Chapitre IV Stratégies de parallélisation des opérations d'une requête

1 Introduction

Dans le deuxième chapitre nous avons vu qu'à partir des approches de placement et des méthodes de répartition de données deux formes de parallélisme peuvent être extraites : le parallélisme intra-opération et le parallélisme inter-opération. Dans le parallélisme inter-opération, on peut extraire deux types de parallélisme : le parallélisme indépendant et le parallélisme dépendant (appelé aussi parallélisme pipeline ou de flux). Le parallélisme intra-opération des opérations de jointure [VAL 84, SCH 89] et de point fixe pour l'évaluation parallèle de requêtes récursives a été déjà étudié respectivement au deuxième chapitre et au troisième chapitre. En ce qui concerne la phase de parallélisation inter-opération, problème central de l'optimisation des requêtes SQL pour une exécution parallèle, plusieurs auteurs [SCH 90, HON 91, GAN 92, HON 92, CHE 92b, TAN 93, ZIA 92, LAN 93, ZIA 93, HAM 93b, HAS 94, CHE 95] ont proposé des stratégies de parallélisation.

Deux approches de parallélisation inter-opération ont été décrites dans la littérature [LU 94] : l'approche *à deux phases* et l'approche *mono-phase*. Dans l'approche à deux phases [HON 92, CHE 92a, HAM 94a, HAM 94b, HAS 94, CHE 95] la première phase, appelée génération de plan, consiste à engendrer un plan d'exécution (sans tenir compte des ressources). A chaque opération est associée la stratégie la plus adéquate. La seconde phase assure une allocation optimale des ressources (mémoire et processeurs) pour le plan généré dans la première. Quant à l'approche mono-phase [SCH 90, CHE 92b,

LAN 93, ZIA 93], la génération de plan et l'allocation des ressources sont intégrées en une seule phase.

Dans le système parallèle Gamma à mémoire distribuée [SCH 90], les stratégies de parallélisation et les programmes parallèles produits, en tenant compte de la contention de ressources (i.e. mémoire), dépendent fortement des structures arborescentes qui constituent l'espace de recherche. Les éléments constituant l'espace de recherche (i.e. les arbres linéaires gauches, les arbres linéaires droits et les arbres Bushy, cf. 2.2) sont engendrés par l'utilisation de la classe des algorithmes énumératifs et/ou de la classe des algorithmes aléatoires [IOA 90, IOA 91, LAN 91, LAN 92, LAN 93]. Dans le cas du système parallèle XPRS à mémoire commune, la génération d'un plan d'exécution parallèle est fondée sur l'hypothèse suivante [HON 91] : " *le meilleur plan parallèle est une parallélisation du meilleur plan séquentiel* ". Cette hypothèse a été validée par des résultats expérimentaux dans un environnement à mémoire commune, s'appuyant principalement sur des requêtes extraites du Benchmark proposé par [BIT 83].

Ces observations montrent clairement l'importance des stratégies d'optimisation physique sur les performances du plan d'exécution parallèle. C'est pour cette raison que nous commençons d'abord, par présenter des stratégies d'optimisation physique dans un environnement centralisé. Ces stratégies de recherche sont fondées soit sur des approches énumératives soit sur des approches aléatoires. Ensuite, nous décrivons d'une part, des stratégies de parallélisation inter-opération mono-phase, et d'autre part, deux stratégies de parallélisation à deux phases. L'exécution parallèle des opérations d'une requête SQL risque souvent d'engendrer un nombre de communications inter-processeur très important. Dans le but de réduire ces communications, nous présentons une méthode d'optimisation de transfert de données et de contrôle. Celle-ci consiste à éviter de transmettre des messages de données et de contrôle qui ne sont pas nécessaires à une exécution parallèle. Le principe est basé sur le mécanisme de propagation des attributs de répartition et du nombre de processeurs alloués à une opération relationnelle. Enfin, nous présentons une analyse des performances permettant de mesurer l'apport et l'efficacité, d'une part, des formes de parallélisme (intra-opération et inter-opération) en fonction du nombre de processeurs, et, d'autre part, du processus de propagation.

2 Stratégies d'optimisation physique

Les stratégies de recherche ont une forte influence sur la qualité du plan d'exécution parallèle. Avant d'étudier les deux approches de parallélisation

inter-opération, nous décrivons les différentes stratégies de recherche utilisées dans un environnement centralisé.

2. 1 Classification des requêtes

Une requête est définie comme étant une suite d'opérateurs algébriques relationnels appliqués aux relations de la base de données. On s'intéresse plus particulièrement à l'opérateur de jointure et à *l'ordre* d'évaluation de ces opérateurs. La requête peut être caractérisée par le nombre de relations qu'elle référence, le nombre de prédicats de jointure, la façon dont ils sont agencés (i.e. forme de la requête) et le type d'utilsation (i.e. ad-hoc ou répétive).

Taille d'une requête

La taille d'une requête dépend du nombre de relations qu'elle référence. Soit N le nombre de relations jointes et N-1 le nombre de jointures, une *requête est dite simple* si $1 \leq N \leq 10$ [SWA 88, SWA 89]. C'est le cas des applications bases de données traditionnelles. D'autres auteurs comme [LAN 93] restreignent N à 6. En revanche, une *requête complexe* possède un nombre de relations $N>10$ pour [SWA 88, SWA 89] et $N \geq 7$ pour [LAN 93]. En effet, cela est dû aux besoins des nouvelles applications telles que la CAO, les systèmes à Base de Connaissances et certaines applications gérées par les SGBD orientés objets qui font appel aux systèmes relationnels pour stocker des données.

Forme d'une requête

La forme d'une requête (chaîne, étoile, clique) (figure IV.1) indique la façon dont les relations sont jointes à l'aide de prédicats, ainsi que le nombre de relations référencées. Une *requête chaîne ou linéaire* est une requête pour laquelle chaque relation peut joindre, au plus, deux relations, exceptées les relations aux extrémités qui ne peuvent en joindre qu'une seule. Une *requête étoile* [ONO 90] est une requête dont une relation et une seule est jointe à toutes les autres relations, représentant ainsi une étoile. Dans le cas de la requête chaîne et de la requête étoile, le nombre de relations et de prédicats de jointure reste le même, seule la forme de la requête change. Le nombre de branches varie de *1* pour une requête linéaire, à *N-1* pour une requête étoile (*N-1* étant le nombre de prédicats de jointure d'une requête) en passant par des requêtes intermédiaires à *N-p* branches ($1 < p < N-1$). Ce type de requête est intéressant car il offre la complexité la plus forte pour un même nombre de relations et de prédicats de jointure. On peut aussi citer le cas d'une *requête clique* où les relations sont toutes joignables deux à deux. Cette forme de

requête est, en principe, peu rencontrée, dans la mesure où elle est due à une mauvaise conception de la base de données. Cependant, le résultat produit est intéressant puisqu'il prend en compte le cas où aucun prédicat de jointure n'est spécifié dans la clause " Where " et où on effectue un produit cartésien entre toutes les relations référencées par la requête.

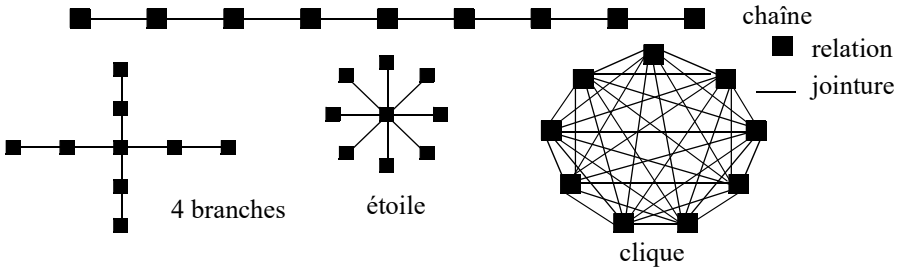

Figure IV.1 : Formes d'une requête [ONO 90]

2. 2 Espace de recherche

2. 2. 1 Caractérisation de l'espace de recherche

D'après [LAN 92], " L' espace de recherche est l'ensemble virtuel de tous les plans d'exécution[1] possibles correspondant à une requête donnée ". Cet espace peut être restreint selon la nature des plans d'exécution et la stratégie de recherche appliquée.

La *nature des plans d'exécution* est déterminée en fonction de deux critères : la forme des structures arborescentes (i.e. arbre linéaire gauche, arbre linéaire droit et arbre bushy) et la prise en compte de plans avec des produits cartésiens[2].

Un arbre est appelé arbre linéaire si tous les noeuds internes ont au moins pour fils un noeud terminal [IOA 91]. Dans le cas contraire, il est appelé arbre bushy. Un arbre linéaire gauche est un arbre linéaire où tous les fils droits (appelés relations intérieures *" inner ")* doivent être des noeuds terminaux (i.e. relations de base) [LU 94]. Un arbre linéaire droit est un arbre linéaire où tous les fils gauches (appelés relations extérieures *" outer ")* doivent être des noeuds terminaux. La figure IV.2 illustre des structures arborescentes d'opéra-

1. Un plan d'exécution est représenté par un arbre binaire appelé aussi arbre de traitement.
2. Un espace de recherche où les produits cartésiens ne sont pas acceptés est tel que le choix des opérandes est conditionné par l'existence d'un prédicat de jointure.

teurs relationnels associées à la requête multi-jointure R1 ⋈ R2 ⋈ R3 ⋈ R4.

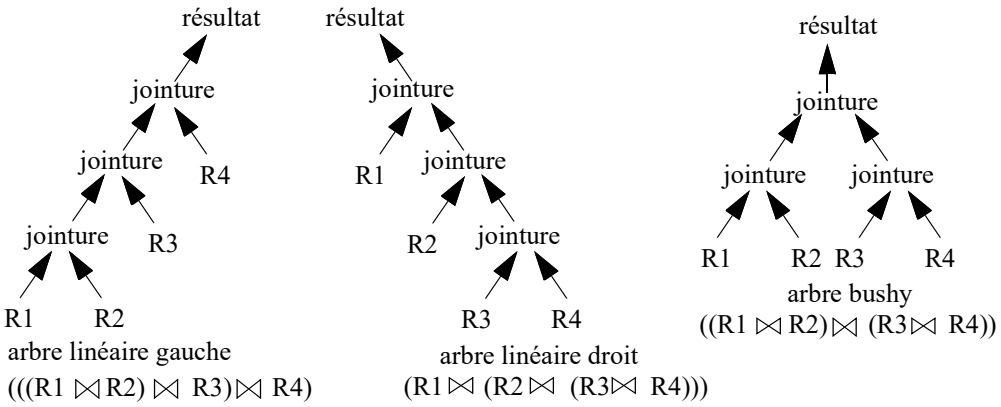

Figure IV.2 : Formes des structures arborescentes

Les requêtes possédant de nombreux prédicats de jointure rendent difficile la gestion d'un espace de recherche devenu trop important ; c'est pourquoi certains auteurs [SWA 88, SWA 89] ont choisi d'éliminer les arbres bushy. Cet espace réduit est appelé *espace valide*. Ce choix est du au fait que cet espace valide représente une portion significative de l'espace de recherche, dans lequel se trouve la solution optimale. Cependant, cette assertion n'a jamais été validée. D'autres, comme [ONO 90], pensent que ces méthodes diminuent les chances d'obtenir des solutions optimales. Plusieurs exemples [ONO 90] montrent l'importance de l'impact de ce choix restrictif.

2. 2. 2 Taille de l'espace de recherche

L'importance de la forme d'une requête et de la nature des plans d'exécution est due à leur incidence sur la *taille de l'espace de recherche*. Si l'on a *N* relations dans une expression de jointure, la question est de savoir combien de plans d'exécution (i.e. arbres de traitement) peuvant être construits, tout en tenant compte de la nature de l'espace de recherche. La taille de cet espace varie aussi en fonction de la forme de la requête. A ce titre, [TAN 91, LAN 92] ont proposé un tableau (table 1) illustrant les bornes inférieures et supérieures de l'espace de recherche en tenant compte à la fois de la nature de celui-ci et des caractéristiques des requêtes (i.e. nombre de relations et forme de la requête linéaire et clique). Ce tableau a été étendu [BLO 94] en calculant le nombre de plans d'exécution pour des requêtes de forme étoile en raison de

l'intérêt de ce résultat. Dans la table 1, l'espace de recherche a été découpé en deux sous-ensembles :

- un premier sous-ensemble contenant les arbres linéaires gauches, appelé espace linéaire gauche,
- un second sous-ensemble contenant à la fois les arbres linéaires et les arbres bushy, appelé, par abus de langage, espace bushy.

Table 1: Bornes inférieures et supérieures de l'espace de recherche [TAN 91, LAN 92]
(unité : nombre de plans d'exécution)

Nombre de relations	**Espace linéaire gauche (ou droit)**			**Espace bushy**		
N	chaîne 2^{N-1}	étoile $2(N-1)!$	clique $N!$	chaîne $\frac{2^{N-1}}{N}\binom{2N-2}{N-1}$	étoile $2^{N-1}(N-1)!$	clique $\frac{(2N-2)!}{(N-1)!}$
3	4	4	6	8	8	12
4	8	12	24	40	48	120
5	16	48	120	224	384	1.680
6	32	240	720	1.344	3.840	30.240
7	64	1.440	5.040	8.448	46.080	665.280
8	128	10.080	40.320	54.912	645.120	17.297.280
9	256	80.640	362.880	366.080	10.321.920	518.918.400
10	512	725.760	3.628.800	2.489.344	185.794.560	17.643.225.600

Ces résultats montrent la croissance exponentielle du nombre de plans d'exécution en fonction du nombre de relations. Ceci met en évidence la difficulté d'une gestion d'un ensemble de solutions parfois très important et donc la nécessité *d'adapter la stratégie de recherche au problème* (i.e. type d'utilisation, taille et forme d'une requête).

2. 3 Stratégies de recherche

Dans la littérature, on distingue deux grandes classes de stratégies permettant de résoudre le problème de l'ordonnancement des jointures dans le cadre de l'optimisation de requêtes :

- les *stratégies énumératives ou déterministes,*
- les *stratégies aléatoires*.

La description des principes des stratégies de recherche s'appuie sur les algorithmes de recherche génériques décrits dans [LAN 91] et sur l'étude comparative entre les algorithmes aléatoires effectuée par [SWA 88, SWA 89, IOA 90, IOA 91, LAN 93].

2. 3. 1 Stratégies énumératives

Ces stratégies sont basées sur l'approche générative. Elles construisent des plans d'exécution à partir de sous-plans déjà optimisés en partant de toutes ou partie des relations de base de la requête. Le choix du noeud courant se fait selon des critères relatifs aux différents principes de recherche des stratégies énumératives. Le noeud courant sélectionné parmi les noeuds de l'ensemble *plan* est retranché de celui-ci. Si l'on est en présence d'un plan d'exécution complet (i.e. à partir de son noeud racine, on capture toutes les relations de la requête) ; c'est qu'on a trouvé une solution, sinon l'ensemble des successeurs du noeud courant est engendré. Des heuristiques sont utilisées pour éliminer certains successeurs selon la stratégie de recherche, afin de réduire l'espace de recherche. Les successeurs conservés sont ajoutés à la suite des noeuds contenus dans l'ensemble *plan* où ils seront eux-mêmes exploités ultérieurement. Ce processus est réitéré jusqu'à ce que l'ensemble *plan* devienne vide. Dans l'ensemble des solutions générées, seul le plan d'exécution optimal est retourné pour l'exécution. Toutes les stratégies énumératives suivent cet algorithme, à quelques variantes près. A ce titre, plusieurs stratégies sont décrites dans [LAN 91] :

Stratégie de recherche en largeur d'abord

Elle consiste à construire des " sous-plans " pour toutes les relations de base et à les enrichir à chaque itération de la stratégie jusqu'à l'obtention de toutes les solutions (figure IV.3). Le noeud le moins récent est sélectionné et le successeur le moins coûteux est conservé. Dans la littérature, on fait souvent référence à la stratégie énumérative du Système R [SEL 79] pour désigner la stratégie de recherche en largeur d'abord. Celle-ci suit le principe de la programmation dynamique et offre une complexité proportionnelle à $O(2^N)$, N étant le nombre de relations de la requête, ce qui explique les problèmes rencontrés lors de l'optimisation de requêtes complexes.

```
Algorithme de recherche en largeur d'abord
Début
plan := {toutes les relations de base};
tant que plan ≠ ∅ faire
    noeud_courant := noeud_moins_récent (Plan); plan := plan - {noeud_courant};
    si plan_complet (noeud_courant) alors solution := solution ∪ {noeud_courant};
    sinon
        succ := générer_successeurs (noeud_courant);
        succ := éliminer_plan_équivalent (noeud_courant); /*si plusieurs successeurs
        sont équivalents, seul le successeur le moins coûteux est conservé */
        plan := plan ∪ {succ};
    fin si
fin tant que
retourner (plan_optimal (solution));
Fin
```

Figure IV.3 : Stratégie de recherche en largeur d'abord RLA [LAN 91]

Stratégies de recherche en profondeur d'abord

Selon l'initialisation de l'ensemble *plan* et l'heuristique exploitée pour la sélection du noeud courant, on obtient deux stratégies de recherche : l'une appelée Heuristique d'Augmentation HA qui ne génère qu'une seule solution, et l'autre appelée Heuristique d'Augmentation* HA* qui génère autant de solutions que de relations contenues dans la requête.

