de la structure. Le nombre d'éléments présents dans la file d'un nœud de valeur v désignera le nombre d'éléments de valeur v présents dans l'arbre de Splay. Les structures Single utilisent les mêmes algorithmes pour la phase de recherche que les "non Single". En revanche, ils diffèrent lors de l'insertion ou de la

suppression en elles-mêmes:
Lors d'un Insert d'un élément de valeur v, si le nœud i correspondant à v est dans l'arbre alors un élément de valeur v est inséré dans la file d'attente de i, sinon un nœud j de valeur v est inséré dans l'arbre, puis un élément de valeur v est inséré dans la file j.
Lors d'un DeleteMin, si la file du nœud i (i nœud de plus petite valeur) est non vide, on retire un élément de la file. Si après cette opération la file de i est vide, i est retiré de l'arbre.

Si le nombre maximal de priorité active est S_a, la complexité amortie des opérations de notre nouvelle structure est $O(log\ S_a)$.

Les figures 4.12 et 4.13 montrent des exemples de DeleteMin et d'Insert. Le nombre présents dans un carré à coté de chaque nœud désigne le nombre d'éléments de la file correspondante.

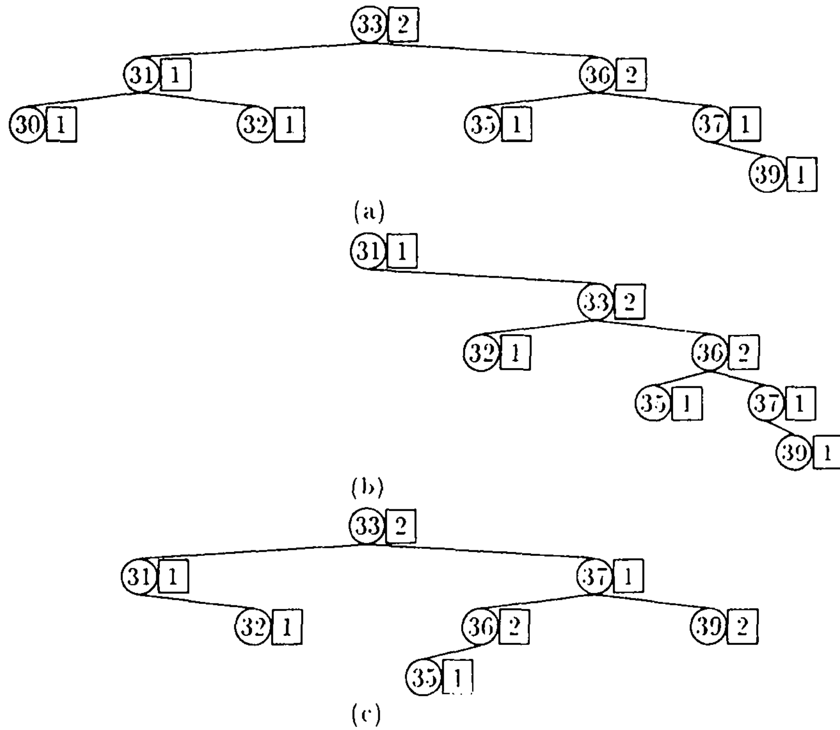


Figure 4.12 : Single-Semi-Splay Tree

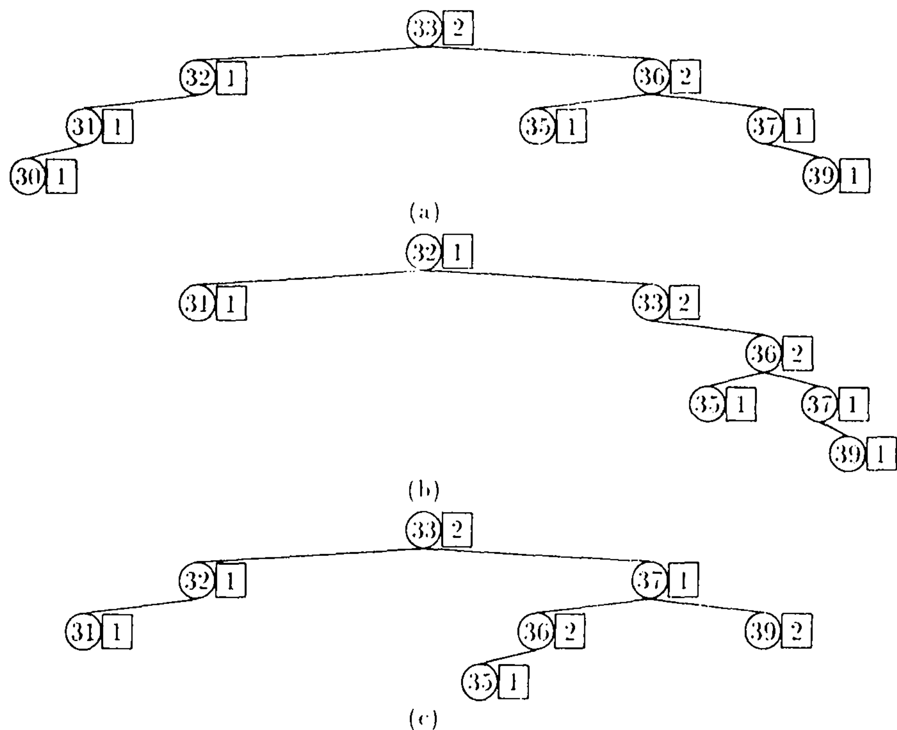


Figure 4.13 : Single-Simple-Semi-Splay Tree

4.1.7 Funnel-Tree

Le Funnel-Tree est une FP qui est spécifique au modèle du BB [MR90]. Elle tire parti du faible intervalle (Borne Inf..Borne Sup) dans lequel s'étendent les priorités.
Sa structure est très simple: elle se compose d'un tableau de S files d'attente (S représente la puissance de deux supérieure à la cardinalité de l'espace de priorités).
Un élément de valeur i sera présent dans la file d'indice $i - BI$.
L'accès à ce tableau ce fait par l'intermédiaire d'un arbre binaire complet de hauteur $log(S)$. Chaque nœud de l'arbre contient un entier qui définit le nombre d'éléments dans les files atteintes par le sous-arbre.

Lors d'un Insert d'un élément i, les valeurs des nœuds du chemin de la racine vers la file d'indice $i - BI$ sont incrémentées. Le parcours de la structure peut ici se faire soit dans le sens feuilles-racine soit dans le sens racine-feuille.

Le DeleteMin consiste à atteindre la file contenant les éléments de plus petite valeur.

Le parcours consiste donc en partant de la racine, à essayer d'aller le plus à gauche possible dans l'arbre tout en décrémentant la valeur de chaque nœud parcouru. Si pour un nœud i la valeur contenue dans son fils gauche est égale à 0, le parcours continuera avec le fils droit de i.

L'algorithme des opération d'Insert et de DeleteMin parcourt $log\ S$ nœuds. Ces opérations ont donc une complexité en $O(logS)$.

L'opération de DeleteGreater par rapport à une valeur v consiste à modifier les valeurs contenues dans l'arbre de façon à interdire l'accès aux files correspondant aux valeurs supérieures ou égales à v. Pour cela, l'arbre est parcouru à partir de la feuille associée à v jusqu'à la racine, en décrémentant les nœuds d'une valeur $nbelim$ désignant le nombre d'éléments à supprimer. La valeur $nbelim$ est initialisé au nombre d'éléments contenus dans la file de valeur v. Si pendant le parcours, un nœud i est un fils gauche, les éléments pouvant être atteints à partir du nœud j frère de i, doivent être éliminés. Donc $nbelim$ est incrémenté de la valeur contenue dans le noeud j. Cette opération a une complexité de $O(log\ S)$.

Nous pouvons remarquer que les éléments de mêmes valeurs étant contenues dans une file d'attente, la structure est stable.

4.1.8 Funnel-Table

Le Funnel-Table, comme le Funnel-Tree, se compose d'un tableau de S files d'attente. Mais l'accès aux files se fait directement.

L'Insert est en $O(1)$: l'élément de valeur i, est directement inséré dans la file d'indice $i - B1$.

L'algorithme de DeleteMin du Funnel-Table maintient une valeur B qui désigne l'indice de la file de plus petite priorité. Si la file est vide, le tableau est parcouru à partir de la file d'indice B, dans l'ordre décroissant des priorités jusqu'à trouver une file non vide. L'indice de cette file est affecté à la valeur B.
Cette méthode est possible car elle tire parti de la propriété de monotonie de notre modèle. En effet, nous sommes sûr que si une file d'indice i contient les éléments de plus grande priorité, les files d'indice inférieur à i sont et resteront vides.

L'algorithme de DeleteGreater par rapport à une valeur v consiste à parcourir les files à partir de celle associée à v en les vidant.

Lors de l'utilisatoin de cette structure pour un algorithme de BB, l'ensemble des opérations de DeleteMin plus celle des DeleteGreater a une complexité de $O(S)$.

Comme pour le funnel-Tree, les éléments étant contenus dans des files d'attente, la structure est stable.

4.2 Résultats expérimentaux

Nous avons expérimenté ces structures sur un SPARC muni du système SUNOS 4.1. Toutes les implémentations ont été réalisées en langage C. A part les D-heap qui utilisent un tableau statique, toutes les structures font appel aux primitives systèmes d'allocation mémoire pour la création et la suppression d'éléments.

Toutes les structures ont été testées avec le même jeux d'essai qui est le suivant:

Borne Inférieure 100,
Incrémentation Maximum 150,
Borne Supérieure varie de 700 à 1150,

La figure 4.14 présente le nombre de nœuds Maximum atteint dans la structure. La figure 4.15 présente le nombre d'opérations de DeleteMin ou d'Insert, pour chaque jeu d'essai. Nous pouvons remarquer que les progressions de ces deux courbes sont exponentielles.