La stratégie de recherche HA exploite une relation de base initialement choisie selon le critère de la relation de plus petite cardinalité. Cette relation est enrichie jusqu'à l'obtention d'un plan solution. Le noeud le plus récent est sélectionné et le successeur ayant la plus petite cardinalité est conservé. Quant à la stratégie HA*, elle est basée sur le même principe que la stratégie précédente, si ce n'est que l'ensemble initial contient toutes les relations de base au lieu d'une, dans le cas précédent.

2. 3. 2 Stratégies aléatoires

Les stratégies énumératives sont inadéquates pour optimiser des requêtes complexes puisque le nombre de plans devient rapidement trop important (table 1). Pour résoudre ce problème, on préfère utiliser des stratégies aléatoires. L'approche *transformationnelle* caractérise ce genre de stratégies.

Plusieurs règles de transformation ont été proposées [SWA 88, IOA 90, IOA 91] dont la validité dépend de la nature de l'espace de recherche considéré [LAN 93]. En effet, les deux règles suivantes décrites dans [SWA 88], ne

s'appliquent que dans le cas d'un espace linéaire. Soient *A* le plan courant et *i*, *j*, *k* des relations ; *A = (... i ... j ... k ...)* est le plan initial.

1- *Swap*

Cette règle consiste à permuter deux relations intérieures *i* et *j* choisies aléatoirement : *nouveauA = (... j ... i ... k ...)*

2- 3Cycle

Il s'agit de faire un cycle avec trois relations *i*, *j* et *k* : *i* prend la place de *j* qui prend la place de *k*, lui-même prenant la place de *i*. *nouveauA = (... k ... i ... j ...)*

Dans [IOA 90, IOA 91], de nouvelles règles sont mentionnées. Soient A, B et C des relations de base ou des expressions de traitement de jointure, l'ensemble des règles de transformation suivant est valable, à la fois, pour des espaces linéaires et des espaces bushy.

1- *commutativité de jointures* $A \bowtie B \rightarrow B \bowtie A$

2- *associativité de jointures* $(A \bowtie B) \bowtie C \rightarrow A \bowtie (B \bowtie C)$

3- *permutation jointure gauche* $(A \bowtie B) \bowtie C \rightarrow (A \bowtie C) \bowtie B$

4- *permutation jointure droite* $A \bowtie (B \bowtie C) \rightarrow B \bowtie (A \bowtie C)$

Certaines de ces transformations, choisies aléatoirement, ne sont donc pas forcément retenues si leur validité n'est pas vérifiée dans l'espace déterminé.

Toutes les stratégies aléatoires partent généralement d'un plan d'exécution initial que l'on cherche itérativement à améliorer par l'application d'un ensemble de règles de transformation. Le(s) plan(s) de départ peut(peuvent) être obtenu(s) par l'intermédiaire d'une stratégie de recherche en profondeur d'abord de type Heuristique d'Augmentation. Deux techniques d'optimisation ont été déjà étudiées et comparées abondamment : l'Amélioration Itérative et le Recuit Simulé [SWA 88, SWA 89, IOA87 IOA 90, IOA 91, LAN 91] dont les principes sont décrits brièvement ci-après.

Amélioration Itérative

La stratégie de l'Amélioration Itérative consiste à appliquer à un plan initial des transformations successives. Ces transformations sont choisies aléatoirement dans l'ensemble des plans adjacents au plan initial. Le choix de plusieurs plans de départ, un pour chaque exécution de la boucle interne de l'algorithme, caractérise la stratégie (figure IV.4). Cette boucle est appelée optimisation locale. A chaque transformation, le nouveau plan obtenu est comparé au précédent et il remplace ce dernier si son coût est inférieur. Après un certain nombre de transformations, on obtient un minimum local pour chaque plan de départ. Un plan est un minimum local si son coût est le plus bas parmi tous les plans voisins. L'algorithme retourne en sortie le minimum local possédant le coût le plus faible parmi tous les minima locaux appelé minimum global.

```
Algorithme Amélioration Itérative
Début
plan_courant := générer_plan_initial (); /* par exemple avec la stratégie HA */
min := plan_courant;
tantque non condition_arret () faire
   répéter
      nouveau_plan := transformer (plan_courant);
      si cout (nouveau_plan) < cout (plan_courant) alors
         plan_courant := nouveau_plan;
      fin si
   jusqu'à " l'obtention d'un minimum local ";
   si cout (plan_courant) < cout (min) alors
      min := plan_courant;
   fin si
   plan_courant := générer_aléatoirement_plan ();
fin tant que
retourner (min);
Fin
```

Figure IV.4 : Stratégie de l'Amélioration Itérative

Recuit Simulé

Cette stratégie (figure IV.5) est dérivée de l'analogie au traitement de la cristallisation d'une solution liquide, ce qui explique la terminologie utilisée (i.e. température, refroidir, ...). Elle diffère de la stratégie de l'Amélioration

Itérative par le fait qu'un seul plan de départ est généré tout au long de l'exécution de l'algorithme. Le nombre de transformations successives appliquées au plan initial est rythmé par la diminution d'une température initiale. La condition d'arrêt est relative, à la fois, à la température et à la stabilité de la solution courante (i.e. on a la même solution pendant plusieurs itérations successives). Contrairement à l'Amélioration Itérative, le Recuit Simulé accepte aussi certains plans avec un coût élevé, dans la mesure d'une certaine probabilité. En effet, accepter de mauvaises transformations, c'est exécuter un " hill climbing " (i.e. " de l'autre côté de la colline, il existe peut-être une meilleure solution ") [IOA 87]. L'algorithme s'arrête lorsqu'il est considéré comme " gelé " (i.e. quand la température est égale à zéro). A ce moment là, un minimum global est obtenu.

```
Algorithme Recuit simulé
Début
génération d'un plan initial;
initialisation de la température T; min := plan_courant;
tantque température finale non atteinte faire
   répéter
      nouveau_plan := transformer (plan_courant);
      ΔC := cout (nouveau_plan) - cout (plan_courant);
      si ΔC <= 0 alors alors plan_courant := nouveau_plan; fin si
      si ΔC > 0 alors
         plan_courant := nouveau_plan avec une certaine probabilité e^(-ΔC/T) ; fin si
      si cout (plan_courant) < cout (min) alors min := plan_courant; fin si
   jusqu'à " la stabilité ";
   réduire (T);
fintantque
retourner (min);
Fin
```

Figure IV.5 : Stratégie du Recuit simulé [IOA 90]

Choix des paramètres

L'évaluation de ces stratégies est très délicate en raison de l'intervention, à la fois, de paramètres et de facteurs aléatoires. La difficulté réside dans le choix de ces paramètres. En effet, de la qualité du choix vont dépendre le coût d'optimisation et la qualité du plan optimal. Suite à la mise au point des paramètres, la comparaison des algorithmes va permettre de déterminer le plus efficace des algorithmes aléatoires face au problème d'optimisation de requêtes complexes. Cependant, les résultats obtenus par [SWA 88] et par [IOA 90] diffèrent radicalement puisque, pour [SWA 88], l'algorithme de l'Amélioration Itérative est meilleur que celui du Recuit Simulé, alors que

pour [IOA 90], on a le contraire (même si pour ces derniers, leur conclusion reste plus tempérée).

Les paramètres ont été déterminés suite à des expériences avec différentes alternatives, dans le cas de [IOA 90], ou en appliquant la méthodologie des expériences factorielles [SWA 88]. A titre référentiel, un exemple de l'utilisation de ces expériences factorielles est donné dans [SWA 87].

Amélioration Itérative

choix du plan de départ

Le choix du plan de départ pour chaque exécution de la boucle interne de l'algorithme se fait aléatoirement, excepté pour la première exécution où le plan de départ est le plan initial.

détection d'un minimum local

La détection d'un minimum local pose un problème dans la mesure où il parait impossible d'énumérer tous les plans voisins et de vérifier ainsi que le plan courant est un minimum local. Pour résoudre ce problème, [SWA 88] et [IOA 90] déterminent une approximation du nombre de plans voisins à tester. Ce nombre est un paramètre à préciser avant chaque exécution de l'algorithme.

critère d'arrêt de l'algorithme

Dans [SWA 88], les auteurs accordent un intérêt particulier au critère d'arrêt de l'algorithme. Il s'agit de fixer une limite temporelle maximum au delà de laquelle il y a arrêt de l'optimisation. Cependant, l'algorithme peut prendre fin plus tôt, si un minimum global est atteint.

minimum global

Le problème consiste désormais à identifier un minimum global. Les expériences montrent qu'il suffit d'estimer une limite inférieure sur le coût du minimum global. Ainsi, dès qu'une solution est suffisamment proche de cette limite, on peut dire qu'un minimum global est atteint. Si cette limite a été mal estimée, l'exécution de l'algorithme risque d'être un peu plus longue que ce qu'elle devrait être, mais il n'y aura aucune conséquence sur la qualité de la solution obtenue.

Recuit Simulé

température initiale et facteur de réduction de la température

Il s'agit de déterminer la température initiale ainsi qu'un facteur de réduction de la température qui vont permettre de "rythmer" l'exécution de l'algorithme.

critère d'arrêt de la boucle interne

Le critère d'arrêt de la boucle interne est relatif à la fois à la température et à la stabilité de la solution courante (i.e. la solution reste la même durant un certain nombre d'itérations).

critère d'arrêt de l'algorithme

Le critère d'arrêt de l'algorithme dépend de la température. Celle-ci doit être suffisamment basse afin que la probabilité d'accepter un plan voisin de coût plus élevé soit négligeable et qu'il n'y ait pas de plans de coût inférieur au plan courant.

2. 3. 3 Discussion

Dans [SWA 88, SWA 89] et [IOA 90, IOA 91], les auteurs ont concentré leurs efforts sur l'évaluation des performances des algorithmes aléatoires de l'Amélioration Itérative et du Recuit Simulé, mais la divergence de leurs résultats souligne la difficulté d'une telle entreprise. En effet, pour Swami et Gupta, l'algorithme du Recuit Simulé n'est jamais supérieur à celui de l'Amélioration Itérative quel que soit le temps consacré à l'optimisation, alors que pour Ioannidis et Cha Kong, il est meilleur que l'algorithme de l'Amélioration Itérative après un certain temps d'optimisation.

Dans [IOA 90, IOA 91], les auteurs tentent d'expliquer cette différence. Tout d'abord, l'espace de recherche considéré est restreint aux arbres linéaires gauches dans le cas de Swami et Gupta, alors que Ioannidis et Cha Kong étudient l'espace de recherche dans sa totalité. Dans [IOA 91], les auteurs ont étendu leurs travaux sur l'étude de la forme de la fonction de coût en accentuant l'analyse des espaces linéaires et bushy, et par là même, ne tiennent compte que des résultats attendus dans cette portion restreinte de l'espace de recherche pour que la comparaison reste cohérente. La deuxième différence concerne la méthode de jointure. Swami et Gupta ont choisi la méthode de jointure par hachage, alors que Ioannidis et Kong mettent en place deux méthodes de jointure : jointure par produit cartésien et jointure par tri-fusion. Ils choisissent même d'intégrer la méthode de jointure par hachage pour mon-

trer que *leurs résultats ne dépendent pas du choix d'une méthode*. Une autre variante dans l'évaluation du coût du plan d'exécution (temps CPU pour les premiers et temps Entrées/Sorties pour les seconds) n'a pas, non plus, une incidence significative sur la divergence des résultats. En revanche, ils pensent intuitivement que le nombre de plans voisins, la détermination du minimum local dans le cas de l'algorithme de l'Amélioration Itérative et la définition des règles de transformation à appliquer sont des éléments importants dans l'explication de cette différence. Par exemple, si le nombre de plans voisins n'est pas assez grand, on peut ainsi écarter des minima locaux potentiels et même en désigner comme tels, alors qu'ils ne le sont pas vraiment. Dans ce cas, les résultats sont faussés. Les règles de transformation appliquées par Swami et Gupta génèrent des plans d'exécution voisins avec des différences de coût importantes, ce qui rend la forme de la fonction de coût plus " plate " (i.e. elle n'a pas la forme d'un gobelet) [IOA 90]. Ainsi, l'algorithme du Recuit Simulé n'a plus la possibilité de passer un long moment dans cette zone de plans à bas coût et offre alors peu d'amélioration. En revanche, l'algorithme de l'Amélioration Itérative peut atteindre facilement un minimum local et étudier beaucoup d'entre eux. Le critère d'arrêt du Recuit Simulé défini dans [SWA 88] ne laisse pas le temps à la probabilité de diminuer suffisamment. En effet, lorsque le temps limite est atteint, la probabilité d'accepter des plans d'exécution de coût élevé est encore forte et le plan optimal produit a un coût encore trop élevé.

Suite à ces remarques et aux résultats apportés par chacun, il est difficile de conclure sur la supériorité d'une stratégie par rapport à l'autre. Cependant, chacun d'eux propose une solution pour améliorer les performances de ces algorithmes. Ioannidis et Kong ont choisi de proposer une nouvelle méthode, appelé Optimisation en Deux Phases ODP [IOA 90], qui consiste à appliquer, d'abord, l'algorithme de l'Amélioration Itérative, et ensuite, l'algorithme de Recuit Simulé. Quant à Swami, il a choisi d'expérimenter un ensemble d'heuristiques dans le but d'améliorer les performances des algorithmes de l'Amélioration Itérative et du Recuit Simulé [SWA 89]. Les travaux de Ioannidis et Kong ont pu montrer que le choix d'une méthode de jointure n'a pas d'influence directe sur les performances des stratégies de recherche. Dans un environnement parallèle, Lanzelotte et al. [LAN 93] ont montré que la stratégie de recherche en largeur d'abord est inapplicable dans un espace de recherche bushy pour des requêtes avec 9 relations ou plus. L'utilisation d'un algorithme aléatoire s'avère alors indispensable. Les auteurs ont donc développé à ce titre un algorithme aléatoire appelé Recuit Simulé Tourné RST " Toured Simulated Annealing " dans un contexte d'exécution parallèle [LAN 93] (cf. 3. 3 page 215).

3 Stratégies de parallélisation inter-opération mono-phase

Récemment, plusieurs auteurs [SCH 90, CHE 92b, ZIA 93] ont proposé des stratégies de parallélisation extraites des *espaces de recherche* représentés par des structures arborescentes appelées arbres linéaires gauches, arbres linéaires droits et arbres bushy. A partir de chaque arbre, on peut engendrer un programme parallèle en intégrant le parallélisme intra-opération et inter-opération.

Schneider [SCH 90] construit, à partir d'un plan d'exécution (ou arbre de traitement) , un *graphe de dépendance* basé sur l'utilisation d'un algorithme de jointure par hachage. Il considère deux méthodes de jointure par hachage : la jointure par hachage simple et la jointure par hachage hybride [SCH 89]. Dans la description de ce graphe, les auteurs considèrent la jointure des relations R et S, où R est la plus petite relation. La plus petite relation est toujours utilisée pour construire la table de hachage, afin de minimiser le nombre de fois où la relation extérieure doit être lue sur le disque.

Les algorithmes de jointure par hachage [DEW 84, SCH 89, MEH 93], s'exécutent en deux étapes. Dans la première phase, une table de hachage est construite avec la plus petite des deux relations (R). Dans la seconde phase, les tuples de la seconde relation (S) sont utilisés pour sonder la table de hachage afin de trouver les tuples pertinents pour calculer la jointure. L'opération de jointure par hachage peut donc être vue comme deux opérations séparées : construire la table de hachage " build " et sonder la table de hachage " probe ". La première phase doit *obligatoirement* être exécutée avant la seconde (i.e. l'opération probe peut s'exécuter uniquement quand l'opération build est totalement terminée). Les relations de base Ri opérandes des jointures sont représentées par l'opérateur $scan_i$ qui effectue la lecture de la relation Ri, la répartition de la relation en sous-relations avec une fonction de type hachage ou intervalle, et l'envoi de ces sous-relations sur les processeurs où s'exécutent les jointures.

Afin de faciliter la description des principales stratégies de parallélisation inter-opération, la représentation suivante sera utilisée : dans un graphe de dépendance, les sous-graphes composés d'opérateurs entourés par un trait noir représenteront les opérateurs pouvant s'exécuter en pipeline. Dans un sous-graphe, les flèches entre deux opérateurs {scan, build} ou {scan, probe} ou {probe, build} montrent la relation de producteur/consommateur. Une flèche arrondie reliant un sous-graphe1 à un sous-graphe2 signifie que le sous-graphe2 doit attendre que le sous-graphe1 soit terminé avant de commencer son exécution.

Dans les plans d'exécution représentés dans la suite, nous utiliserons les constructeurs Seq, Pipe et Par [MAU 93]. Le constructeur *Seq op ; op2 End_Seq* signifie que l'opération op2 doit attendre la fin de l'opération op1 avant de commencer son exécution. Le constructeur *Pipe op1 - op2 End_Pipe* dénote que l'opération op2 commence son exécution dès qu'un des tuples qu'elle consomme a été produit par op1. Les tuples produits par op1 sont immédiatement consommés par op2 ; il y a donc une exécution parallèle entre op1 et op2. Le constructeur *Par op1 op2 End_Par* exprime le fait que les opérations op1 et op2 s'exécutent simultanément et indépendamment.

3. 1 Arbre linéaire gauche

La figure IV.6 montre une requête composée de N jointures, représentée par un arbre linéaire gauche et son graphe de dépendance associé avec B_i, P_i et S_i désignant respectivement $build_i$, $probe_i$ et $scan_i$.

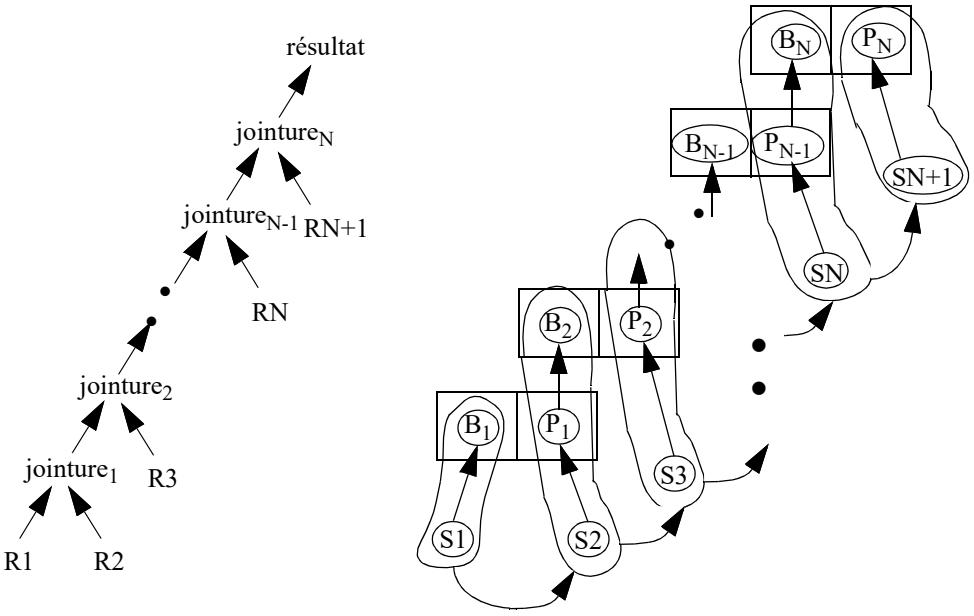

Figure IV.6 : Arbre linéaire gauche et graphe de dépendance associé [SCH 90]

A partir du graphe de dépendance, nous pouvons constater que les opérations $scan_i$ ne peuvent pas s'exécuter en parallèle. De plus, la chaîne de pipeline la plus longue est de la forme Pipe $scan_k$ - probe $J_{(k-1)}$ - $build_k$ End_Pipe pour $k \in [2, N]$, étant donné qu'une opération $probe_i$ ne peut pas commencer

son exécution avant que l'opération $build_i$ correspondante soit totalement finie. Ainsi, à partir d'un arbre linéaire gauche, on obtient le plan d'exécution parallèle suivant :

```
Seq
    Pipe      scan1      -      build1      End_Pipe;
    Pipe   scan2   -   probe  J1   -   build2   End_Pipe;
    Pipe   scan3   -   probe  J2   -   build3   End_Pipe;
    •              •                  •
    Pipe   scanN   -   probe  JN-1   buildN   End_Pipe;
    Pipe    scanN+1    -    probe     JN      End_Pipe;
End_Seq
```

Le plan d'exécution ci-dessus montre qu'une seule opération $scan_i$ et deux jointures peuvent être actives à un instant donné. Considérons la chaîne de pipe Pipe $scan_N$ - probe J_{N-1} -$build_N$ End_Pipe. Avant l'exécution de cette chaîne de pipeline, la table de hachage de la $jointure_{N-1}$ a été construite à l'étape précédente. Les tuples produits par l'opération $scan_N$ sont immédiatement utilisés pour sonder (probe) la table de hachage de la $jointure_{N-1}$ afin de produire le résultat de la $jointure_{N-1}$. Ces tuples produits par la $jointure_{N-1}$ sont à leur tour utilisés pour construire la table de hachage de la $jointure_N$. La mémoire occupée par la table de hachage de la $jointure_{N-1}$ peut être libérée uniquement après que la jointure ait consommé tous les tuples produits par l'opération $scan_N$. Donc, quelque soit l'instant considéré, l'espace mémoire maximum occupé par les tables de hachage est l'espace requis pour exécuter *deux jointures consécutives*. Ce plan d'exécution s'adapte donc bien en cas de limitation de mémoire dans la mesure où il requiert, uniquement, l'espace mémoire occupé par deux jointures consécutives.

3. 2 Arbre linéaire droit

La figure IV.7 montre une requête composée de N jointures, représentée par un arbre linéaire droit et son graphe de dépendance associé.

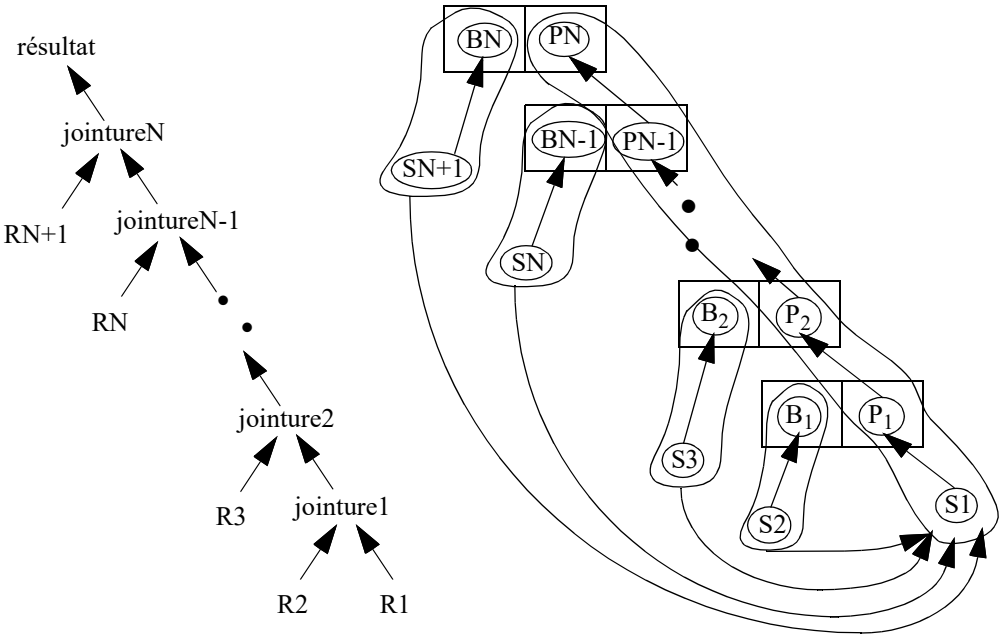

Figure IV.7 : Arbre linéaire droit et graphe de dépendance associé [SCH 90]

A partir de ce plan, nous pouvons remarquer que tous les opérateurs $scan_i$ avec $i \in [2, N]$, et les opérateurs $build_i$ avec $i \in [1, N]$ peuvent s'exécuter en parallèle. Après avoir exécuté ces opérateurs, l'opérateur $scan_1$ et l'ensemble des opérateurs $probe_i$ avec $i \in [1, N]$ peuvent s'exécuter en pipeline. Le plan d'exécution parallèle suivant peut être engendré :

```
Seq
    Par
        Pipe    scan2       -    build1      End_Pipe
        Pipe    scan3       -    build2      End_Pipe
        Pipe    scan4       -    build3      End_Pipe
        •           •           •           •
        Pipe    scanN+1     -    buildN      End_Pipe
    End_Par;
    Pipe scan1 - probe J1 - probe J2- • • • probe JN End_Pipe;
End_Seq
```

Ce plan d'exécution montre qu'il est possible d'obtenir un très haut niveau de parallélisme avec les arbres linéaires droits. Cependant, cette stratégie requiert beaucoup de mémoire pour stocker les N tables de hachage pendant l'exécution de la requête.

Il existe plusieurs stratégies permettant d'exploiter les potentialités des arbres linéaires droits quand la mémoire est limitée. La stratégie, nommée " static right-deep scheduling " [SCH 90] consiste à découper l'arbre linéaire droit (figure IV.7) en plusieurs sous-arbres disjoints de manière à ce que toutes les tables de hachage d'un sous-arbre puissent tenir en mémoire (figure IV.8). Le résultat d'un sous-arbre est écrit sur disque et servira de relation de base pour le sous-arbre suivant. Si seulement la moitié des tables de hachage peuvent tenir en mémoire, alors le plan d'exécution suivant est obtenu sur un arbre linéaire droit contenant quatre jointures :

```
Seq
    Par
        Pipe  scan2  -  build1  End_Pipe
        Pipe  scan3  -  build2  End_Pipe
    End_Par;
    Pipe scan1 - probe J1 - probe J2 End_pipe;
    Par
        Pipe  scan4  -  build3  End_Pipe
        Pipe  scan5  -  build4  End_Pipe
    End_Par;
    Pipe probe J3 - probe J4 End_Pipe;
End_Seq
```

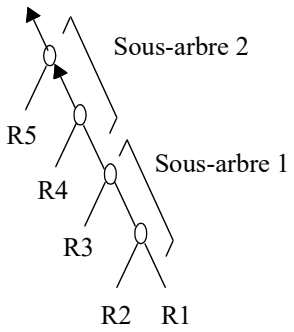

Figure IV.8 : Arbre linéaire droit découpé

Une autre stratégie [SCH 90], appelée " dynamic bottom-up ", découpe dynamiquement l'arbre linéaire droit. D'abord, l'opérateur $scan_2$ est exécuté et les tuples produits par cet opérateur sont utilisés pour construire la première table de hachage. Une fois que ces deux opérateurs ont terminé leur exécution, le gestionnaire de mémoire teste s'il y a assez de mémoire pour exécuter l'opérateur $scan_3$ et l'opérateur $build_2$ de la deuxième jointure. Si c'est le cas, alors ces deux opérateurs sont exécutés. Cette procédure est itérée tant que l'espace mémoire restant est suffisant. Si toutes les tables de hachage peuvent tenir en mémoire, alors la chaîne de pipeline Pipe $scan_1$ - probe J_1 - probe J_2- ... - probe J_N End_Pipe est exécutée. Dans le cas contraire, si le $scan_{i+1}$ n'a pas pu être exécuté par manque de mémoire, alors la chaîne de pipeline Pipe $scan_1$ - probe J_1 - probe J_2- ... - probe J_{i-1} End_Pipe est exécutée. Le résultat de cette chaîne de pipeline est stocké dans une relation S'1. Une fois cette chaîne de pipeline terminée, le processus est réitéré avec $scan_{i+1}$ comme première opération.

Cette méthode dynamique a l'avantage de déterminer exactement le nombre de relations de base pouvant tenir en mémoire. Cependant, elle sérialise les chaînes de pipeline Pipe $scan_i$ - $build_{i-1}$ End_pipe.

3. 3 Arbre bushy

La représentation la plus riche en possibilités d'ordonnancement pour une requête composée de plusieurs jointures est l'arbre bushy. La figure IV.9 montre un arbre bushy composé de sept jointures.

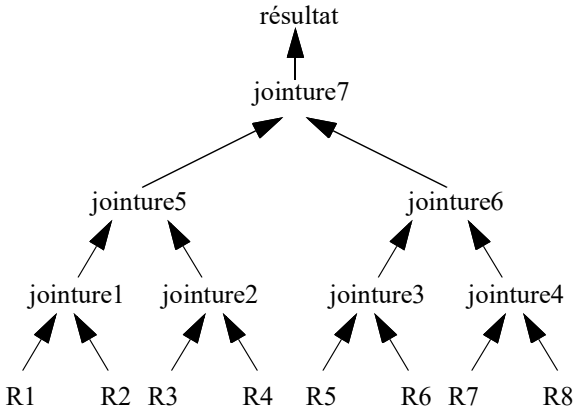

Figure IV.9 : Arbre bushy comportant 7 jointures

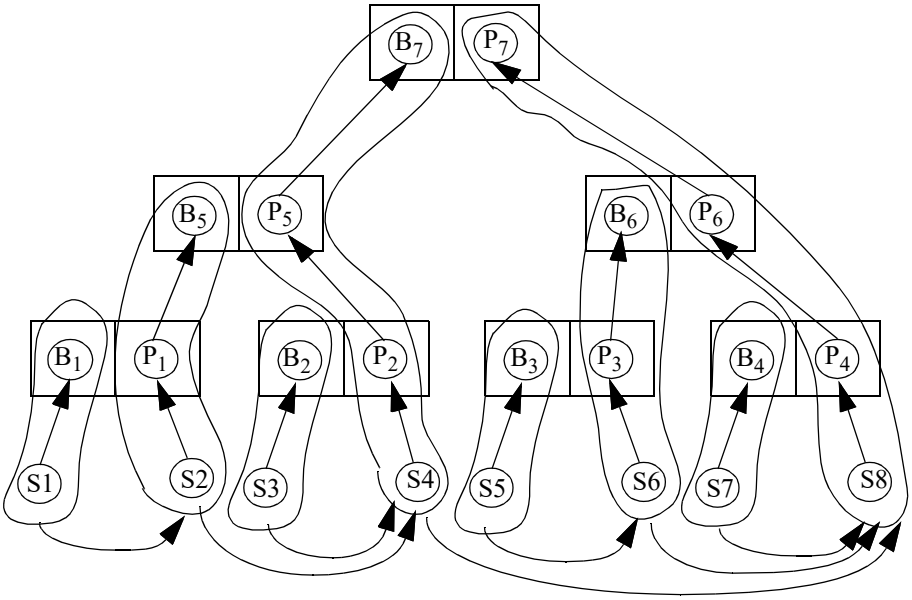

Figure IV.10 : Graphe de dépendance associé à l'arbre bushy ci-dessus [SCH 90]

En suivant les flèches arrondies (figure IV.10) entre les différents sous-graphes, nous pouvons observer que la chaîne de dépendance la plus longue comprend les opérateurs scan S1, S2, S4 et S8. Les opérateurs de ces sous-graphes doivent s'exécuter de manière séquentielle afin de maximiser le processus de flot de données. En effet, ceci évite de stocker des résultats temporaires sur disque. Un plan d'exécution associé au graphe de dépendance de la figure IV.9 contient donc au moins quatre étapes E1, E2, E3 et E4 :

```
Seq
   (E1)                                                          Par
      Pipe        scan1        -        build1                 End_Pipe
      Pipe        scan3        -        build2                 End_Pipe
      Pipe        scan5        -        build3                 End_Pipe
      Pipe        scan7        -        build4                 End_Pipe
   End_Par;
   (E2)                                                          Par
      Pipe     scan2    -    probe    J1-    build5            End_Pipe
      Pipe     scan6    -    probe    J3     -    build6       End_Pipe
   End_Par;
   (E3) Pipe  scan4  -  probe  J2  -  probe  J5  -build7        End_Pipe
   (E4) Pipe  scan8  -  probe  J4  -  probe  J6  -  probe  J7   End_Pipe
End_Seq
```

Nous pouvons remarquer qu'il est possible de déplacer la chaîne de pipeline Pipe $scan_7$ - $build_4$ End_Pipe. Cette chaîne peut être exécutée en parallèle avec (E2) ou (E3) sans violer aucune dépendance. Ceci montre que les possibilités d'ordonnancement sont nombreuses pour un arbre bushy, par rapport à l'ordonnancement des arbres linéaires gauches et des arbres linéaires droits.