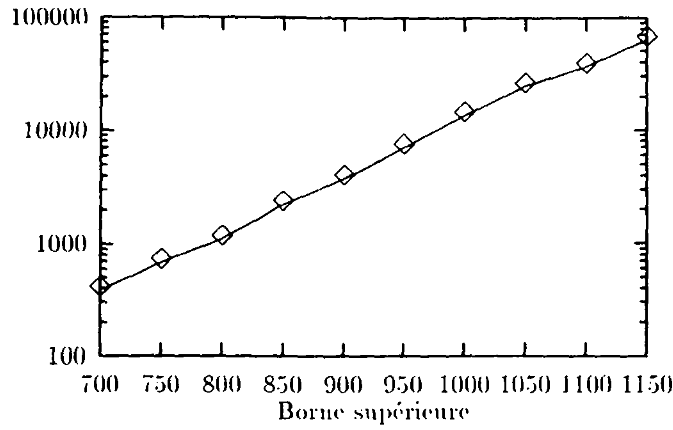


Figure 4.14 : Nombre maximal d'éléments présents dans la file

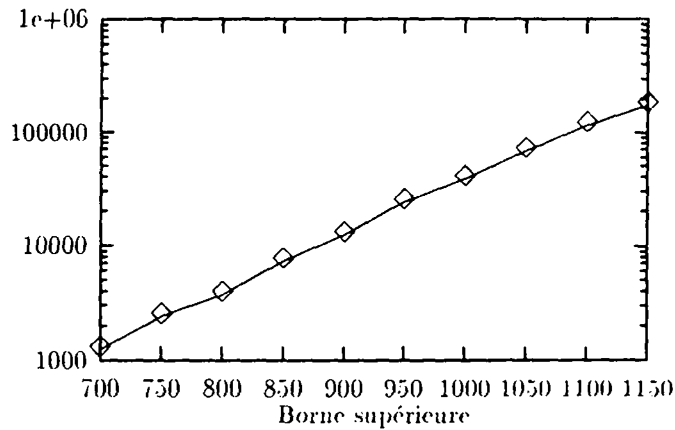


Figure 4.15 : Nombre total d'opérations.

Sur toutes les courbes suivantes, l'axe des ordonnées représentant le temps d'exécution en seconde est en échelle logarithmique, l'axe des abscisses représente les différentes bornes supérieures.

Bien que la version du D-heap avec 2 fils soit la plus utilisée, nous pouvons remarquer sur la figure 4.16 que les versions 3 et 4 fils sont un peu plus performantes.

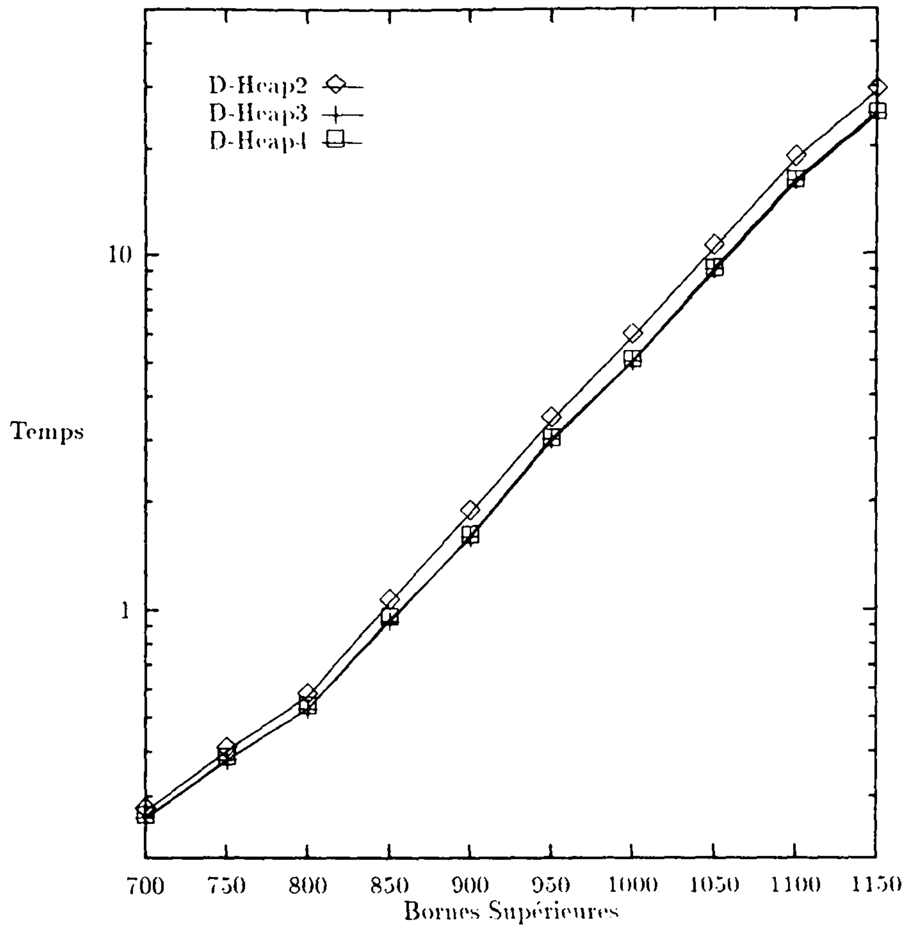


Figure 4.16 : Test des D-Heaps: 2, 3 et 4 Fils.

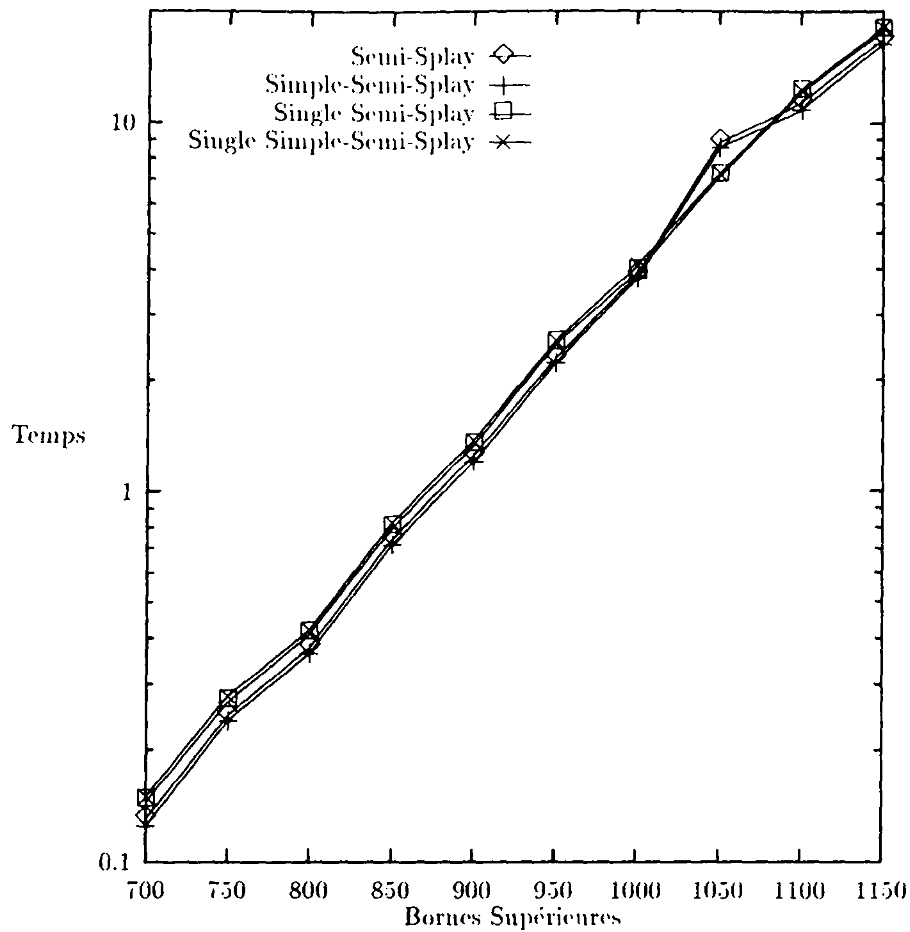


Figure 4.17 : Test des 4 Versions de Splay Tree.

Nous pouvons remarquer sur la figure 4.17 que les Simple-Semi-Splay sont équivalents aux Semi-Splay. Ce qui signifie que le fait d'avoir remplacer les rotations ZIG-ZAG par deux rotations ZIG, ne modifie que très peu les résultats. Cela semble vérifier les études théoriques de Tarjan. En revanche, cette modification simplifie la programmation de l'algorithme.
D'autre part, bien que les complexité théoriques soient différentes les résultats expérimentaux montrent que les algorithmes Single sont équivalents aux algorithmes originaux.
En effet, les évaluations très proches de la borne supérieure restreignent le nombre d'insertion effectuées. Les DeleteMin successifs déséquilibreront l'arbre vers la droite. Donc une opération de suppression reviendra à retirer la racine de l'arbre. Ceci sera èquivalent en complexité à une opération de suppression dans une file d'attente.

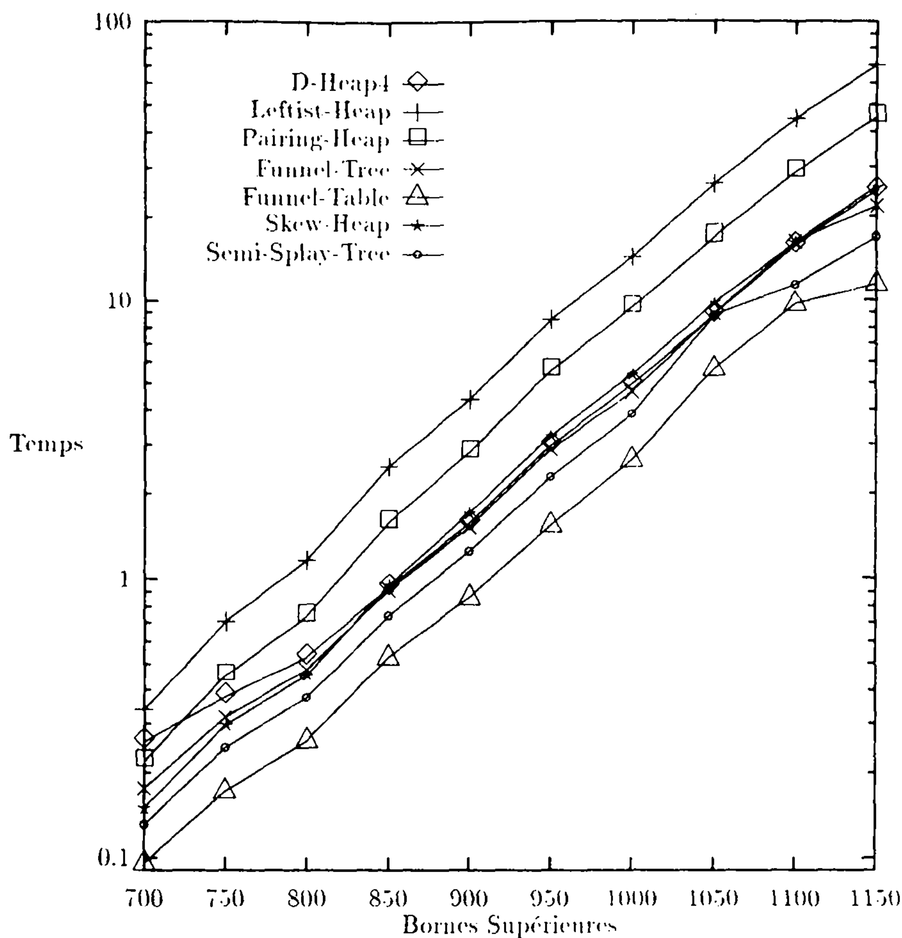


Figure 4.18 : Test de toutes les FP.