L'analyse des stratégies d'exécution correspondant aux structures arborescentes associées à des requêtes montre l'impact du choix de l'arbre sur le plan d'exécution obtenu. De plus, l'analyse des performances proposée par Schneider [SCH 90] a démontré que la stratégie associée à l'arbre linéaire droit, en utilisant la jointure par hachage et sans contrainte mémoire, s'adapte bien à une exploitation maximale du parallélisme dans les systèmes de bases de données parallèles.

Cependant, dans le cas où toutes les tables de hachage de l'arbre linéaire droit ne tiennent pas en mémoire, [SCH 90] propose de découper l'arbre en plusieurs sous-arbres disjoints de manière que la somme des tailles de toutes les tables de chaque sous-arbre tienne en mémoire. Les résultats temporaires T1, T2,...,Tn seront stockés sur disques entre deux phases[1] (figure IV.8). L'inconvénient de cette approche est que le nombre des sous-arbres augmente avec le nombre de relations de base qui ne tiennent pas en mémoire. Ainsi, cette méthode réduit la chaîne du pipeline et augmente le temps de réponse. Deux solutions ont été proposées, l'une basée sur les arbres linéaires droits segmentés [CHE 92b], et l'autre basée sur les arbres Zig-zag [ZIA 93]. Ces deux solutions sont développées ci-après.

3. 4 Arbre linéaire droit segmenté

Dans [CHE 92b] est étudiée une stratégie d'exécution associée à un arbre linéaire droit segmenté tenant compte de la limitation mémoire. L'idée de base de cette stratégie est d'explorer un espace de recherche plus large que les arbres linéaires droits compris entre les arbres linéaires droits et les arbres bushy. Un arbre linéaire droit segmenté (figure IV.11) est un arbre bushy composé d'un ensemble de sous-arbres linéaires droits. Un arbre linéaire droit segmenté est similaire à un arbre linéaire droit, dans le sens où tous les noeuds d'un segment s'effectuent en pipeline, cependant la relation résultat d'un segment peut être opérande soit d'un opérateur build soit d'un opérateur probe dans le prochain segment. Lors de l'exécution d'un segment, la relation résultat temporaire d'un segment est écrite sur disque.

1. Une phase correspond à l'exécution d'un sous-arbre.

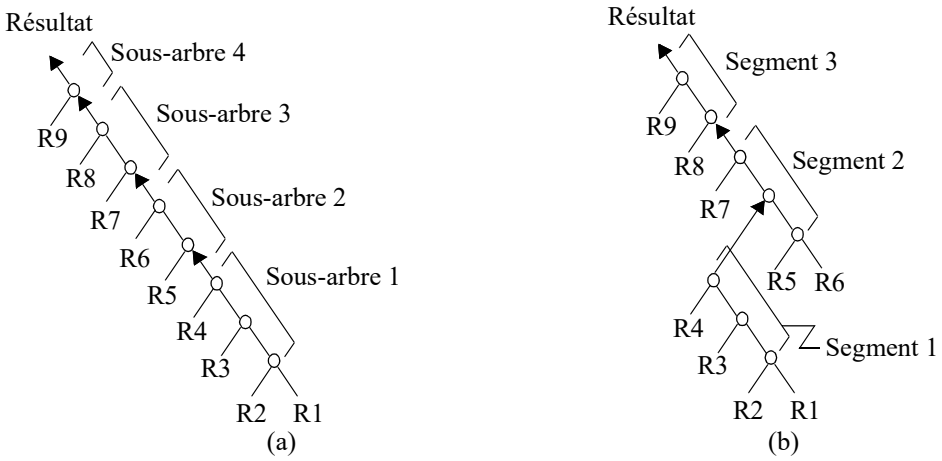

Figure IV.11 : Arbre linéaire droit découpé (a) et arbre linéaire droit segmenté (b)

La construction d'un arbre linéaire droit segmenté est basée sur une heuristique. Celle-ci calcule le nombre de segments et choisit les relations opérandes des opérateurs build et les relations opérandes des opérateurs probe afin que toutes les tables de hachage de chaque segment puissent tenir en mémoire. Le nombre de segments estimé m, est choisi pour être le plus proche possible du nombre de segments nécessaires afin que toutes les relations de base de chaque segment puissent tenir en mémoire. Ce nombre de segments est calculé à partir de la taille des relations de base et de la taille mémoire. Le calcul de m est donné ci-dessous où |Ri| est la taille d'une relation de base en octets, M la taille mémoire par processeur, q le nombre de relations de base, N le nombre de processeurs et $\lceil x \rceil$ représente la partie entière supérieure de x.

$$m = \left\lceil \sum_{i=1}^{q} \frac{|Ri|}{N \times M} \right\rceil$$

Après avoir calculé m, on peut alors estimer le nombre de relations de base par segment $k = \lceil q / m \rceil$. Ensuite, les relations d'un segment sont sélectionnées, une par une, jusqu'à ce que l'une des deux conditions suivantes soit vérifiée :

- les k relations sont déterminées,
- la taille totale des relations opérandes des opérateurs build devient plus grande que la taille de la mémoire.

Le choix des relations opérandes d'un segment est basé sur deux heuristiques : " Minimal Work " et " Balanced Consideration " présentées ci-dessous.

3. 4. 1 Heuristique " Minimal Work " MW

L'objectif de MW est de sélectionner les relations d'un segment une par une de manière à ce que la charge totale de travail (en terme de temps de réponse total) du segment en construction soit minimale. C'est-à-dire qu'à partir d'un ensemble de relations et du nombre k, l'heuristique détermine une séquence de k relations opérandes des opérateurs build et une relation S opérande de l'opérateur probe de la première jointure telle que le temps de réponse total soit minimal. Par exemple, si l'on choisit la deuxième relation utilisée par un opérateur build R_{h2} d'un segment h, alors le temps de réponse totale TT (R_{h1}, S, R_{h2}) <= TT (R_{h1}, S, R_i) $\forall$ $R_i \in$ {relations non affectées à un segment}. Les relations d'un segment sont sélectionnées en suivant l'ordre : $R_{h1} \rightarrow S \rightarrow R_{h2} \rightarrow R_{h3} \ldots \rightarrow R_{hk}$.

La stratégie MW associée aux arbres linéaires droits segmentés améliore souvent le temps de réponse par rapport à la stratégie associée aux arbres linéaires droits découpés SADD [CHE 92b]. Cependant, les performances de la stratégie MW ne sont pas stables, dans la mesure où, dans certains cas, elles sont beaucoup moins bonnes que celles de SADD. Cette instabilité provient du fait que l'heuristique MW a tendance à sélectionner les plus petites relations pour les premiers segments et les plus grandes relations pour les derniers segments. Cette sélection entraîne d'une part, que ces derniers ne peuvent pas contenir les k relations R_{h1}, R_{h2}, ...R_{hk} et a pour conséquence que le nombre de segments se trouve être plus élevé que le nombre estimé. D'autre part, les premiers segments contenant les relations R_{h1}, R_{h2}, ..., R_{hk}, n'utilisent pas entièrement tout l'espace mémoire.

3. 4. 2 Heuristique " Balanced Consideration " BC

Afin d'éliminer l'instabilité de l'heuristique MW, Chen et al. [CHE 92b] proposent une nouvelle heuristique qui diminue la tendance à sélectionner les plus petites relations pour les premiers segments. Dans l'heuristique BC, une pénalité P et un bénéfice B sont définis pour chaque segment. La pénalité est définie comme la charge du segment en terme de temps de réponse total, et le bénéfice comme la réduction en taille après l'exécution de ce segment (i. e., ($|S| + |R_{h1}| + |R_{h2}| + \ldots + |R_{hk}|$) - $|I_k|$ où I_k est la relation résultat du segment). Les relations sont sélectionnées de manière à ce que la fonction Y = P - w B soit minimale (w est un facteur de poids[1]). La considération d'un équilibre

entre la pénalité et le bénéfice élimine la tendance à choisir les plus petites relations pour les premiers segments.

L'avantage d'un arbre linéaire droit segmenté en cas de limitation mémoire, par rapport à un arbre linéaire droit, est que le processus de génération de l'arbre est plus flexible étant donné que l'espace de recherche est plus large. Cette flexibilité permet à un arbre linéaire droit segmenté d'avoir des relations déduites comme opérandes des opérateurs build. De plus, si une ou plusieurs relations déduites ont une taille inférieure à celle des relations de base, alors le plan d'exécution engendré à partir de l'arbre droit segmenté peut comporter moins de phases que le plan généré à partir d'un arbre linéaire droit. La diminution du nombre de phases améliore donc les performances [CHE 92b] en terme de temps de réponse.

3. 5 Arbre zig-zag

L'approche de Ziane et al. [ZIA 93] prenant en compte le problème de limitation mémoire consiste à explorer un espace de recherche assez large pour inclure une structure intermédiaire entre les arbres linéaires gauches et les arbres linéaires droits. Dans les arbres linéaires droits découpés et les arbres linéaires droits segmentés, la relation temporaire d'un sous-arbre droit ou d'un segment est stockée sur disque. Ziane et al. [ZIA 93] proposent une alternative pour éviter de stocker ces relations temporaires sur disque en introduisant les arbres zig-zag.

Il est possible d'éliminer les E/S d'un arbre linéaire droit découpé en gardant en mémoire la relation temporaire de chaque sous-arbre. Ceci implique qu'il faut prendre en compte la taille des relations temporaires pour découper l'arbre linéaire droit. L'inconvénient de cette méthode est que, dans un sous-arbre, moins de relations opérandes des opérateurs build peuvent tenir en mémoire. Par conséquent, le nombre de sous-arbres linéaires droit se trouve augmenté. Cette stratégie sacrifie une partie du parallélisme en augmentant le nombre de phases du plan d'exécution. Cependant, il est possible d'améliorer le temps de réponse en éliminant les E/S. Avec la stratégie précédente, les relations temporaires T1, T2, ..., Tn sont utilisées comme opérandes d'un opérateur probe. Ces relations T1, T2, ..., Tn peuvent être aussi utilisées comme opérandes d'un opérateur build. Ziane et al. [ZIA 93] transforment ainsi un

1. w = temps de lecture disque d'un tuple + temps pour appliquer la fonction de répartition + temps pour envoyer un tuple sur le réseau + temps pour recevoir un tuple du réseau + temps pour appliquer la fonction de hachage + temps pour insérer un tuple dans une table de hachage

arbre linéaire droit découpé en un arbre appelé zig-zag (figure IV.12). La stratégie d'exécution associée à un arbre zig-zag consiste à consommer les relations de base (R1, R(i1)+1, ..., R(in)+1) en pipeline et à appliquer l'opérateur build sur les relations temporaires T1, T2, ..., Tn pour construire des tables de hachage.

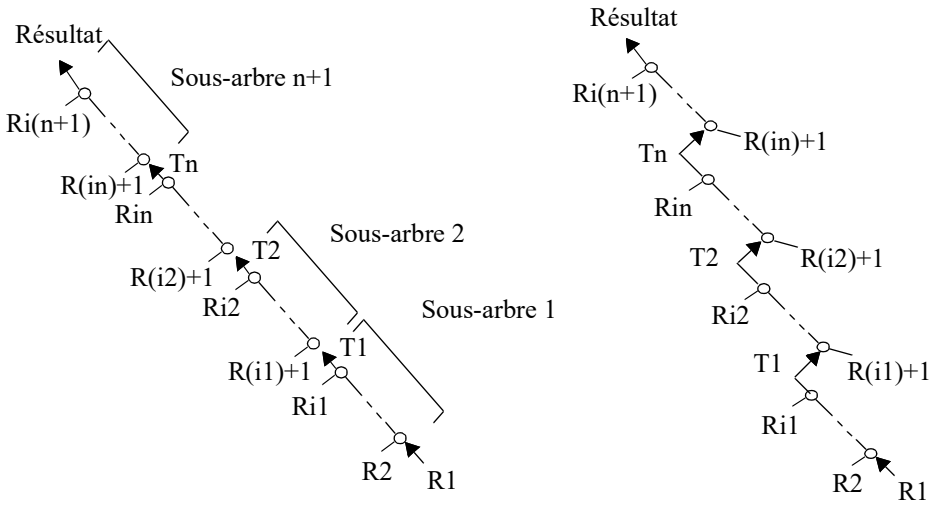

Figure IV.12 : Arbre linéaire droit découpé et arbre zig-zag [ZIA 93]

L'espace mémoire nécessaire à l'exécution d'un arbre linéaire droit découpé est l'espace mémoire maximum requis par chaque phase associée à un sous-arbre. Dans le cas où toutes les relations temporaires T1, T2, ..., Tn sont stockées en mémoire, la taille mémoire TM occupée par l'arbre linéaire droit découpé ADD est la suivante [ZIA 93] :

$$TM\ (ADD) = Max\ (T1+(\sum_{j=2}^{i_1} Rj),\ T1+T2+(\sum_{j=i1+1}^{i_2} Rj),\ ...,\ Tn+(\sum_{j=in+1}^{i_{n+1}} Rj))$$

La taille mémoire occupée par l'arbre zig-zag est la suivante [ZIA 93] :

$$TM\ (Zig\text{-}Zag) = Max\ (T1+(\sum_{j=2}^{i_1} Rj),\ T1+T2+(\sum_{j=i1+2}^{i_2} Rj),...,Tn+(\sum_{j=in+2}^{i_{n+1}} Rj))$$

Pour chaque phase, mise à part la première, l'arbre linéaire droit découpé a besoin de charger en mémoire une relation de plus que l'arbre zig-zag. Par

conséquent, le nombre de phases pour exécuter un arbre linéaire droit découpé est plus important que le nombre de phases de l'arbre zig-zag.

Dans le cas où les relations temporaires de l'arbre linéaire droit découpé sont stockées sur disque, la taille mémoire occupée par l'arbre linéaire droit découpé TM2 est plus petite que la taille mémoire occupée par l'arbre zig-zag :

$$\text{TM2 (AD)} = \text{Max}\left(\left(\sum_{j=2}^{i_1} Rj\right), \left(\sum_{j=i_1+1}^{i_2} Rj\right), \ldots, \left(\sum_{j=i_n+1}^{} Rj\right)\right)$$

L'analyse des performances proposée par [ZIA 93] montre que la stratégie d'exécution associée au format d'arbre zig-zag améliore le temps de réponse par rapport à l'arbre linéaire droit découpé quand les relations temporaires sont stockées sur disque.

En conclusion, l'utilisation des arbres zig-zag, en cas de limitation mémoire, évite de stocker sur disque les relations temporaires T1, T2, ..., Tn de chaque sous-arbre. De plus, la stratégie associée à ce format d'arbre améliore le temps de réponse par rapport aux deux stratégies associées à l'arbre linéaire droit découpé, correspondant aux cas où les relations temporaires T1, T2, ..., Tn sont stockées ou non sur disque [ZIA 93].

4 Stratégies de parallélisation inter-opération à deux phases

Dans cette section, nous décrivons deux stratégies de parallélisation inter-opération à deux phases. La première consiste à allouer deux fragments (i.e. ensemble d'opérations) de manière à maximiser l'utilisation des ressources du système parallèle en termes de bande passante en entrées/sorties et de processeurs [HON 92]. La seconde stratégie s'appuie sur les méthodes de recherche de chemin critique [CHR 92, CON 67, COS 93] pour déterminer un ordonnancement parallèle des opérations en tenant compte du nombre de processeurs [HAM 93a, HAM 95b].

4. 1 Méthode d'ordonnancement adaptatif de XPRS

Dans le système XPRS (eXtended Postgres on Raid and Sprite) [STO 88] basé sur un modèle d'architecture à mémoire commune, le processus de traitement d'une requête est constitué de deux phases (figure IV.13). La première phase engendre un plan d'exécution séquentiel. Ce plan d'exécution est représenté par un arbre bushy où chaque noeud est un opérateur de l'algèbre relationnelle de base. Dans la deuxième phase, ce plan est décomposé en

fragments. Un fragment est un ensemble d'opérations qui ne contient aucun arc bloquant (i.e. une chaîne de pipeline du plan d'exécution). Un arc bloquant entre deux opérations signifie que l'opération correspondant à l'extrémité finale doit attendre que l'opération représentée par l'extrémité initiale ait produit tous ses tuples avant de commencer son exécution. Par exemple, il existe un arc bloquant entre les opérateurs build et probe d'une même opération. Les fragments sont utilisés comme unité d'exécution parallèle et ils seront aussi appelés tâches.

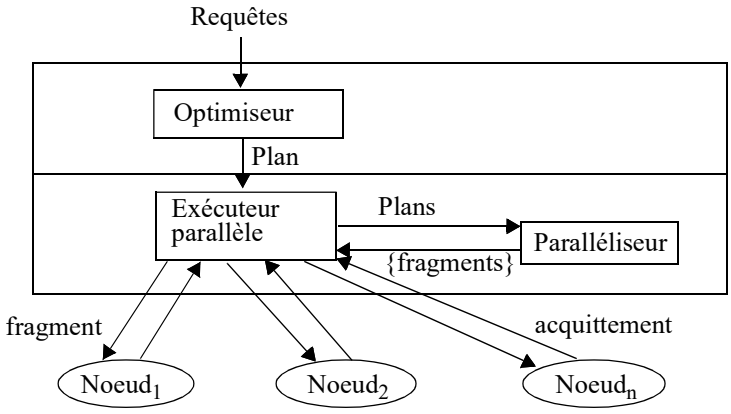

Figure IV.13 : Architecture logiciel du traitement de requêtes dans XPRS [HON 92]

Après avoir identifié tous les fragments, le processus de parallélisation consiste à trouver un *ordonnancement* entre les différents fragments et à choisir un nombre de processeurs pour chaque fragment.

Dans la suite, nous décrivons d'abord, une classification des tâches en " IO-bound " et " CPU-bound ", et une méthode de calcul du point d'équilibre IO-CPU. Ensuite, nous étudions le mécanisme d'ajustement dynamique du degré de parallélisme. Enfin, nous montrons comment ces méthodes sont intégrées dans l'algorithme d'ordonnancement adaptatif.

4. 1. 1 Tâches IO-bound et CPU-bound

La stratégie proposée par [HON 92] consiste à trouver un ordonnancement des tâches de manière à maximiser l'utilisation des processeurs et des disques, et à minimiser le temps de réponse. Pour cela, il définit deux types de tâche (figure IV.14) : les tâches IO-bound (bornées par les entrées/sorties) et les tâches CPU-bound (bornées par le nombre de processeurs).

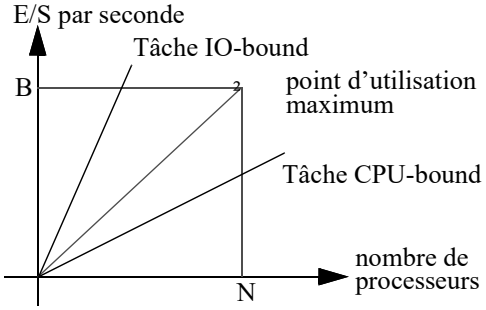

Figure IV.14 : Tâches IO-bound et CPU-bound [HON 92]

Supposons qu'une tâche f_i soit exécutée de manière séquentielle, et qu'elle génère des entrées/sorties à un coût C_i. Si cette tâche est exécutée sur x processeurs, son coût en entrées/sorties devient : $IO_i(x) = C_i.x$.

Supposons maintenant que la bande passante soit de B entrées/sorties par seconde et que le nombre total de processeurs soit N. Une tâche est dite IO-bound si $C_i > B/N$. Une tâche est définie CPU-bound dans le cas contraire. La fonction $y = IO_i(x)$, figure IV.14, est bornée par la bande passante B et par le nombre de processeurs N. Pour une tâche IO-bound, le nombre de processeurs maximal est B / C_i puisqu'on est borné par B. Si l'on dépasse ce nombre, alors certains processeurs seront inactifs. Par contre, une tâche CPU-bound n'est pas bornée par B, donc son nombre de processeurs maximal est uniquement borné par N. En général, le nombre de processeurs maximal d'une tâche, noté maxp, pour une tâche f_i est : $maxp(f_i) = \min(B/C_i, N)$.

Si l'on souhaite maximiser l'utilisation des ressources (disques et processeurs), alors il faut que le système soit exploité au point de coordonnées (N, B). Il est clair que si l'on exécute une seule tâche à la fois en exploitant le parallélisme intra-opération, on gaspille soit une partie de la bande passante, soit une partie des processeurs. L'idée consiste à exécuter simultanément une tâche IO-bound et une tâche CPU-bound afin que le système atteigne le point (N, B).

Quand on exécute deux tâches f_i et f_j avec un degré de parallélisme x_i et x_j (figure IV.15), le système travaille au point $(x_i + x_j, C_i.x_i + C_j.x_j)$. Il est possible de maximiser l'utilisation des ressources si l'on choisit x_i et x_j vérifiant les équations ci-dessous (a) :

$$\begin{aligned} x_i + x_j &= N \\ C_i.x_i + C_j.x_j &= B \end{aligned} \quad (a) \qquad \begin{aligned} x_i &= (B - C_j.N)/(C_i - C_j) \\ x_j &= (C_i.N - B)/(C_i - C_j) \end{aligned} \quad (b)$$

Ces équations admettent une solution unique *si une tâche est IO-bound et l'autre CPU-bound* (i.e. $C_i \neq C_j$). La résolution des équations (a) est donnée ci-dessus (b). L'auteur appelle (x_i, x_j) le point d'équilibre IO-CPU pour les tâches f_i et f_j.

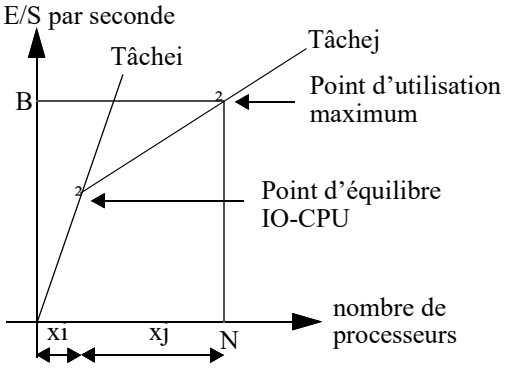

Figure IV.15 : Point d'utilisation maximum

4. 1. 2 Ajustement dynamique du parallélisme

Nous avons présenté, précédemment, une méthode pour calculer le point d'équilibre IO-CPU pour deux tâches. Cependant, lorsque l'une des deux tâches se termine, une partie des ressources (processeurs ou disques) reste inutilisée. Nous montrons maintenant, comment [HON 92] peut dynamiquement ajuster (augmenter ou diminuer) le nombre de ressources d'une tâche.

Une implantation du parallélisme intra-opération dans le système parallèle XPRS est décrite dans [HON 91]. Ce parallélisme s'appuie sur deux méthodes de répartition : la répartition circulaire effectuée page par page " page partitioning "et la répartition par intervalles. Dans la répartition circulaire par page, un sous-ensemble de pages est assigné à chaque processeur participant au calcul. Par exemple, si l'on considère un ensemble de n processeurs, alors le processeur i effectuera son opération avec l'ensemble des pages p / p mod n = i. Si l'on fixe n = 3 alors les pages {0, 3, 6, ...} sont affectées au processeur 0, les pages {1, 4, 7, ...} sont affectées au processeur 1 et les pages {2, 5, 8, ...} sont affectées au processeur 2.

Dans la répartition par intervalles, les relations sont réparties suivant la valeur d'un attribut. Par exemple, on distribue la relation employés sur deux processeurs. Un processeur travaille avec les tuples employés.salaire < 3000 et l'autre processeur travaille avec les tuples employés.salaire >= 3000.

Hong [HON 92] propose pour chacune des approches de répartition, une méthode d'ajustement dynamique du degré de parallélisme.

Méthode d'ajustement dynamique pour la répartition circulaire

Le processeur maître envoie d'abord un signal à tous les processeurs esclaves participant au calcul. Après avoir reçu le signal, chaque processeur esclave (i = 0, 1, ..., n-1) envoie au processeur maître le dernier numéro de la page qu'il a traitée. Le processeur maître, après avoir reçu tous ces numéros, calcule le maximum maxpage et l'envoie, ainsi que le nouveau degré de parallélisme n', à chaque processeur. On distingue deux cas (figure IV.16) :

1- si n' > n alors le processeur maître déclenche n'-n processus sur les processeurs supplémentaires. Ces processus commencent la lecture de la relation à partir d'un numéro de page supérieur à maxpage. Quant aux n processeurs participant au calcul, après avoir reçu maxpage et n', ils continuent leur exécution, avec les pages p / p mod n = i, jusqu'à ce que toutes les pages inférieures à maxpage soient traitées. A partir de ce point, un processeur i effectuera son opération avec les pages p / p > maxpage et p mod n' = i ;

2- si n' < n, alors les processeurs i ∈[n', n-1], après avoir traité toutes les pages jusqu'à maxpage, signalent au processeur maître qu'ils ont fini leur traitement et qu'ils sont libres. Les autres processeurs continuent leur exécution avec les pages p / p mod n = i jusqu'à ce que toutes les pages inférieures à maxpage soient traitées. A partir de ce point, les processeurs i ∈ [0, n'-1] lisent toutes les pages p / p > maxpage et p mod n' = i.

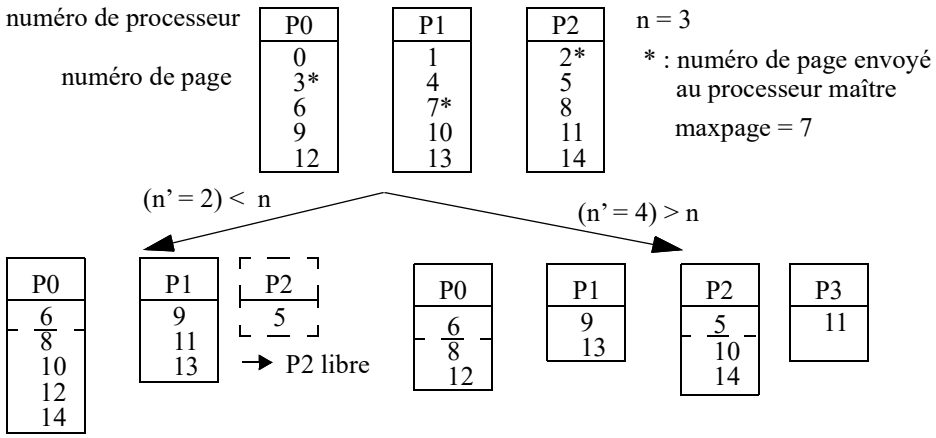

Figure IV.16 : Ajustement du parallélisme pour la répartition circulaire par page

Méthode d'ajustement pour la répartition par intervalles

Le processeur maître envoie un signal à tous les processeurs participant au calcul. Après avoir reçu le signal, chaque processeur esclave envoie l'intervalle de valeurs restant à traiter. Par exemple, si l'intervalle [1, h] est assigné à un processeur esclave et que la valeur courante examinée par ce processeur est c, alors ce processeur envoie au processeur maître l'intervalle [c, h]. Une fois tous les intervalles reçus par le processeur maître, celui-ci redistribue les intervalles sur les n' processeurs. Après l'ajustement du parallélisme, un processeur esclave peut avoir plusieurs intervalles à examiner.

4. 1. 3 Algorithme d'ordonnancement adaptatif

L'idée principale de l'algorithme d'ordonnancement adaptatif est d'utiliser la méthode d'ajustement dynamique pour que le système travaille toujours au point d'équilibre IO-CPU. Si l'on considère un ensemble de tâches exécutables $S = \{f_1, f_2, ..., f_n\}$, et que l'on note par :

- T_i le temps d'exécution séquentiel de f_i,
- $T_{intra}(f_i)$ le temps de f_i en utilisant uniquement le parallélisme intra-opération,
- $T_{inter}(f_i, f_j)$ le temps de réponse pour exécuter f_i et f_j à leur point d'équilibre IO-CPU,

nous avons alors :

- $T_{intra}(f_i) = T_i / maxp(f_i)$ avec $maxp(f_i) = min(B / C_i, N)$ (cf. 4. 1. 1 page 174)
- $T_{inter}(f_i, f_j) = min(T_i / x_i, T_j / x_j) + (T_{ij} / maxp_{ij})$,
- $min(T_i / x_i, T_j / x_j)$ = temps pour que f_i ou f_j finisse,
- $T_{ij} / maxp_{ij}$ = temps pour que la tâche restante termine son exécution après avoir réajusté le parallélisme.

$$maxp_{ij} = \begin{cases} maxp(f_i) & \text{si } T_i / x_i > T_j / x_j \\ maxp(f_j) & \text{sinon} \end{cases}$$

$$T_{ij} = \begin{cases} T_i - T_j.x_i / x_j & \text{si } T_i / x_i > T_j / x_j \\ T_j - T_i.x_j / x_i & \text{sinon} \end{cases}$$

Dans l'algorithme d'ordonnancement adaptatif de [HON 92] (figure IV.17) le parallélisme inter-opération de deux tâches t_1 et de t_2 est exploité uniquement si le temps de réponse de t_1 et t_2, exécutées en parallèle, est meilleur que

le temps de réponse de t_1 suivie de t_2. La stratégie utilisée, pour choisir une paire de tâches IO-bound et CPU-bound, consiste à sélectionner la plus grande tâche IO-bound et la plus grande tâche CPU-bound. De cette manière, le système peut être exploité au plus proche du point d'utilisation maximum des ressources.

```
Algorithme d'ordonnancement adaptatif
Début
  diviser S en Sio et Scpu tel que S = Sio ∪ Scpu, où
  Sio contient toutes les tâches IO-bound et Scpu contient toutes les tâches CPU-bound;
  tant que S≠ ∅ faire
    s'il n'existe pas de tâche en cours d'exécution alors
      si Sio ≠ ∅ alors choisir une tâche t1 ∈ Sio; S1 := Sio;
      sinon choisir une tâche t1 ∈ Scpu; S1 := Scpu;
      fin si
    fin si
    choisir une tâche t2 ∈ S - S1 telle que Tinter (t1, t2) < Tintra (t1) + Tintra (t2);
    si t2 n'existe pas alors
      si t1 est une nouvelle tâche alors exécuter t1 avec le degré de parallélisme maxp(t1)
      sinon ajuster le degré de parallélisme de t1 à maxp (t1);
      fin si
      attendre la fin de t1; S := S-{t1}; /* quand une tâche est éliminée de S elle est
                                  aussi implicitement éliminée de Sio ou de Scpu */
    sinon
      calculer le point d'équilibre IO-CPU (x1,x2) entre t1 et t2;
      si t1 est une nouvelle tâche alors exécuter t1 avec le degré de parallélisme x1;
      sinon ajuster le parallélisme de t1 à x1;
      fin si
      exécuter t2 avec le degré de parallélisme x2;
      attendre la fin de t1 ou de t2;
      si t1 finit en premier alors S := S-{t1}; S1 := S - S1;
      sinon S := S-{t1};
      fin si
    fin si
  fin tant que
Fin
```

Figure IV.17 : Algorithme d'ordonnancement adaptatif

L'algorithme figure IV.17 peut être facilement étendu à une suite continue de tâches $\{f_1, f_2, ..., f_n\}$ au lieu d'un ensemble de tâches fixées. En d'autres termes, cet algorithme peut être étendu à un algorithme en ligne. Pour cela, il suffit que les ensembles S_{io} et S_{cpu} soient gérés en file. Quand une tâche IO-bound ou CPU-bound arrive, alors elle est classée dans son ensemble correspondant selon son nombre d'entrées/sorties. Les tâches IO-bound sont classées de manière descendante, alors que les tâches CPU-bound le sont de manière ascendante.