Comme nous l'avons dit précédemment le Skew-Heap est la version auto-ajustable du Leftist-Heap. Nous pouvons remarquer sur la figure 4.18 que le surcoût des tests d'équilibrage du Leftist-Heap le rend moins performant que le Skew-Heap.

Les opérations du Funnel-Tree sont en $O(log\ S)$ quelque soit le nombres d'éléments présents dans la structure. Cette remarque explique le fait qu'il soit équivalent au Skew-Heap.

Ayant des complexités très intéressantes quant à ces opérations, il semble normal que le Funnel-Table s'avère aussi efficace en pratique qu'il le laissait supposer en théorie.

De même, l'intérêt d'utiliser les arbre binaires de recherche comme les Semi-Splay-Tree en tant que file de priorité semble être non négligeable dans notre modèle.

Le tableau ci-après présente une comparaison de toutes ces structures.

FP	Insert	DeleteMin	DeleteGreater	Perf	Stab	Mon	Mem	Prog
Funnel-Table	O(1)	O(1)	O(1)	1	o	o	-	++
Splay-Tree	O(log n)	O(log n)	O(log n)	2	o	n	+	
Funnel-Tree	O(log S)	O(log S)	O(log S)	3	o	n		++
Skew-Heap	O(log n)	O(log n)	O(n)	3	n	n	+	+
D-Heap	O(log n)	O(log n)	O(n)	3	n	n	+	++
Pairing-Heap	O(1)	O(log n)	O(n)	6	n	n	+	+
Leftist-Heap	O(log n)	O(log n)	O(n)	7	n	n	+	+

Légende :

DeleteGreater : les complexités n'incluent pas la libération de la place mémoire prise par les éléments à élaguer.
Perf : performance séquentielle.
Stab : stabilité (oui/non).
Mon : besoin de la Monotonie (oui/non).
Mem : ressource mémoire.
Prog : facilité de programmation.
++ : trés bonne.
+ : bonne.
- : mauvaise.
– : très mauvaise.

Chapitre 5

Méthodologie des accès aux files de priorité

Comme nous l'avons dit précédemment, les machines parallèles posent un certain nombre de problèmes lors de l'implémentation des algorithmes.
L'accès simultané par des processeurs asynchrones, à une structure de donnée résidant en mémoire partagée en est un.
Une approche simple consiste à manipuler les structures de façon exclusive. Un processus voulant réaliser une opération de base d'une structure, interdit à tout autre processus l'accès à cette structure. Cette approche, bien que très facile à implémenter, implique un speed-up limité.

Dans le cas des structures arborescentes, une autre approche consiste à permettre l'accès à la structure à plusieurs processus simultanément. Un processus ne doit pas modifier une partie de la structure sans en avoir prévenu les autres, la structure risquant de devenir inconsistante. Une solution appelée **verrouillage local** (partial locking) consiste à assurer une exclusion mutuelle entre les processus, pour chaque entité modifiable de la structure. L'exclusion mutuelle sera assuré par des primitives de synchronisation que nous appellerons "verrou". Lorsqu'un processus voudra réaliser une opération de base sur une structure, il "verrouillera" les parties de la structure qu'il veut modifier.

L'utilisation de verrou pose des problémes:
En effet, les processus ne peuvent verrouiller une partie de la structure sans obéir à certaines régles. L'utilisation de verrous peut entrainer l'interblocage des processus. De plus, si le protocole d'exclusion mutuelle n'est pas équitable, certains processus risquent d'être bloqués éternellement, et donc d'être en état de famine.

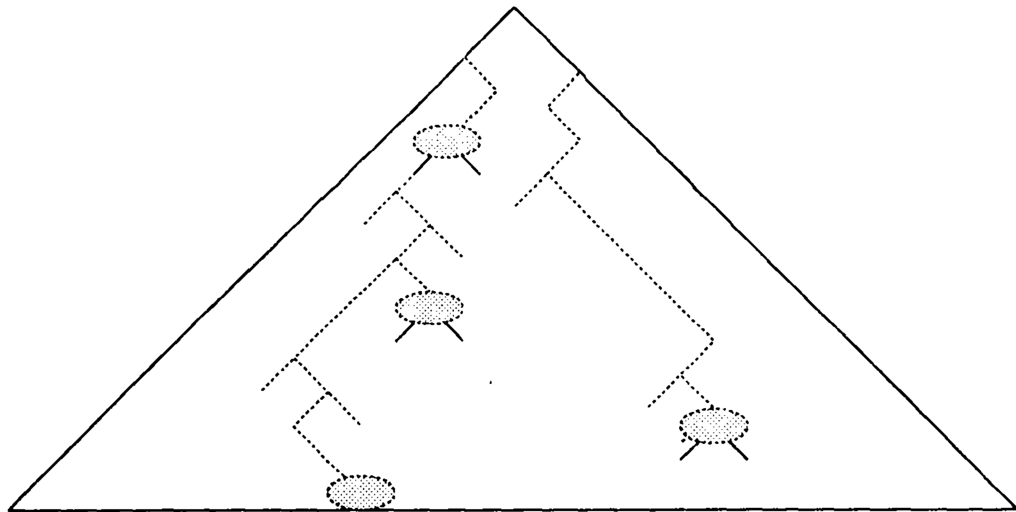


Figure 5.1 : Structure parcourue en Partial Locking.

Dans une première section nous allons exposer deux méthodologies qui, dans le cas de la parallélisation des opérations de base sur une FP, assurent l'inexistence d'interblocage.

L'utilisation de verrous implique un surcout. L'exclusion mutuelle résolue par des moyens logiciels ou matériels se réalise en un temps non négligeable. Nous verrons dans une deuxième section, que la quasi totalité des primitives peuvent être remplacée par des simples attentes sur des marquages. Nous illustrerons cette optimisation à travers les parallélisations existantes du Skew-Heap et du Funnel-Tree.

5.1 Méthodologie avec verrou

Dans la littérature, la partie de l'arbre verrouillée par un processus est appelée **fenêtre** ou **bulle** (Window ou Bubble).

Définition 4 *La fenêtre d'un processus réalisant une opération de base sur une structure arborescente, est définie comme étant l'ensemble des nœuds de cette structure sur lesquels le processus a un verrou.*

Pour éviter les interblocages de processus, il suffit que ces derniers, lors de la réalisation d'une opération de base, obéissent à un schéma de verrouillage qui interdise la création de cycle de processus attendant sur des ressources.

Nous proposons deux schémas de ce type, s'appliquant aux structures arborescentes dont toutes les opérations de base parcourent l'arbre dans un même sens: soit le sens feuilles-racine, soit le sens racine-feuilles.
Le principe de ces schémas est de permettre aux processus, la construction et le déplacement de fenêtres, pendant la durée d'une opération de base, sous certaines contraintes assurant l'inexistence d'interblocage.

Nous supposons que le protocole d'exclusion mutuelle utilisé pour l'implémentation des verrous est équitable: si un processus demande l'accès à une ressource, il est sûr de l'obtenir dans un temps fini.
De plus, dans le cas des FP, les primitives de synchronisation seront d'un seul niveau (voir chapitre 2).
Les schémas de verrouillage présentés supposent qu'un processus n'essayera pas de verrouiller un nœud sur lequel il détient déjà un verrou.

Pour toutes les démonstrations, nous utiliserons les notations suivantes:
Nous notons par S une structure arborescente dont les opérations de base se réalisent toutes dans le même sens.
$\mathcal{N}$ est l'ensemble des nœuds composants S.
r est la racine de S.
Nous désignons par $\mathcal{P} = \{P_1, \ldots, P_p\}$ l'ensemble des p processus pouvant réaliser des opérations de base sur une structure S.
Un antécédent de x est défini comme étant un sommet appartenant au chemin (unique) entre x et la racine.

Définition 5 *Soit une structure arborescente S dont toutes les opérations de base sont réalisées dans le sens racine-feuilles (resp: sens feuilles-racine). Le schéma 1 est défini par les contraintes suivantes:*

Verrouillage *: un processus peut verrouiller un nœud x de S.*

- **Cas 1** *: ssi il détient le verrou du nœud père (resp: d'un des nœud fils) de x.*
- **Cas 2** *: un processus peut verrouiller la racine (resp: une feuille) de S, ssi il ne détient plus de verrou dans S.*

Déverrouillage *: un processus peut déverrouiller un nœud x de S, ssi il ne détient pas de verrou sur le père de x, (resp: sur un des fils de x).*

La fenêtre que génère le schéma 1 dans la sens feuille-racine, recouvre un sous-arbre connexe de l'arbre. Plus précisement, il recouvre un chemin de

l'arbre, le schéma fait déplacer la fenêtre vers la racine.
Dans le sens racine-feuille, la fenêtre recouvre initialement la racine, puis elle peut s'étendre, pour recouvrir un ou plusieurs sous-arbres de la structure.