4. 2 Méthode d'ordonnancement parallèle basée sur la recherche d'un chemin critique

4. 2. 1 Graphe de dépendance des opérations

Dans cette section, nous présentons un exemple de transformation d'un graphe de résolution en un graphe de dépendance introduit par Schneider [SCH 90]. Le graphe de dépendance (figure IV.18) est basé sur l'utilisation de l'un des algorithmes de jointure par hachage [DEW 84, SCH 89]. Dans le but d'exploiter le parallélisme pipeline des opérations de différence R - S et d'union R ∪ S, leur second opérande doit être *complètement calculé* avant de commencer leur exécution. Par conséquent, la relation S est utilisée pour construire la table de hachage.

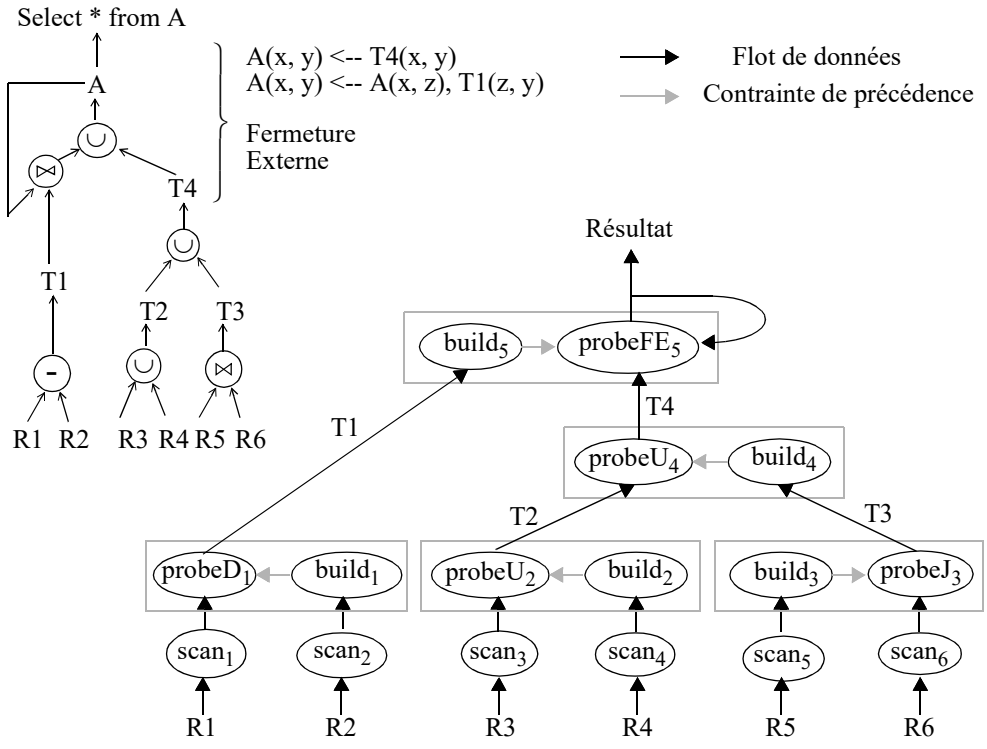

Figure IV.18 : Graphe de dépendance des opérations associé au graphe de résolution

Pour des raisons d'homogénéité avec la méthodologie des systèmes relationnels et de facilité d'implantation, l'évaluation parallèle de requêtes récursives est fondée sur l'une des méthodes d'évaluation basées sur l'approche de

répartition de données comme la méthode semi-naïve ou celle des ensembles magiques [BEE 87]. Dans le graphe de dépendance (figure IV.18) associé au graphe de résolution, chaque opération X_i relationnelle binaire ou de point fixe, sera représentée par les opérations {$build_i$; $probeX_i$}. Dans le cas de l'opérateur de point fixe, l'opération $probeX_i$ a été étendue par l'intégration d'un processus itératif basé, par exemple, sur la méthode semi-naïve.

4. 2. 2 Principe de la méthode d'ordonnancement parallèle

Le but de cette section est d'une part, de montrer comment les méthodes sérielles [CHR 92, COS 93] ont été étendues pour ordonnancer les opérations d'une requête SQL étendu (à la récursion) et, d'autre part, de présenter la stratégie de génération d'un programme parallèle en tenant compte de l'allocation physique des processeurs aux différentes opérations, dans une architecture parallèle à mémoire distribuée.

L'algorithme d'ordonnancement parallèle PSA (Parallel Scheduling Algorithm) de requêtes SQL étendu est constitué de trois phases : une phase d'ordonnancement, une phase de placement des opérations et une phase de génération de programme parallèle.

Ordonnancement des opérations

Le principe de la méthode d'ordonnancement des opérations est d'incrémenter le temps à partir de l'instant 0. A l'instant tp (i.e. temps de déclenchement), on affecte logiquement les processeurs à l'opération de plus haute priorité parmi les opérations prêtes. Une opération prête est une opération vérifiant les conditions suivantes [HAM 93a] :

1- tous les prédécesseurs directs de l'opération sont soit terminés soit en cours d'exécution,

2- si l'opération est un $probeX_i$ alors l'opération $build_i$ correspondante est terminée,

3- l'opération n'utilise pas plus de ressources (i.e. processeurs) que la quantité disponible à l'instant tp.

La règle de priorité appliquée est celle qui ordonne les opérations dans le sens des *dates de fin au plus tard croissantes* [CHR 92, CON 67, COS 93]. Cette heuristique permet de déterminer un *chemin critique* du graphe de dépendance. La méthode d'ordonnancement est constituée de deux étapes :

E1 : calculer les dates de fin au plus tard de chaque opération du graphe de dépendance [HAM 93a]. Chaque opération i du graphe de dépendance sera

étiquetée par un quadruplet (x1, x2, x3, x4) où x1 est le nombre de processeurs, x2 est le temps de réponse local, x3 est le retard et x4 est la date de fin au plus tard. Dans le chapitre III est décrite une méthode analytique pour allouer logiquement un nombre de processeurs économique x1 à une opération relationnelle ou de point fixe. Les équations permettant de définir et de calculer le retard, le temps de réponse et la date de fin au plus tard sont décrites ci-après :

$$TR(i) = \begin{cases} TRL(i) & \text{si } pred(i) = \varnothing \\ TRL(i) + \max(TR(j)) & \text{sinon avec } j \in pred(i) \end{cases}$$

$$RETARD(i) = \begin{cases} TR(i) - TRL(i) & \text{si } succ(i) = \varnothing \\ RETARD(succ(i)) - TRL(i) & \text{sinon} \end{cases}$$

$$D\_f\_tard(i) = \begin{cases} TR(i) & \text{si } succ(i) = \varnothing \\ RETARD(succ(i)) & \text{sinon} \end{cases}$$

E2 : ordonnancer les opérations selon la règle de priorité et les contraintes définies ci-dessus (si deux opérations ont la même date de fin au plus tard alors on prend en compte le retard).

L'algorithme d'ordonnancement consiste à affecter un temps de déclenchement tp à chaque opération du graphe de dépendance en tenant compte du nombre de processeurs. Le principe de l'algorithme est de déterminer l'ensemble P des opérations prêtes à un instant tp en fonction du nombre de processeurs non affectés et des opérations non traitées. Si P est non vide, alors on affecte logiquement des processeurs à l'opération de plus haute priorité et on lui assigne tp comme temps de déclenchement. Dans le cas contraire, si P est vide (le nombre de processeurs restant est insuffisant ou une ou plusieurs opérations " probeX_i " attendent la fin de leurs " $build_i$ " correspondants), alors on détermine le plus petit instant tp tel qu'au moins une opération de l'ensemble des opérations en cours d'exécution à l'instant tp, soit terminée. Le processus d'ordonnancement est réitéré jusqu'à ce que toutes les opérations du graphe de dépendance aient été traitées.

Les notations de l'algorithme d'ordonnancement sont les suivantes :

I : ensemble des opérations du graphe de dépendance,
Liste _E : liste des opérations ordonnées,
S_N : ensemble des opérations non traitées,
O_E : opération élue,

tp : temps de déclenchement,
P : ensemble des opérations prêtes à l'instant tp,
E_O_E : ensemble des opérations en cours de traitement à l'instant tp,
Dmax : nombre de processeurs non utilisés à l'instant tp,
Nb_p : nombre total de processeurs de la machine parallèle,
di : nombre de processeurs alloués à l'opération i.

```
Procédure ordonnancement
Début
    tp := 0; Dmax := nb_esclave; S_N := I; E_O_E := ∅ ;
    tant que S_N<> ∅ faire
        opération_prête (S_N, P, Dmax);
        si P <> ∅ alors
            élection_opération (P, O_E);
            déclenchement (O_E) := tp;
            chaîner (O_E, Liste_E);
            S_N := S_N - {O_E};
            E_O_E := E_O_E ∪ {O_E};
            Dmax := Dmax - nb_processeur (O_E);
        sinon
            term_op (E_O_E, Dmax, tp);
        fin si
    fin tant que
Fin
```

Procédure opération_prête (S_N, P : opération, Dmax : nb_processeur);
Cette procédure détermine l'ensemble des opérations prêtes P à partir de l'ensemble des opérations non traitées S_N, et du nombre de processeurs Dmax.

Procédure term_op (E_O_E : opération, Dmax : nb_processeur, tp : temps);
Cette procédure détermine le plus petit instant tp tel qu'au moins une opération de E_O_E soit terminée. Elle met à jour l'ensemble E_O_E, la variable Dmax et la variable tp.

```
Procédure élection_opération (P : opération, O_E : opération élu
Début
    minima (P, S_P);  /* cette fonction détermine l'ensemble des opératio
                      S_P tel que RETARD (i)+TRL (i)) / i ∈ P soit minimum
    si card (S_P) > 1 al
        minima2 (S_P, O_E); /* cette fonction détermine l'opération élue O
                            tel que RETARD (i) / i ∈ S_P soit minimum
    sinon O_E := première_opération (S_
    fin
Fin
```

Afin d'illustrer l'algorithme d'ordonnancement, nous l'appliquons sur l'exemple de la figure IV.18. L'application sur les données du graphe de dépendance (table 2) engendre un ordonnancement représenté par le diagramme de Gantt figure IV.19.

Table 2 : Données du graphe de dépendance des opérations

	$scan_1$ $probeD_1$	$scan_2$ $build_1$	$scan_3$ $probeU_2$	$scan_4$ $build_2$	$scan_5$ $build_3$	$scan_6$ $probeJ_3$	$build_4$	$probeU_4$	$build_5$	probe FE_5
di	4 5	3 5	3 5	2 5	2 3	4 3	5	5	5	5
TRL (i)	5 5	4 3	4 7	3 2	2 3	4 6	4	9	3	12
retard (i)	15	8	8	3	0	5	15	19	25	28
D_f_tard(i)	25	15	19	8	5	15	19	28	28	40

Dmax = 20 processeurs

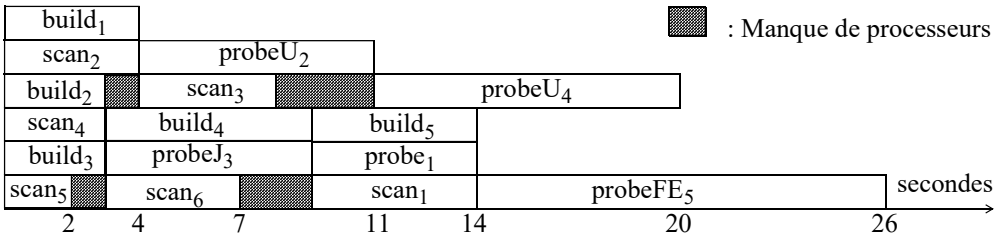

Figure IV.19 : Représentation de l'ordonnancement par le diagramme de Gantt

Placement des opérations

L'algorithme développé ci-dessus ne réalise pas le placement des opérations sur l'ensemble des processeurs de la machine cible. Par définition, le placement d'un programme composé de N opérations (tâches) sur une architecture parallèle disposant de Nb_p processeurs, est une application Alloc : I $\rightarrow$ *P* où I est l'ensemble des opérations et *P,* l'ensemble des processeurs. En d'autres termes, un placement est une allocation physique des processeurs de la machine aux tâches du programme, garantissant une utilisation optimale des ressources ainsi que les meilleures conditions possibles d'exécution du programme [AND 88]. Ce problème NP-complet [CHR 92] consiste à optimiser une fonction coût parmi toutes celles qui permettent de déterminer les caractéristiques du placement optimal recherché (temps de réponse, temps de communication inter-processeur, ...). Pour éviter l'obtention de solutions triviales, il est parfois nécessaire de faire intervenir des contraintes d'allocation telles que l'équilibrage de charge entre les processeurs, des ensembles de processeurs " possiblement allouables " aux tâches du programme. La littérature fournit une classification complète des multiples méthodes de résolution de ce problème dont les buts -contradictoires- sont l'efficacité et la performance. On notera les méthodes statiques réalisant le placement du programme lors de sa

compilation, comprenant des algorithmes exacts (proches des énumératifs) et approchés (proches des aléatoires).

Le principe de la méthode de placement des opérations d'une requête SQL repose sur les hypothèses suivantes :

H1 : architecture parallèle à mémoire distribuée,

H2 : homogénéité des processeurs ; tous les processeurs sont identiques,

H3 : architecture non reconfigurable ; la topologie du réseau d'interconnexion est fixe,

H4 : mono-utilisateur ; tous les processeurs de la machine cible alloués aux opérations du programme n'exécutent que celles-ci. En conséquence, il devient possible d'effectuer en même temps l'ordonnancement et le placement ;

H5 : placement statique ; on s'intéresse uniquement aux algorithmes de placement statique.