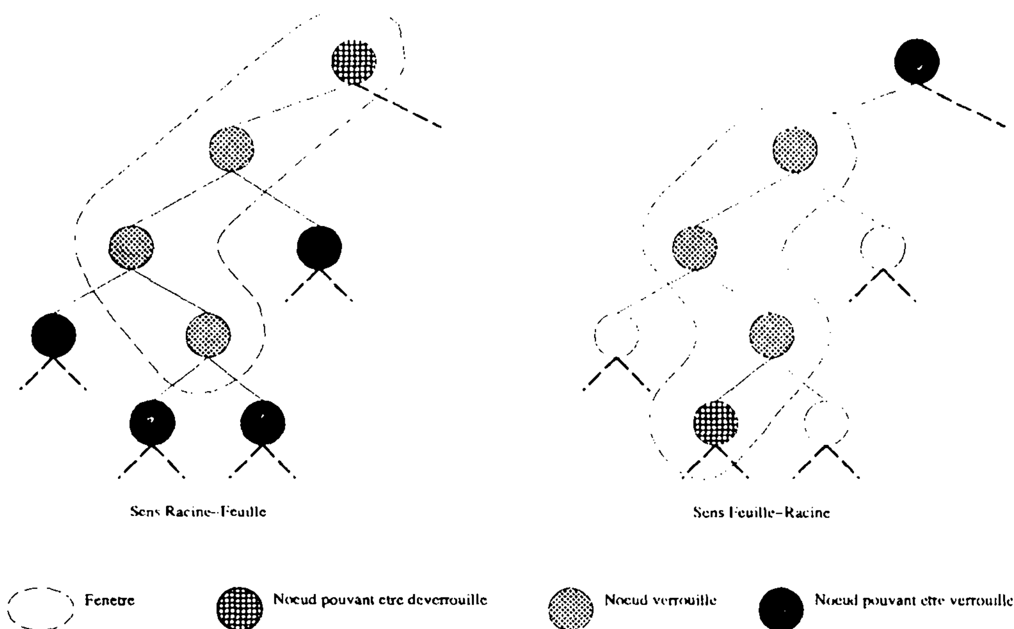


Figure 5.2 : Exemple de fenêtre pour le schéma 1.

Une variation de ce schéma a été présenté dans [MR90] où il a été utilisé pour la parallélisation du Skew-Heap et du Funnel-Tree.
De même, Jones, pour une autre parallélisation du Skew-Heap utilise un schéma similaire [Jon89].

Nous dégageons plusieurs propriétés, nous permettant de définir les caractéristiques de notre schéma.

Lemme 1 *Le schéma 1 assure que pendant la durée d'une opération de base sur S de sens racine-feuille (resp:feuille-racine), le verrouillage de la racine (resp: d'une feuille) correspond au premier verrouillage possible de la structure S.*

Preuve: La preuve est immédiate par le cas 2 du schéma. □

Lemme 2 *Soit x un nœud de la structure S, avec toutes les opérations de base se réalisant dans le sens racine-feuille (resp:feuille-racine).*
Soit P_i appartenant à $\mathcal{P}$, détenant un verrou sur x et ne détenant aucun

verrou sur tout descendant (resp: antécédent) de x.
Par le schéma 1 si P_i veut un verrou sur un nœud y descendant (resp:antécédent) de x, alors il devra verrouiller, par étapes successives, tous les nœud du chemin de x à y.

Preuve: Dans le sens racine-feuille:
Si P_i veut posé un verrou sur y, alors par le cas 1 de schéma, il doit d'abord verrouiller le père de y. De même, pour verrouiller le père de y il doit d'abord verrouiller le grand père de y. Ceci jusqu'à x. Donc si P_i veut un verrou sur y, il doit d'abord détenir le verrou de chaque nœud composant le chemin de x à y.
Dans le sens feuille-racine: La démonstration dans ce sens ce fait de manière analogue. □

Lemme 3 *Soit la structure S, avec toutes les opérations de base se réalisant dans le sens racine-feuille. Soit x un antécédent de y, avec x et y appartenant à S.*
Le schéma 1 assure que si P_i détient un verrou sur x et si P_j détient un verrou sur y, alors P_i a verrouillé la racine après P_j.

Preuve: Suivant le lemme 1, initialement, P_i et P_j n'ont de verrou sur aucun nœud de S.
Si P_i a un verrou sur x alors par le lemme 2, il a du détenir un verrou sur chacun des nœuds antécédents de x.
Considérons que P_j a détenu un verrou sur la racine après P_i.
Si P_j a un verrou sur y alors par le lemme 2, il a du détenir un verrou sur chacun des nœuds antécédents de y.
Or x est un antécédent de y donc, si P_i a un verrou sur x, P_j n'a pas pu verrouiller tous les antécédents de y. Donc P_j ne poura pas avoir de verrou sur y.
Nous pouvons en déduire que si P_i a un verrou sur x et si P_j a un verrou sur y alors il est impossible que P_i ait détenu un verrou sur la racine de S avant P_j. □

L'ordre de verrouillage de la racine par les processus est conservé tout au long d'un chemin descendant.

Lemme 4 *Soit la structure S, avec toutes les opérations de base se réalisant dans le sens racine-feuille (resp:feuille-racine). Soit x un antécédent (resp: descendant) de y, avec x et y appartenant à S.*

Par le schéma 1 si un processus P_i a un verrou sur x et y, alors il n'existe pas de processus P_j $(i \neq j)$, ayant un verrou sur un nœud z appartenant au chemin de x à y.

Preuve: Si P_i a un verrou sur x et P_j a un verrou sur z (avec x antécédent de z), alors par le lemme 3, P_j a détenu un verrou sur la racine de S avant P_i.
De même, si P_j a un verrou sur z et P_i a un verrou sur y (avec z antécédent de y), alors par le lemme 3, P_i a détenu un verrou sur la racine de S avant P_j.
Nous avons une contradiction, donc si P_i détient un verrou sur x et y alors il n'existe pas de processus P_j $(i \neq j)$, ayant un verrou sur un nœud z appartenant au chemin de x à y. □

Il n'existe pas d'intersection des chemins recouverts par les fenêtre de différents processus.

Propriété 1 *Le schéma 1 assure l'inexistence d'interblocage des processus l'utilisant durant une opération de base sur la structure S.*

Preuve: Nous allons démontrer que pour tout x et y appartenant à $\mathcal{N}$ et pour tout P_i et P_j appartenant à $\mathcal{P}$, avec $i \neq j$, si P_i a un verrou sur x et P_j a un verrou sur y, alors P_i ne peut demander un verrou sur y alors que P_j demande un verrou sur x.

Sens Feuilles-Racine . Dans les deux cas suivants:

x antécédent de y.

Si P_i veut mettre un verrou sur y, il doit détenir le verrou d'un des fils z de y, donc P_i doit avoir les verrous de x et z et P_j le verrou de y. Par le lemme 4 ce cas est impossible, donc il ne peut pas se créer d'interblocage dans ce cas.

il n'existe pas de chemin entre x et y.

Si P_i veut mettre un verrou sur y, alors par le lemme 2, il a du verrouiller une feuille z, descendante de y. Or il détient un verrou sur x, dont les feuilles descendantes sont différentes de z.
Le cas 2 du schéma a donc été violé. Donc il ne peut pas se créer d'interblocage dans ce cas.

Sens Racine-Feuilles . Deux cas se présentent à nouveau:

- x antécédent de y.

  Si P_j veut mettre un verrou sur x, il doit alors détenir le verrou de z, père de x.

  Nous avons donc la configuration où P_j détient un verrou sur z et y sachant que P_i détient un verrou sur x (x appartenant au chemin de z à y).

  Par le lemme 4, ce cas est impossible. Donc il ne peut pas se créer d'interblocage dans ce cas.

- il n'existe pas de chemin entre x et y.

  Si P_i veut posé un verrou sur y, il doit alors avoir un verrou sur le père de y. Or P_j ayant un verrou sur y, par le lemme 3: P_i a verrouillé la racine après P_j.

  De même, si P_j veut posé un verrou sur x, il doit alors avoir un verrou sur le père de x. Or P_i ayant un verrou sur x, par le lemme 3: P_i a verrouillé la racine avant P_j.

  Nous arrivons donc à une contradiction. Donc il ne peut y avoir d'interblocage dans ce cas.

Pour chaque sens, la généralisation à n processus se fait de meme. □

Dans le sens racine-feuilles uniquement, nous pouvons donner un autre schéma moins contraignant.

Définition 6 *Soit une structure arborescente S dont toutes les opérations de base sont réalisées dans l'unique sens racine-feuilles. Le schéma 2 est défini par les contraintes suivantes:*

Verrouillage *: un processus ne peut verrouiller un nœud x de S, que*

- **Cas 1** *: s'il détient le verrou du nœud père de x.*
- **Cas 2** *: s'il détient déjà le verrou d'un nœud antécédent x et un nœud descendant de x.*
- **Cas 3** *: Un processus peut verrouiller la racine r de S, ssi il ne détient plus de verrou dans S.*

Déverrouillage *: il n'y a aucune contrainte sur le déverrouillage.*

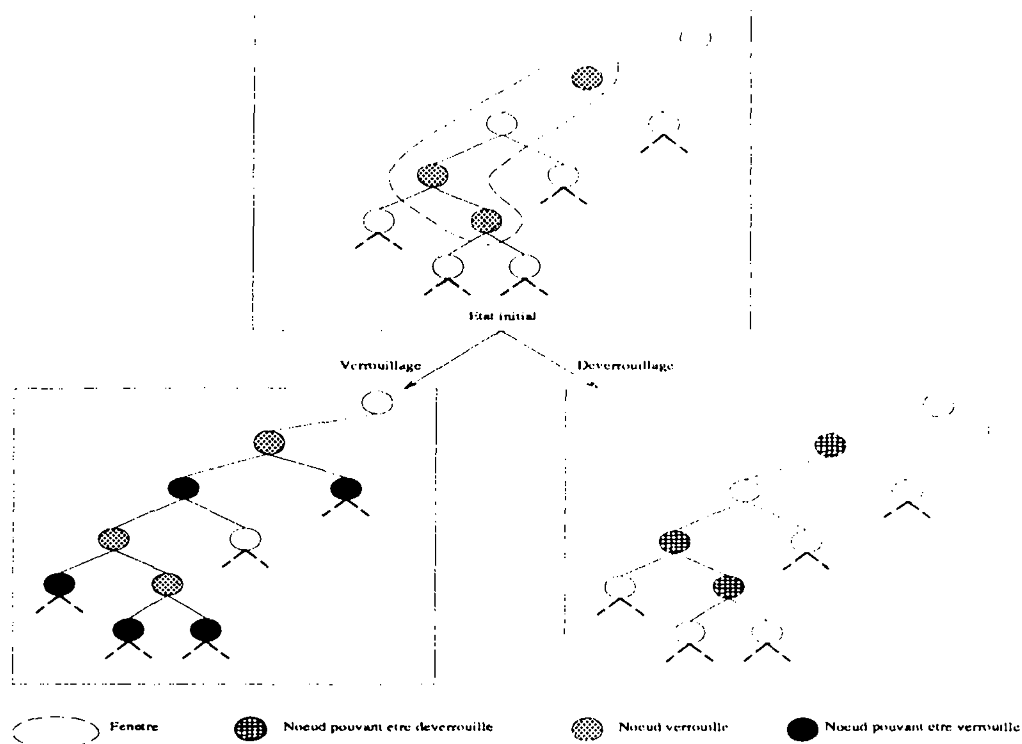


Figure 5.3 : Exemple de fenêtre pour le schéma 2

Contrairement au schéma 1, le schéma 2 permet l'existence de nœuds non verrouillés, appartenant au chemin reliant des nœuds verouillés par un processus (figure 5.3).

Les lemmes 1, 2, 3 et 4 appliqués pour les opérations de sens racine-feuille, sont démontrables de la même façon pour le schéma 2 que pour le schéma 1.

Propriété 2 *Le schéma 2 assure l'inexistence d'interblocage des processus l'utilisant durant une opération de base sur la structure S.*

Preuve: Nous allons démontrer que pour tout x et y appartenant à $\mathcal{N}$ et pour P_i et P_j appartenant à $\mathcal{P}$ avec $i \neq j$, si P_i a un verrou sur x et P_j a un verrou sur y alors P_i ne peut demander de verrou sur y alors que P_j demande un verrou sur x. Deux cas se présentent:

x antécédent de y.

Si P_j peut demander un verrou sur x alors,

soit il a un verrou sur le père de x (Cas 1).

soit il a un verrou sur un nœud z antécédent de x et un sur un descendant de x (Cas 2).

Les deux cas reviennent à la même configuration où le processus P_j à un verrou sur les nœuds z et y et le processus P_i a un verrou sur x (avec x appartenant au chemin de z à y). Par le lemme 4, cette configuration est impossible donc il ne peut pas se créer d'interblocage dans ce cas.

il n'existe pas de chemin entre x et y.

Si P_i demande un verrou sur y alors deux cas se présentent:

P_i a un verrou sur le père de y (Cas 1).

Si P_i veut posé un verrou sur y, il doit alors avoir un verrou sur le père de y.

Or P_j ayant un verrou sur y, par le lemme 3: P_i a verrouillé la racine après P_j.

De même, si P_j veut posé un verrou sur x, il doit alors avoir un verrou sur le père de x. Or P_i ayant un verrou sur x, par le lemme 3: P_i a verrouillé la racine avant P_j.

Nous arrivons donc à une contradiction. Donc il ne peut y avoir d'interblocage dans ce cas.

P_i a un verrou sur un nœud z antécédent de y et un sur un nœud z' descendant de y (Cas 2).

Par le lemme 4, cette configuration est impossible donc il ne peut pas se créer d'interblocage dans ce cas.

La généralisation à n processus se fait de même. □

Des parallélisations utilisant ce deuxième schéma sont exposées dans le chapitre suivant.

Le but de ces schémas est de proposé des méthodologies assurant l'inexistence d'interblocage des processus. La consistance d'une structure dépendra de la façon dont elles sont utilisées pour un algorithme particulier.

Les performances de la structures en parallèle dépendront de la taille de la fenêtre utilisée. Plus le nombre de nœuds verrouillés dans une fenêtre est important, plus le nombre de processus pouvant être présents à un instant donné dans la structure sera petit. Ceci augmentera le temps d'attente des autres processus désirant accéder à la structure.

5.2 Méthodologie avec marquage

Un facteur important à prendre en compte, lors de l'élaboration de structure concurrente est le surcoût qu'entraine l'utilisation de primitives de synchronisation ou de parcours redondants de la structure afin d'en assurer la consistance.
Les primitives de synchronisation présentes sur beaucoup de machines commerciales apportent des solutions matérielles ou logicielles au problèmes de l'exclusion mutuelle. Mais elles ont un temps d'exécution non négligeable.
Les algorithmes d'exclusion mutuelle consistent à assurer qu'un seul processus obtienne une ressource alors que plusieurs en ont fait la demande. Nous montrons que dans le cas d'opérations réalisées dans le sens racine-feuilles, les primitives de synchronisation peuvent être remplacées par des simples mécanismes de marquages.

Théorème 1 *Considérons le contexte d'opérations concurrentes de sens Racine-feuilles, réalisées sur une structure arborescente S, selon l'un des deux schémas énoncés précédemment.*
Soit x un nœud quelconque de S, si un processus P_i détient un verrou sur x, alors quelque soit y un nœud fils de x, P_i est le seul processus a pouvoir demander un verrou sur y.

Preuve: Nous allons démontrer par l'absurde qu'il n'existe pas de processus P_j ($j \neq i$) pouvant faire une demande de verrou sur l'un de fils de x.
En effet, si P_j peut demander le verrou d'un nœud fils y de x alors
- soit P_j a verrou sur le père de y, ce qui est impossible car P_i le détient,
- soit il détient un nœud ancêtre de y et un nœud appartenant à la descendance de y (deuxième schéma), le lemme 4 montre que cette configuration est impossible.
Donc P_i est le seul processus a pouvoir demander un verrou sur un des fils de x. □

Ce théorème implique que si on utilise un des schémas de verrouillages énoncés plus haut dans le sens racine-feuilles alors, sauf pour la racine de l'arbre, nous pouvons utiliser un simple mécanisme de marquage.
Le marquage d'un nœud x consistera à attendre que x ne soit plus marqué, puis à le marquer. Si un processeur n'a plus l'utilité d'un nœud, il lui suffira de retirer la marque.
Le nœud racine n'a pas de père donc dans ce cas, plusieurs processus peuvent essayer de le verrouiller. Un algorithme d'exclusion mutuelle devra donc être utilisé pour ce nœud particulier.

5.3 Comparaison des méthodologies pour deux files de priorité

Tous les algorithmes présentés ici ont été implémentés en langage C sur une SEQUENT comportant 9 processeurs. Le système de cette machine appelé DYNIX est un UNIX parallèle compatible BSD4.3. La gestion des processus, les primitives de synchronisation ainsi que la gestion de la mémoire partagée, ont été réalisée à l'aide des librairies standards de DYNIX.

Le jeux d'essai est le suivant:

Borne Inférieure 100.
Borne Supérieure 1000.
Incrémentation Maximale 150.
Nombres de processeurs Varie de 1 à 9.

Les algorithmes utilisés pour les tests avec un processeur sont les mêmes que ceux utilisés en séquentiel. Les primitives ou les marquages ont été retirés du programme pour cette exécution.

5.3.1 Skew-Heap

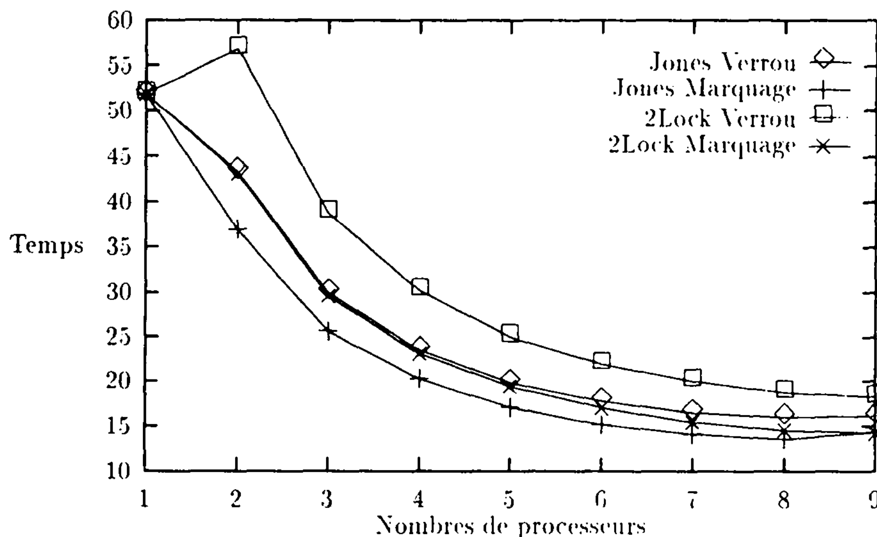


Figure 5.4 : Test sur les différents Skew-Heap

La version concurrente du Skew-Heap présentée par Jones est récursive [Jon89]. Afin d'obtenir de meilleurs résultats, nous l'avons rendu itérative.

Une version concurrente du Skew-Heap appelée 2Lock est également présentée ici. Le but de ce nouvel algorithme est d'augmenter la concurrence. Pour cela, nous utilisons deux verrous par nœud: un sur la référence du fils gauche et l'autre sur celle du fils droit. Sur un nœud, un processus pourra modifier la référence droite, pendant qu'un autre pourra modifié celle de gauche. Comme on peut le voir sur la figure 5.4, le Skew-Heap de Jones semble "saturer" à 8 processeurs. L'algorithme concurrent ne parait pas permettre la présence de plus de 9 processeurs dans la structure.
La version 2Lock ne semble pas "saturer" avec 8 processeurs. Malheureusement, le surcoût de l'utilisation d'un deuxième verrou par nœud semble la rendre moins performante.