Pour garantir les meilleures conditions possibles d'exécution, il est nécessaire de respecter certaines contraintes d'allocation. En effet, la répartition des données propre aux systèmes parallèles à mémoire distribuée, oblige certaines tâches à s'exécuter à l'endroit précis où se trouvent les données qu'elles utilisent : c'est le cas de l'opérateur scan qui lit les données sur disque. D'autre part, si le processus d'ordonnancement a détecté qu'un ensemble d'opérations peuvent s'exécuter simultanément, le placement doit éviter de placer ces opérations sur les mêmes processeurs. De plus, si les opérations $\{build_i ; probeX_i\}$ s'exécutent sur des processeurs différents, le système est amené à transférer les données (tables de hachage du build) des processeurs alloués au $build_i$ vers les processeurs alloués au $probeX_i$. Dans le but d'éviter ces communications et éventuellement des E/S, on doit assurer que les opérations $\{build_i ; probeX_i\}$ s'exécutent sur les mêmes processeurs. La prise en compte de cette contrainte nécessite l'extension de l'algorithme d'ordonnancement en ajoutant une condition supplémentaire à la définition d'une opération prête : *4- si l'opération est un $probeX_i$ alors les processeurs alloués à l'opération $build_i$ correspondante sont libres*. Enfin, le processus de placement essaie de maximiser les chaînes de pipeline potentielles (par exemple, les processeurs alloués à l'opération build3 ne devraient pas -a première vue- être identiques à ceux alloués à l'opération $scan_6$; il en est de même pour les opérations $probeU_4$ et $probeJ_3$). Néanmoins, intuitivement, deux opérateurs qui s'exécutent en pipeline ne devraient être placés sur des ensembles de processeurs différents que si le coût de leur communication est inférieur au coût de leur traitement.

Toutes ces contraintes peuvent être définies formellement comme suit :

C0 : contrainte de localité des données (non relâchable) : les opérateurs scan s'exécutent là où se situent les données qu'ils utilisent,

C1 : contrainte de localisation des données (non relâchable) : Alloc ($build_i$) = Alloc ($probeX_i$)

C2 : contrainte de parallélisme indépendant (non relâchable) : Alloc (T_i) ≠ Alloc (T_j), si i ≠ j et s'il existe un instant où T_i et T_j s'exécutent simultanément,

C3 : contrainte de parallélisme pipeline (relâchable) : Alloc ($probeX_i$) ≠ Alloc (pred ($probeX_i$)) et Alloc ($build_i$) ≠ Alloc (pred ($build_i$)).

Cependant, même si la méthode de placement prenait en compte la totalité des contraintes d'allocation, le coût des communications de l'allocation réalisée ne serait pas minimal. D'une part, le coût de communication inter-tâche dépend du mode de communication des données entre les tâches adjacentes composant la requête. D'autre part, selon la topologie du réseau d'interconnexion du système cible, les coûts de communication inter-processeur peuvent être uniformes ou pas. La méthode de placement doit donc aussi minimiser ces coûts.

Pour toutes ces raisons, la méthode de placement adéquate doit allouer physiquement un intervalle de processeurs à chaque tâche du graphe de dépendance de la requête, en prenant en compte toutes les contraintes d'allocation non relâchables (i.e. C0, C1 et C2) et en relâchant la contrainte C3 si le coût de communication entre deux tâches qui s'exécutent en pipeline excède le coût de leur traitement. En même temps, elle doit essayer de réduire à chaque fois que cela est possible, le coût des communications inter-processeur, traduit par la formule suivante [AND 88] :

$$f = \sum_i \sum_k \sum_l \sum_j (C(i,j) \times d(k,l)) \times X(i,k) \times X(j,l)$$

avec

C(i, j) : quantités de données unitaires transférées de la tâche i à la tâche j,

d(k, l) : coûts de communication en terme de distance du processeur k au processeur l,

X(q, r) : placement de la tâche q sur le processeur r si X(q, r) = 1.

Génération de programme parallèle

Le processus de génération d'un programme parallèle est constitué de deux étapes :

E3 : intégration du constructeur Pipe : cette étape consiste à transformer la liste Liste_E en une liste Liste _E_P en intégrant le constructeur Pipe,

E4 : intégration du constructeur Par : cette étape consiste à réécrire la liste Liste _E_P en intégrant le constructeur Par.

Le processus de génération d'un programme parallèle doit vérifier les 2 contraintes suivantes :

1- respecter le temps de déclenchement des opérations déterminé à l'étape E2,

2- séquentialiser les opérations {$build_i$; $probeX_i$}.

Les mécanismes de contrôle associés aux constructeurs Seq, Par et Pipe ne sont pas suffisants pour garantir une exécution conforme au résultat de la stratégie PSA. Ceci provient de la dissociation logique de l'opération {$build_i$; $probeX_i$}. Du point de vue de la stratégie PSA, ces deux opérations sont considérées comme deux opérations indépendantes, alors qu'elles doivent s'exécuter de manière séquentielle. Pour cela, une approche a été proposée, (figure IV.20) consistant à respecter le déclenchement des opérations par l'exploitation des mécanismes de contrôle Seq et Par, et à assurer la séquentialité des opérations {$build_i$; $probeX_i$} à l'aide des messages asynchrones de contrôle (Envoyer_mi et Recevoir_mi).

```
Seq
    Par
        Seq
            Pipe scan5, build3 End_Pipe;
            Pipe scan6, probeJ3, build4 End_Pipe; Envoyer_m1;
            Recevoir_m3; Pipe scan1, probeD1, build5 End_Pipe;
            probeFE5
        End_Seq
        Seq
            Pipe scan4, build2 End_Pipe; Envoyer_m2;
        End_Seq
        Seq
            Pipe scan2, build1 End_Pipe; Envoyer_m3;
            Recevoir_m2; Pipe scan3, probeU2 End_Pipe;
            Recevoir_m1; probeU4
        End_Seq
    End_Par
End_Seq
```

Figure IV.20 : Programme parallèle associé au graphe de dépendance de la figure IV.18

Pour simplifier le processus de génération de code, les messages de contrôle sont intégrés directement dans les opérateurs $build_i$ et $probeX_i$ (figure IV.21).

```
Procédure Build_ni
Début
    pi := Recevoir_données (Ri);
    tant que non terminaison faire
        build (Bi, pi);
        pi := Recevoir_données (Ri);
    fin tant que
    Envoyer_Ctrl_Build;
Fin
    pi : ensemble de pages
```

```
Procédure ProbeX_ni
/* X : opération relationnel */
Début
    Recevoir_Ctrl_Build;
    pj := Recevoir_données (Sj);
    tant que non terminaison faire
        pk := probeX (Bi, pj);
        Envoyer_données (pk);
        pj := Recevoir_données (Sj);
    fin tant que
Fin
```

Figure IV.21 : Intégration du contrôle d'une opération $\{build_i ; probeX_i\}$.

5 Optimisation des communications de données et de contrôle

5. 1 Position du problème

La prise en compte du parallélisme inter-opération (parallélisme pipeline ou parallélisme indépendant) pose, en plus du problème d'équilibrage de charge dû à une mauvaise répartition de données inhérente à leur nature [SCH 89, WOL 91, LU 92], le problème de *communication des données et de contrôle* d'une exécution parallèle [GRA 88, PIR 90, ENG 95, HAM 93b, HAS 95]. Pour illustrer cette problématique, considérons le cas d'une exécution parallèle d'une requête SQL dont le graphe des opérations est montré dans la figure IV.22. Les opérations de Jointure J et d'Union U s'exécutent respectivement sur 3 et 2 processeurs disjoints. Les relations R1, R2 et R3 sont distribuées horizontalement respectivement sur 3, 2 et 2 disques.

L'exécution des opérations U et J nécessite la mise en place d'un opérateur pour redistribuer les données et d'un opérateur pour recevoir les données [DEW 92]. L'opérateur recevoir " Merge " sert à fusionner plusieurs entrées de flot de donnés en un seul flot. L'opération répartir " Split " applique sur chaque tuple produit par un opérateur relationnel la fonction de répartition et envoie chacun d'eux au processeur correspondant à la valeur retournée par la fonction de répartition.

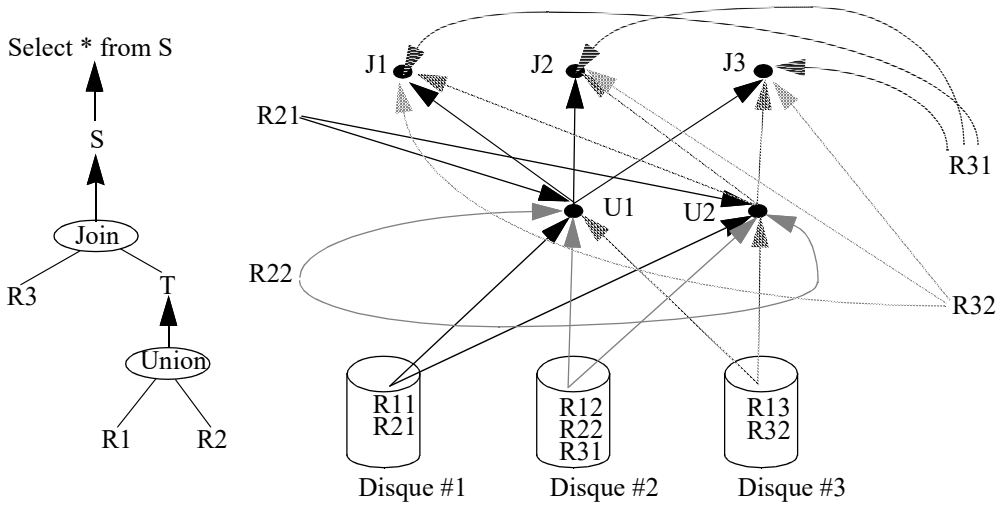

Figure IV.22 : Graphe de flot de données associé à une requête SQL

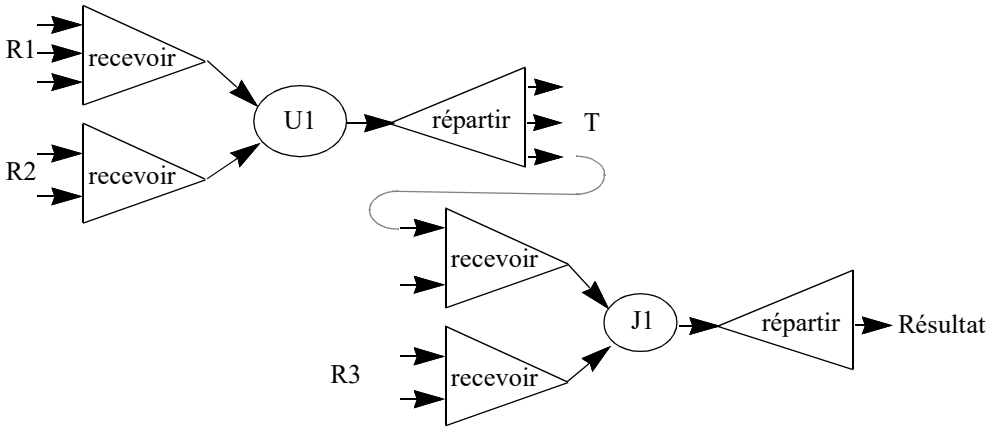

Figure IV.23 : Fusion des entrées et répartition de la sortie des opérateurs U1 et J1 [DEW 92]

Après chaque sous-opération de U (figure IV.23), le système exécute l'opération répartir dont le nombre de sorties est égal au nombre de processeurs alloués à l'opération successeur direct J. De plus, avant chaque sous-opération de J, le système exécute deux opérations recevoir afin de préparer les opérandes des sous-opérations de J. Chaque opération recevoir fusionne les flots de données provenant des sous-opérations de U et des sous-relations R31 et R32 pour préparer les opérandes d'une opération de jointure J. Quant aux messages de contrôle, sans tenir compte des conflits d'accès, nous devons intégrer des primitives de contrôle locales au niveau de chaque sous-opération

et des primitives de contrôle globales au niveau de la sous-opération élue coordinatrice [BOS 91].

Intuitivement, une relation résultat d'une opération doit être répartie afin de traiter l'opération successeur si l'une des conditions suivantes est vérifiée :

1- *fonction de répartition* : une relation résultat produite par une opération (par exemple la relation T = Union (R1, R2)) n'est pas consommée avec la même fonction de répartition par l'opération successeur (jointure (R3, T)),

2- *attribut de répartition* : une relation produite (par exemple T = Union (R1, R2)) est consommée par l'opération successeur (jointure (R3, T)) avec un attribut de répartition différent,

3- *nombre de processeurs* : le nombre de processeurs alloués à une opération est différent du nombre de processeurs alloués à l'opération successeur.

En conséquence, l'exécution de la séquence {U ; J} nécessite un nombre important de messages de communication de données et de contrôle. De manière générale, si les opérations U et J sont exécutées respectivement sur N et M processeurs, alors le nombre de liens de communication de données entre U et J est exactement = N.M, ce qui est prohibitif si N ou M est grand. De plus, il est évident que la réduction des communications de données entre les noeuds d'un système parallèle se traduit aussi par une réduction des messages de contrôle puisqu'on évite d'attendre la fin locale d'une sous-opération, de synchroniser deux opérations et de déclencher explicitement une opération (déclenchement par les données).

Dans la suite, nous décrivons une méthode d'optimisation de transfert de données et de contrôle qui s'intègre dans le processus de conception d'un optimiseur de requêtes SQL dans un environnement parallèle. Cette optimisation consiste donc à éviter de transmettre des messages de données et de contrôle qui ne sont pas nécessaires à une exécution parallèle conforme au plan d'exécution engendré par l'optimiseur. Le principe de la méthode est basée sur la propagation des attributs de répartition et du nombre de processeurs durant la phase de parallélisation intra-opération. Le principal problème qui se pose est le choix des attributs de répartition des opérations d'une requête qui dépend fortement des prédicats de jointure et des algorithmes de jointure. Récemment, dans le cadre du projet Papyrus [GAN 92], une méthode d'optimisation des coûts de communication a été développée [HAS 95]. Le principe de la méthode est fondée sur le concept classique de *coloriage* d'un arbre [DAH 92]. Les attributs de répartition sont considérés comme des couleurs. Le problème revient donc à colorer le maximum d'opérations d'un plan d'exé-

cution d'une requête avec la même couleur. Ceci permet d'éviter de redistribuer les données lorsque deux opérations adjacentes ont la même couleur.

5. 2 Principe de la méthode de propagation

Pour réduire les communications de données et de contrôle, l'idée de base est de propager le nombre de processeurs et l'attribut de répartition d'une relation, en utilisant *la même fonction de répartition* pour toutes les relations et en n'acceptant pas la duplication des relations opérandes. Dans cette perspective, l'étude des opérateurs relationnels par rapport au problème de choix des attributs de répartition a permis de distinguer deux cas :

1- pour l'opérateur de jointure, les attributs sur lesquels peuvent être réparties les relations opérandes, sont les attributs de jointure,

2- quant aux autres opérateurs comme l'union et la différence, il n'y a aucune contrainte sur le choix d'un attribut de répartition.

Nous exploitons ces deux propriétés pour propager le nombre de processeurs d'une opération et l'attribut de répartition de ses *relations opérandes* en utilisant la même fonction de répartition pour toutes les relations. Dans la figure IV.22, la relation T produite par l'union (R1, R2) n'aura pas besoin d'être répartie pour être consommée par la jointure (R3, T) si les deux conditions suivantes sont vérifiées :

1- l'union (R1, R2) a le même nombre de processeurs que la jointure (R3, T),

2- l'attribut de répartition des relations opérandes R1 et R2 correspond à l'attribut de répartition de la relation opérande T.

Le processus de propagation consiste, pour chaque opération du graphe de la requête :

1- à calculer le couple (nombre de processeurs, temps de réponse local) associé à l'algorithme le plus approprié à l'opération (cf. 4. 1 page 216),

2- à déterminer si les conditions requises pour la propagation sont vérifiées :

- soit l'opération considérée est une jointure et l'attribut de répartition de la relation opérande de l'opération successeur est un attribut de jointure soit l'opération considérée n'est pas une jointure (e.g. différence, union) et
- le temps de réponse est amélioré par le processus de propagation (cf. 6. 2 page 196).