La figure 5.4 montre bien que le remplacement des primitives de synchronisation par le mécanisme de marquage réduit le surcout et donc donne de meilleurs résultats. D'où l'équivalence des résultats entre les deux versions (Jones, 2Lock) avec 9 processeurs. Il serait intéressant d'étudier le comportement de ces deux structures avec un nombre supérieur de processeurs. La version 2Lock saturera aussi, mais sa plus grande concurrence devrait lui permettre d'être plus performante.

5.3.2 Funnel-Tree

L'algorithme concurrent du Funnel-Tree présenté ici, est celle de B. Mans et C. Roucairol [MR90]. La différence entre la version avec verrou et celle avec marquage est flagrante. La figure 5.5, nous montre l'intérêt de ce changement. En particulier, avec deux processeurs, le surcoût dû aux primitives de synchronisation est supérieur au temps gagné par la concurrence. L'utilisation de marquages par contre, ne provoque pas ce phénomène.

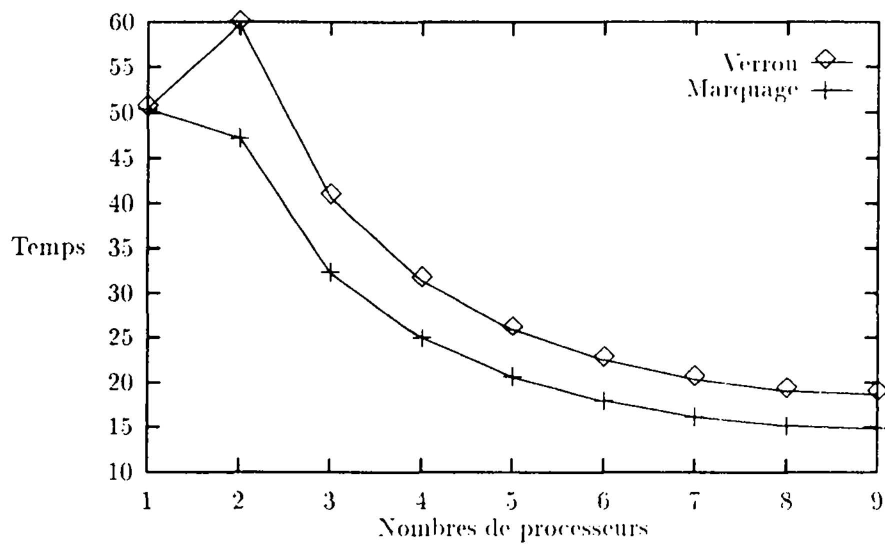


Figure 5.5 : Test sur les différents Funnel-Tree

Chapitre 6

Nouveaux algorithmes concurrents pour l'accès à d'autres files de priorité

Nous allons dans ce chapitre, présenter la parallélisation du Funnel-Table et des différents Semi-Splay-Tree.

6.1 Funnel-Table

Le Funnel-Table n'est pas une structure arborescente, donc les méthodologies présentées précédemment ne peuvent s'appliquer.
Le principe de l'algorithme séquentiel est d'accéder directement à la file d'attente considérée. Pour assurer la consistance de la structure, il faut assurer une exclusion mutuelle entre les processus accédant à une file particulière. Nous garantissons la consistance d'une file, en utilisant une primitive de synchronisation: un verrou. Chaque file du tableau pourra être verrouiller séparément.

Nous désignons par f_t la file d'attente d'indice t, et par $\mathcal{P} = (P_1, \ldots, P_p)$ les p processus susceptibles d'accéder à la structure.

Lors d'un Insert, le processus verrouillera la file considérée f_t, effectuera une insertion séquentielle dans f_t, puis déverrouillera f_t.

Considérons maintenant l'opération de DeleteMin. Dans l'algorithme séquentiel, une variable B désigne l'indice de la file f_B , où se trouve les éléments de plus grande priorité. Dans le cas où, il existe un élément dans

f_B, il suffit d'effectuer la suppression suivant le meme schéma que l'Insert. En revanche, si f_B est vide, une remise à jour de B doit etre effectuée. Dans notre solution, la variable B sera partagée par tous les processus p appartenant à $\mathcal{P}$, afin que ces derniers soient informés du nouvel indice de la file possédant les éléments de plus grande priorité.
Un processus P_i voulant réalisé un DeleteMin, et trouvant f_B vide ne peut, comme en séquentiel, affecter l'indice de la prochaine file non-vide à B.
En effet, comme les processus travaillent de façon asynchrone, nous pouvons considérer un processus P_j ayant une insertion à effectuer sur la file f_{B+1} d'indice $B+1$. Si P_i teste la file f_{B+1} avant que P_j n'est pu insérer son élément e , alors e sera perdu.

Afin de résoudre ce problème, nous considérons qu'une file f_i est "définitivement vide", ssi tous les processus ont fini de séparer tous les nœuds de priorités inférieures ou égales à celle associée à f_i.
Pour cela, nous ajoutons à la structure de file un champs entier $NbProc$ initialement à 0. Un processus P_i effectuant un DeleteMin sur une file f_i, incrémentera le champs $NbProc$ de f_i. Il le décrémentera lorsqu'il aura fini la génération des nœuds fils. Si tous les processus accédant à la structure utilisent cette algorithme, la structure restera alors consistante. En effet, soient les processus $P_i, P_j \in \mathcal{P}$, supposons que P_i ait pris par un DeleteMin, le dernier élément de la file f_B, et P_j veut réaliser un DeleteMin sur f_B.
Comme f_B est vide, P_j va essayer de trouver la prochaine file définitivement vide en parcourant le tableau à partir de f_B. Or le champs nb de f_B n'est pas nul, donc P_j ne pourra modifier B.
La structure restera donc consistante. Ce meme processus P_j ne pouvant remettre à jour B, n'a pas de travail à effectuer. Nous pouvons considérer que P_j peut s'en procurer lui-même, en allant chercher la première file non vide.

Cette modification de l'algorithme change la complexité de l'opération de DeleteMin. En effet, pour l'algorithme séquentiel, la complexité de l'ensemble des DeleteMin plus les DeleteGreater était $O(S)$. En revanche pour l'algorithme parallèle, le pire des cas se présente lorsqu'un processus P_i effectue un DeleteMin sur la première file, puis ne fait plus rien. Les autres processus, à chaque DeleteMin réalisé, vont être obligé de parcourir une importante partie du tableau, afin de trouver du travail. Dans le pire des cas l'opération de DeleteMin va donc être en $O(S)$.

6.2 Splay-Tree

Il nous a paru intéressant d'étudier une version concurrente des Splay-Tree car en effet, bien qu'étant, dans notre cas moins performant que le

Funnel-Table, il n'a pas besoin de la monotonie du modèle du GBB. De plus la stabilité et un DeleteGreater efficace, le rend plus intéressant que les tas.

Les opérations se réalisant dans le sens Racine-feuilles, nous allons utiliser le schéma de verrouillage de la définition 6.
Soit un processus P_i, voulant réaliser une insertion d'un élément w dans un Semi-Splay-Tree appelé S.

L'algorithme séquentiel d'Insert du Splay-Tree, dans sa version Semi-Adjusting doit maintenir trois arbres: T, L et R.
A chaque rotation ZIG-ZIG rencontrée, l'arbre T est remis à jour (figure 4.9) et le nœud Top auquel les arbres R et L se rattacheront doit être modifié.

Afin de garantir la consistance de la structure, P_i doit verrouiller le nœud Top pendant qu'il construit les arbres R et L. Au début du parcours l'arbre T est vide, donc le verrouillage du nœud Top sera remplacé par le verrouillage du pointeur de référence de S (le pointeur sur la racine de S). Lors du parcours pour rechercher la place de w, chaque nouveau nœud parcouru y doit être verrouillé pour s'assurer qu'il ne pourra plus être modifié. Dans chaque cas de rotation, nous allons décrire les différents verrouillages et déverrouillages de notre algorithme concurrent.
Nous reprendront les noms donnés au nœuds de la figure 4.7 pour les rotations ZIG et ZIG-ZAG, et ceux de la figure 4.9 pour la rotation ZIG-ZIG.
Nous exposerons que les sens de rotations des figures, les sens symétriques étant exécutés de même.

ZIG : le nœud x est verrouillé,
- si le nœud y n'existe pas alors la place de w est trouvée,
- x devient le nœud le plus à gauche de R,
- x est déverrouillé,

ZIG-ZAG : x est verrouillé,
- y est verrouillé,
- si z n'existe pas alors la place de w est trouvée,
- x devient le nœud le plus à gauche de R,
- x est déverouillé,
- y devient le nœud le plus à droite de L,
- y est déverrouillé,

ZIG-ZIG : x est verrouillé,
- y est verrouillé,
- si z n'existe pas alors la place de w est trouvée,
- y devient le fils gauche de Top,
- le sous arbre B de y devient la sous arbre gauche de x,
- Top est déverrouillé (y l'est toujours),
- x devient le nœud le plus à gauche de R,

x est déverrouillé,
le nouveau Top est verrouillé,
enfin y est déverrouillé.

Nous pouvons déverrouiller les nœuds insérés dans R ou L, car le nœud Top étant verrouillé, le schéma nous assure que les nœuds des arbres R et L ne peuvent pas être verrouillés par un autre processus que P_i.

D'autre part, le verrouillage du nœud y (cas ZIG) et du nœud z (cas ZIG-ZAG et ZIG-ZIG) n'est pas obligatoire, car lors de ces rotations, ces nœuds ne sont ni modifiés, ni lus.

La rotation ZIG-ZIG comporte une exception: Si l'arbre L ou R est vide, le nouveau nœud Top sera le nœud y. Dans ce cas, les deux dernières opérations du cas ZIG-ZIG ne doivent pas être exécutées.

Ce phénomène est présent dans le cas d'un DeleteMin. En effet, cette opération recherchera le nœud le plus à gauche, les rotations susceptibles d'être trouvées seront des rotations ZIG-ZIG ou ZIG dans le sens Gauche. Donc le nouveau nœud Top verrouillé lors d'une rotation ZIG-ZIG sera toujours le nœud y.

Pour obtenir l'algorithme de Simple-Semi-Splay-Tree concurrent, il suffit de retirer le cas ZIG-ZAG de l'algorithme précédent. Les Single-Simple-Semi-Splay-Tree et Single-Semi-Splay-Tree vont pouvoir être réalisés de la même façon. Il suffit pour chaque cas d'arrêter l'algorithme, lorsque les nœuds x ou y correspondent au nœud cherché.

6.3 Expérimentations

Tous les algorithmes présentés ici ont été implémentés en langage C sur une SEQUENT comportant 9 processeurs. Le système de cette machine appelé DYNIX est un UNIX parallèle compatible BSD4.3. La gestion des processus, les primitives de synchronisation ainsi que la gestion de la mémoire partagée, ont été réalisées à l'aide des librairies standards de DYNIX.

Le jeu d'essai est le suivant:

Borne Inférieure 100.
Borne Supérieure 1000.
Incrémentation Maximale 150.
Nombres de processeurs Varie de 1 à 9.

Les algorithmes utilisés pour les tests avec un processeur sont les mêmes que ceux utilisés en séquentiel. Les primitives ou les marquages ont été retirés du programme pour cette exécution.

6.3.1 Splay-Tree

Nous avons implémenté les versions concurrentes des quatre versions de Splay-Tree: Semi-Splay, Simple-Semi-Splay, Single-Semi-Splay, Single-Simple-Semi-Splay, sa version verrou marquage. Les huit versions sont présentées sur la figure 6.1.

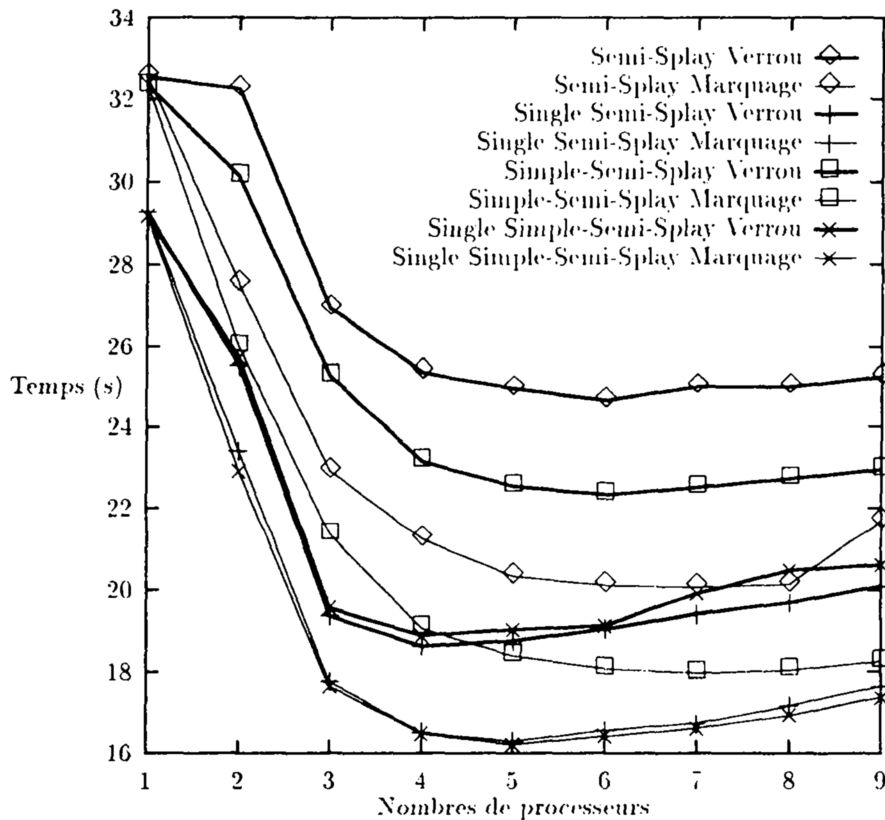


Figure 6.1 : Test sur les différents Splay-Tree

Nous pouvons déjà noter la prédominance des versions avec marquage sur celles avec verrou, comme nous l'avons dit précédemment l'utilisation de marquages réduit de beaucoup le surcoût dû aux primitives de synchronisation.

La version Simple-Semi-Splay est plus performante que la version Semi-Splay, alors qu'en séquentiel elles sont équivalentes.
En effet, lors d'un Insert, l'algorithme de Simple-Semi-Splay remplace le ZIG-ZAG par deux ZIG. Or un ZIG-ZAG concurrent a besoin de deux verrous, alors qu'un ZIG n'en a besoin que d'un.
Si un processus réalise un ZIG, puis un autre, il a plus de chance de trouver le deuxième verrou libre que si il réalise un ZIG-ZAG, où il a besoin du deuxième verrou tout de suite. Cette remarque s'applique aussi au cas de l'algorithme avec marquage.

Nous avons vu que la complexité amortie des versions Single du Semi-Splay et du Simple-Semi-Splay était $O(log\ S)$. Il est donc normal que ces versions soient plus rapides.
La taille de la structure étant plus petite: les processus seront moins nombreux à pouvoir accéder à la structure au même moment, et il est donc aussi normal qu'elles saturent beaucoup plus tôt.
Au contraire des versions non-Single, les résultats des deux versions Single sont équivalentes. En effet, d'une part les arbres des versions Single sont plus petits, d'autre part ces versions n'ont pas besoin de parcourir une branche complète pour trouver l'emplacement du nœud. Ceci souligne le fait que le remplacement de la rotation ZIG-ZAG par deux rotations ZIG, ne porte pas à conséquence sur ce test.

6.3.2 Comparaison avec les files de priorité concurrentes existantes

La figure 6.2 montre les temps de différents types de files de priorités. Nous pouvons remarquer tout d'abord que le Funnel-Table qui était le plus efficace en séquentiel le reste en concurrent malgré sa mauvaise complexité dans le pire des cas. Cette structure est très adaptée aux algorithmes de Branch and Bound, mais elle a besoin de caractéristiques qui la rende peu utilisable dans d'autres contextes.

D'autre part, les Semi-Splay-Tree présentés ici, sont intéressants jusqu'à 5 processeurs car, comme nous l'avons vu précédemment ils saturent trés tôt par rapport aux autres files de priorité.

La version du Skew-Heap donnée ici est la version de Jones [Jon89], mais elle a été modifiée: d'une part, elle est itérative, et d'autre part, elle utilise le mécanisme de marquage.

La structure de Funnel-Tree n'est pas très performante avec peu de processeurs. En effet, sa complexité théorique est de $O(logS)$, quelques soit

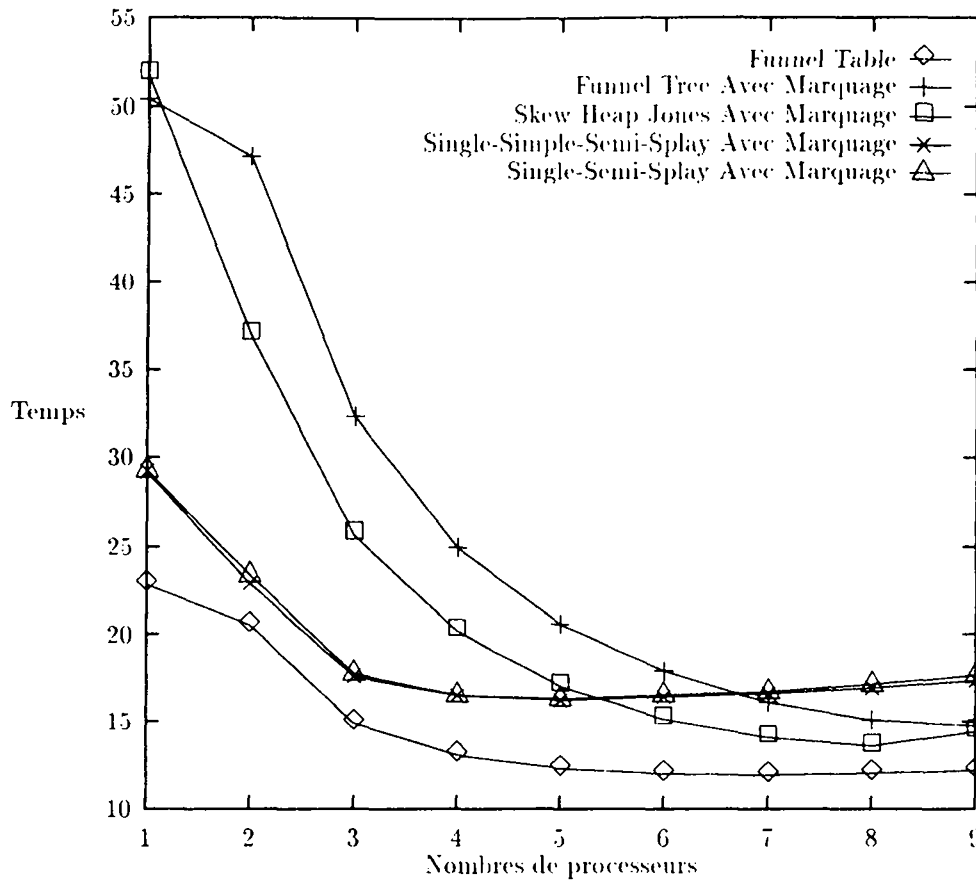


Figure 6.2 : Test de toutes les FP implémentées

le nombre d'éléments présents dans la structure. Mais, elle est plus efficace que les Splay-Tree à partir de 7 processeurs et devient équivalent au Skew-Heap avec 9 processeurs.

Chapitre 7

Conclusion

Nous avons tout d'abord caractérisé la file de priorité utilisée dans les Branch and Bound et dégagé les opérations de base nécessaires à sa manipulation au cours de l'algorithme. Nous nous sommes ensuite intéressés à l'adéquation d'un grand nombre de files de priorités classiquement proposées en Algorithmique et avons dégagés, dans un contexte séquentiel, les plus appropriées à un algorithme Branch and Bound. Nous avons en particulier pris en compte la façon dont sont gérés par ces structures de données les éléments de même priorité, problème important dans le développement de l'arbre d'un Branch and Bound.

Nous avons ensuite étudié l'utilisation de ces files de priorité par des processus concurrents. Nous avons ainsi proposé de distinguer parmi celles de la littérature, deux types de structure de données, actives ou passives, suivant la façon dont sont réalisées les opérations concurrentes par les processus.

Dans un contexte d'implémentation sur machine multiprocesseur à mémoire partagée, nous avons utilisé la technique du verrouillage partiel pour rendre concurrent l'accès à ces structures de données arborescentes.

Nous en avons déduit une méthodologie générale permettant de rendre concurrentes de nombreuses structures de données construites sous forme d'arborescence. Nous avons amélioré nos précédents travaux, [MR90], en remplaçant dans certains cas la plupart des primitives de synchronisation par de simples mécanismes de marquage, optimisant ainsi l'implémentation, quelque soit la machine multiprocesseur désignée.

Une comparaison empirique en séquentiel et en parallèle de plusieurs files de priorité nous a permis de dégager deux d'entre elles.

Dans le cas où l'intervalle de recherche du Branch and Bound est petit, le Funnel-Table, spécialement conçu pour ce cas particulier, est la plus efficace des files de priorité que nous avons expérimentées : facilité de programmation, gestion des éléments de même propriété et opération DeleteGreater efficace.

Dans un cadre général, le Splay-Tree, bien que non spécifique aux Branch and Bound, s'avère très performant.

En fait, la file de priorité, appelé Single Splay Tree, que nous avons proposée, reprenant à son compte l'avantage tiré du faible intervalle du Funnel Table et la généralité d'application du Splay-Tree, regroupe l'ensemble des qualités et performances tant théoriques que pratiques des deux structures que nous attendions.

L'ensemble des travaux présentés contribue à l'étude de certaines files de priorité à accès concurrents et met l'accent sur l'intérêt du développement de nouvelles structures de données dans la cadre du parallélisme.

Bibliographie

[BB87] Bitwas (J.) et Browne (J.C.). – Simultaneous update of priority structures. *IEEE international conference on parallel computing*, 1987, pp. 124–131.

[Bro78] Brown (M.R.). – Implementation and analysis of binomial queue algorithms. *SIAM Comput.*, vol. 7, 1978, pp. 298–319.

[Che90] Cheng (K.H.). – A simultaneous access queue. *Journal of parallel and distributed computing*, vol. 9, 1990, pp. 83–86.

[Cra72] Crane (C.A.). – *Linear Lists and priority queues as balanced binary trees*. – Rapport technique n° 259, STAN, 1972.

[Deo89] Deo (N.). – Data structures for parallel computation on shared-memory machine. *In : SuperComputing*, pp. 341–345.

[Ell80] Ellis (C.S.). – Concurrent search and insertion in 2-3 trees. *Acta Informatica*, vol. 14, 1980, pp. 63–86.

[Ell81] Ellis (C.S.). – Concurrent search and insertion in avl trees. *IEEE Trans. on Computers*, vol. 29, n° 9, septembre 1981, pp. 811–817.

[FC89] Fan (Z.) et Cheng (K.H.). – A simultaneous access priority queue. *International conference on parallel processing*, 1989, pp. I–95–98.

[FC90] Fan (Z.) et Cheng (K.H.). – Design and analysis of simultaneous access priority queue. *Journal of parallel and distributed computing*, vol. 9, 1990, pp. 387–397.

[FT84] Fredman (M.L.) et Tarjan (R.E.). – Fibonnaci heap and their uses and improved network optimizatoin algorithms. *Journal of the ACM*, 1984, pp. 338–346.

[Her90] Herlihy (M.). – A methodology for implementing highly concurrent data structures. *SIGPLAN Not*, vol. 25, n° 3, mars 1990, pp. 197–206.

[Jon86] Jones (D.W.). – An empirical comparaison of priority queue and event set implementation. *Comm. ACM*, vol. 29, n° 4, avril 1986, pp. 191–194.

[Jon89] Jones (D.W.). – Concurrent operations on priority queues. *Comm. ACM*, vol. 32, n° 1, janvier 1989, pp. 132–137.

[KL80] Kung (H.) et Lehman (P.). – Concurrent manipulation of binary search trees. *ACM trans. on Database Systems*, vol. 5, n° 3, 1980, pp. 354–382.

[KW83] Kwong (Y.) et Wood (D.). – A new method for concurrency in b-tree. *IEEE Trans. Software Engr.* vol. 8, n° 3, mai 1983, pp. 211–221.

[LS84] Lai (T.-H.) et Sahni (S.). – Anomalies in parallel branch-and-bound algorithms. *Communication A.C.M.*, vol. 27, june 1984, pp. 594–602.

[LW86] Li (G.) et Wah (B.W.). – Coping with anomalies in parallel branch-and-bound. *IEEE Trans. on Computers*, vol. C-35, n° 6, June 1986, pp. 568–573.

[LY81] Lehman (P.) et Yao (S.). – Efficient locking for concurrent operation on b-tree. *ACM trans. on Database Systems*, vol. 6, n° 4, décembre 1981, pp. 650–670.

[MR90] Mans (B.) et Roucairol (C.). – *Concurrency in Priority Queues for Branch and Bound algorithms*. – RR n° 1311, INRIA, 1990.

[MR91] Mans (Bernard) et Roucairol (Catherine). – *Theoretical comparisons of parallel best-first search branch and bound algorithms*. – RR n° à paraître, INRIA-Rocquencourt, 1991.

[RK88] Rao (V.N.) et Kumar (V.). – Concurrent insertions and deletions in a priority queue. *IEEE proceedings of International Conference on Parallel Processing*, 1988, pp. 207–211.

[Rou87] Roucairol (C.). – *Du séquentiel au parallèle: la recherche arborescente et son application à la programmation quadratique en variable 0-1*. – Thèse d'état, Université Paris VI, juin 1987.

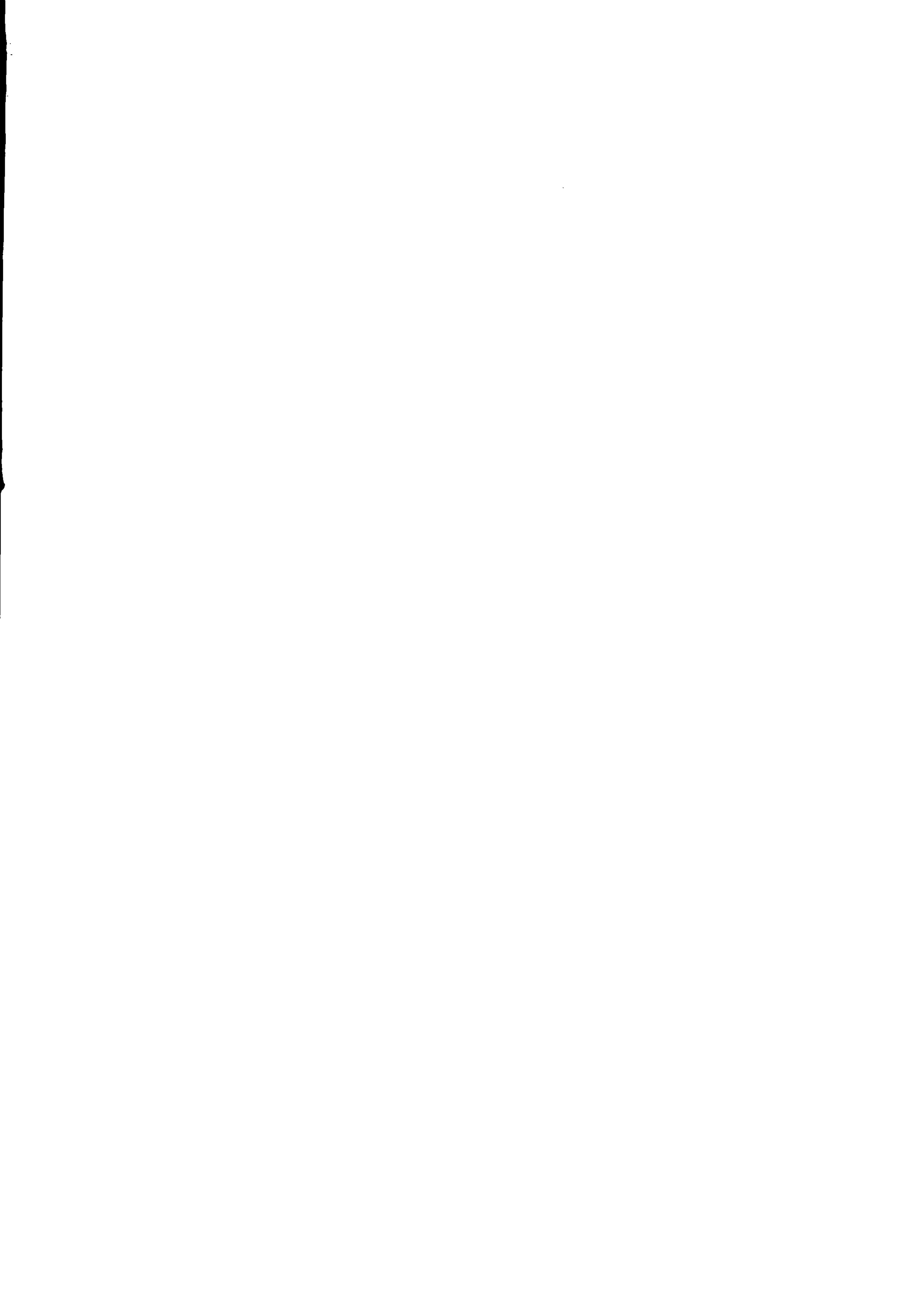
[ST83] Sleator (D.D.) et Tarjan (R.E). – Self-adjusting trees. *In: 15th ACM Symposium on theory of computing*, pp. 235–246.

[ST86] Sleator (D.D.) et Tarjan (R.E). – Self-adjusting heaps. *SIAM J. Comput.*, vol. 15, n° 1, février 1986, pp. 52–69.

[Tar85] Tarjan (R.E). — Amortized computational complexity. *SIAM J. Comput.*, vol. 6, n° 2, avril 1985, pp. 306–318.

[Vui78] Vuillemin (J.). — A data structure for manipulating priority queues. *Comm. ACM*, vol. 21, 1978, pp. 309–314.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique.



ISSN 0249-6399