6 Evaluation des performances et comparaison

6. 1 Comparaison de l'efficacité des différentes formes de parallélisme

Plusieurs travaux ont comparé l'efficacité de chaque forme de parallélisme [DEE 90, TAN 92, HAM 94a]. Deen et al. [DEE 90] ont évalué les performances de requêtes contenant 2 ou 3 jointures en utilisant un petit nombre de processeurs (11 processeurs) et en considérant que toutes les relations tiennent en mémoire. Le résultat de cette étude montre qu'en utilisant uniquement le parallélisme intra-opération, on obtient de meilleures performances qu'en exploitant le parallélisme inter-opération. En d'autres termes, il est plus efficace d'utiliser tous les processeurs pour calculer chaque jointure, l'une après l'autre, que de les calculer en parallèle, avec moins de processeurs.

Tan et Lu [TAN 92] obtiennent un résultat différent de Deen et al. lorsque [DEE 90] le nombre de processeurs est important. Ils ont montré que l'exécution de plusieurs jointures en parallèle peut être avantageuse quand le nombre de processeurs est important. Ceci provient du gain en temps de réponse de l'opération de jointure, qui est très important quand le nombre de processeurs est faible. Par contre, lorsque le nombre de processeurs excède une certaine valeur, ce gain est très faible. Ainsi, pour un nombre de processeurs important, il devient bénéfique d'exécuter plusieurs opérations de jointure en parallèle au lieu de les calculer l'une après l'autre.

Dans la suite, nous illustrons les résultats apportés par chacun en évaluant le temps de réponse du plan d'exécution représenté par le graphe de dépendance des opérations (figure IV.24) dans les cas suivants :

1- *Partitionnement total* : une opération s'exécute donc sur l'ensemble des processeurs de la machine. Le plan d'exécution de type P1 utilise uniquement le parallélisme intra-opération ;

2- *Partitionnement partiel* : une opération s'exécute sur une partie des processeurs.

- le plan d'exécution de type P2 utilise le parallélisme intra-opération et le parallélisme indépendant inter-opération,
- le plan d'exécution de type P3 contient les différents types de parallélisme tels que le parallélisme intra-opération, le parallélisme indépendant inter-opération et le pipeline.

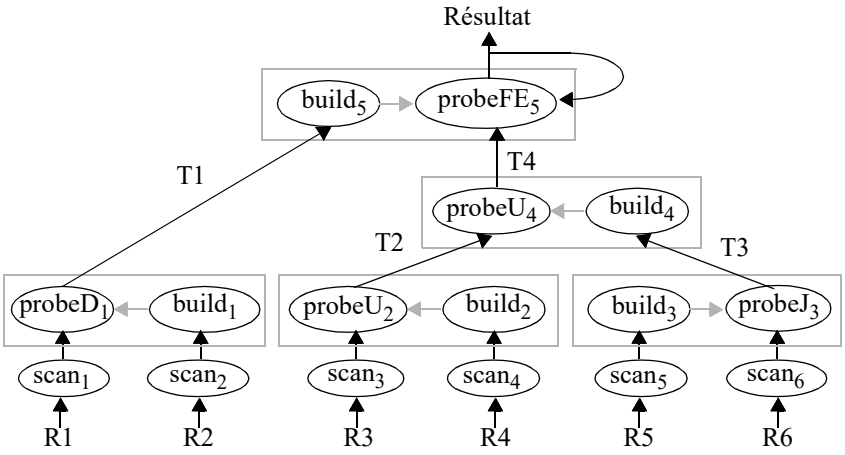

Figure IV.24 : Graphe de dépendance des opérations

Nous décrivons d'abord les hypothèses qui constituent la base du modèle analytique introduit par [VAL 88b] :

H1 : le nombre de tuples produits par processeur est identique,

H2 : le temps de répartition d'un tuple est une constante,

H3 : il n'y a pas de conflit d'accès sur le réseau.

Les paramètres du modèle analytique utilisés [VAL 84, VAL 88b, CHE 89] sont :

|R| : nombre de tuples de la relation R,
||R|| : nombre de pages de la relation R,
t_r : nombre de tuples par page de la relation R,
CR : temps de lecture d'une page,
CW : temps d'écriture d'une page,
I : temps pour déplacer un tuple,
O : temps pour effectuer une opération simple (e.g. temps de comparaison de deux attributs),
CJO : temps pour joindre deux pages non triées = $t_r.t_s.O$,
th : temps pour répartir un tuple,
trf : temps pour transférer un tuple,
msg : temps pour envoyer un message,
d : nombre de processeurs alloués à une opération,
p : nombre de processeurs de l'opération successeur dans le graphe de dépendance,

q : nombre de paquets de hachage,
h : nombre d'itérations de la boucle de jointure de l'opérateur de fermeture externe (hauteur du graphe associé à la relation T1),
$\lceil x \rceil$: représente la partie entière supérieure de x.

Dans la suite, T_{ef}, T_d et T_{com} désignent respectivement le temps effectif de traitement d'une opération, le temps de répartition et le temps de communication entre deux ensembles de processeurs (i.e. temps de transfert des tuples + temps de communication des messages).

Le temps de réponse local d'une opération de jointure $T \leftarrow R \bowtie S$ par hachage simple est $= T_{ef} + T_d + T_{com}$ avec

T_{ef} = (|R|/d).th + ((||R||/d) + (||S||/d)).CR + (||R||/d/q).(||S||/d).CJO + |T|.I + ||T||/d.CW
= temps pour construire la table de hachage + temps de lecture + temps de comparaison + temps de déplacement + temps d'écriture

T_d = (|T|/d).th et T_{com} = ((|T|/d).trf + p.msg). $\lceil d/p \rceil$

Pour calculer les temps de réponse locaux de la différence (R - S) et de l'union ($R \cup S$), nous pouvons nous appuyer sur les propriétés décrites dans [BOU 90]. En effet, (R - S) revient à exécuter la semi-jointure de R et S avec l'égalité de tous les attributs des deux relations. Lorsque l'algorithme de jointure détermine un couple de tuples se joignant ensemble, le tuple correspondant de R est éliminé. En conséquence, la fonction de temps de réponse local de la différence (R - S) est de la même forme que la jointure mis à part que : (i) le temps de comparaison est plus important puisque tous les attributs de R sont comparés à tous les attributs de S, (ii) le temps de déplacement des tuples résultats est remplacé par le temps pour marquer les tuples de R qui sont éliminés. Pour ce qui est de l'union, $(R \cup S) = (R - S) + S$ où " + " dénote l'opération de concaténation de deux relations. Dans un contexte monoprocesseur, l'opération de concaténation se matérialise soit par la recopie des pages de S à la suite des pages de (R - S) soit par le chaînage des pages des deux relations si les deux opérandes de la concaténation ne sont pas utilisées ultérieurement dans le plan d'exécution [BOU 90]. En ce qui concerne la version parallèle pour une architecture multiprocesseur à mémoire distribuée, deux cas peuvent être considérés :

1- si la relation résultat Ti = (Ri - Si) + Si calculée par le processeur Pi doit être transmise à d'autres processeurs, alors la concaténation est réalisée par les processus de répartition et de communication des données,

2- sinon on se ramène au cas monoprocesseur.

Le temps de réponse local de la fermeture externe (figure IV.24) est :

T_{ef} = ||T1||/d.CR + (|T1|/d).th + ||T4||/d.CR + (||Résultat||/d.||T1||/d).CJO + |Résultat|/d.I + h.(d-1).(|Résultat|.trf/(h.d^2) + msg) + ||Résultat||/d.CW ;
= temps pour hacher T1 + temps de lecture de T4 + temps de comparaison + temps de déplacement + temps de communication intra-opération + temps d'écriture ;

T_d = (|Résultat|/d).th et T_{com} = ((|Résultat|/d).trf + p.msg). $\lceil d/p \rceil$.

Pour mesurer l'apport des différents formes de parallélisme, nous nous sommes appuyés sur le modèle de référence (i.e Benchmark) décrit dans [BIT 83, GRA 93, SCH 90]. La base de données est composée de six relations de 10^6 tuples. Chaque relation est caractérisée par 16 attributs (13 entiers de 4 octets et 3 chaînes de caractères de 52 octets). Les valeurs des paramètres de simulation utilisés [VAL 88b, SCH 90] sont :

- le nombre d'instructions par seconde d'un processeur = 4,000,000,
- le temps de lecture d'une page disque de 18 K ≡ 32 800 instructions,
- le temps d'écriture d'une page disque de 18 K ≡ 61500 instructions,
- le temps pour transférer un tuple = 5 µs,
- le temps pour envoyer un message = 1 ms.

La figure IV.25 représente le temps de réponse des différents plans d'exécution en fonction du nombre de processeurs. A partir de ces courbes, quatre observations peuvent être faites :

1- le plan d'exécution de la forme P1 qui exploite uniquement le parallélisme intra-opération est le plus efficace pour un petit nombre de processeurs, de 32 à 130 processeurs sur l'exemple,

2- le temps de réponse du plan d'exécution de la forme P1 se dégrade à partir de 256 processeurs. Ceci provient du temps de communication des messages de données, lequel augmente le temps de réponse,

3- le plan d'exécution de la forme P2 qui exploite le parallélisme intra-opération et le parallélisme indépendant inter-opération a le meilleur temps de réponse entre 130 et 512 processeurs,

4- le plan d'exécution de la forme P3 qui exploite toutes les formes de parallélisme a les meilleures performances pour un nombre important de processeurs, plus de 512 dans l'exemple.

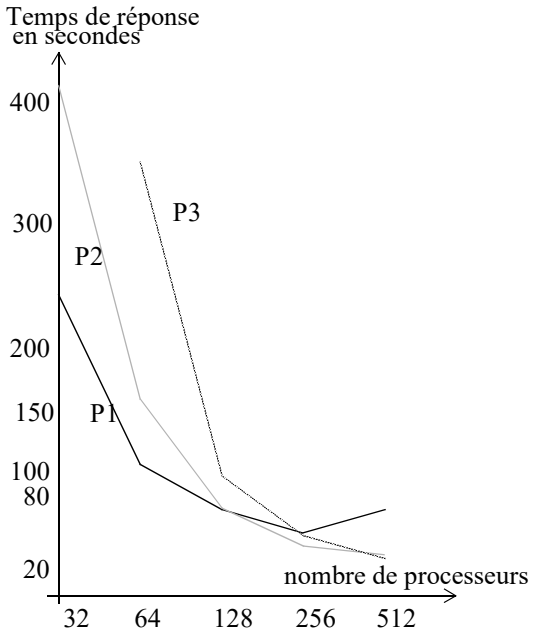

Figure IV.25 : Temps de réponse des plans d'exécution de type P1, P2 et P3

En conclusion, les résultats obtenus montrent que le bénéfice du pipeline est très limité. En effet, les chaînes de pipeline sont rarement très longues (la longueur maximum dans l'exemple est de 3), soit à cause de la limitation de ressources soit parce-qu'elles doivent attendre d'avoir consommé entièrement leur opérande pour produire un résultat (comme par exemple les fonctions d'agrégats). De plus, si dans une chaîne de pipeline, le temps de réponse (local) d'une opération en début de chaîne est beaucoup plus grand que les autres, ces dernières immobilisent des ressources plus longtemps. Ainsi, le rendement du système et le facteur d'accélération sont limités. Par exemple, dans la chaîne de pipeline *Pipe scan1, probeD1, build5 End_Pipe*, les processeurs alloués à l'opérateur $build_5$ restent inactifs approximativement 80% de leur temps, en attente des tuples produits par l'opérateur $probeD_1$.

6. 2 Impact du processus de propagation sur la réduction des communications

Dans cette partie, nous comparons les temps de réponse locaux d'une opération avec et sans propagation de l'attribut de répartition et du nombre de processeurs, pour calculer une relation déduite définie par une union/ différence/ sélection. Cette opération appartient à une requête constituée d'au

moins deux opérations, et toutes les relations sont matérialisées (i.e. écrites sur disques). Nous ajoutons au modèle analytique l'hypothèse H4 [VAL 88b] : " le temps pour produire un tuple est une constante t ", incluant le temps de traitement et le temps d'entrées/sorties. Par ailleurs, on retrouve des résultats très proches dans le cas où les accès disques et le temps de traitement sont détaillés.

Dans l'exemple donné dans la figure IV.26, nous supposons que les opérations de jointure et d'union s'effectuent respectivement sur d1 processeurs et sur d2 processeurs.

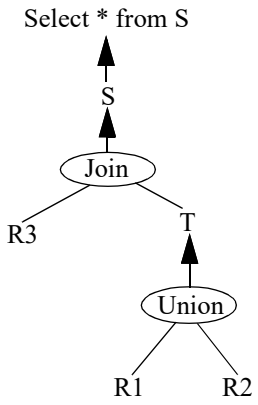

Figure IV.26 : Exemple de requête simple

Le temps de réponse local d'une opération sans propagation, noté $TRL1 = T_{ef} + T_d + T_{com} = (|T|.t/d2)+(|T|.th/d2)+\lceil d2/d1 \rceil ((|T|.trf/d2)+(d1.msg))$.

Le temps de réponse local d'une opération avec propagation, noté $TRL2 = T_{ef} + T_{com} = (|T|.t/d1) + ((|T|.trf/d1) + msg)$.

Pour évaluer le bénéfice de l'algorithme de propagation, nous définissons le facteur d'accélération comme étant le rapport $\Omega = TRL1 / TRL2$.

Les valeurs suivantes ont été choisies pour tracer les courbes graphiques de Ω en fonction de $d = d1 - x$ avec $x \in [1, d1 - 1]$ dans le cas où $d1 > d2$ et $d = x - d1$ avec $x \in [d1+1, +\infty[$ dans le cas où $d1 < d2$ (figure IV.27). $|DT| = 10^6$ tuples ; msg = 1 ms ; trf = 5 µs ; t = 0,2 ms ; th = 3 µs.

Dans ces courbes nous considérons toujours que les deux opérations du graphe s'exécutent sur des ensembles de processeurs disjoints (i.e. dans le cas de la propagation le temps de communication n'est jamais égale à zéro).

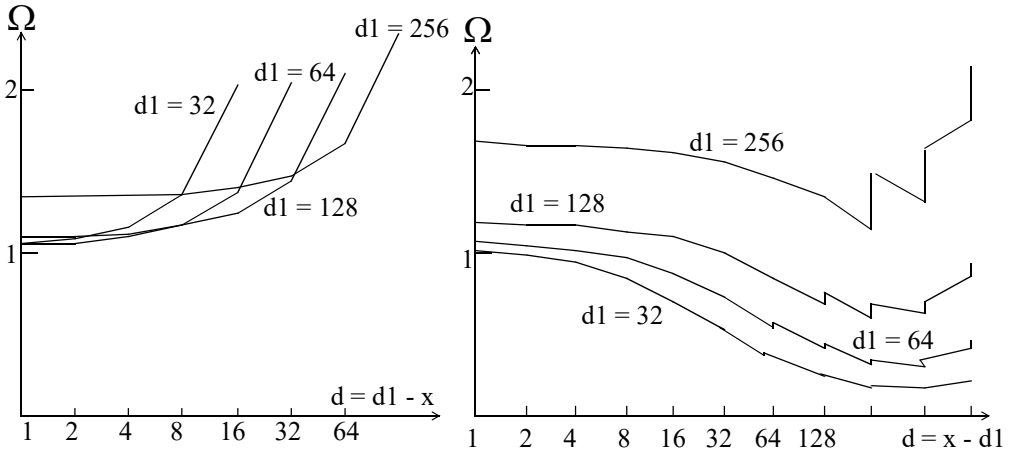

Figure IV.27 : Facteur d'accélération de la propagation en fonction de d1 et d2

La figure IV.27 gauche représentant les valeurs du facteur d'accélération pour d1 > d2 montre que le bénéfice de la propagation est proportionnel à la différence entre les nombres de processeurs d1 et d2. En effet, dans la figure IV.27 gauche, à une faible différence de processeurs $\Delta d = d1 - d2$ correspond à un bénéfice approximatif de 15-20% jusqu'à $\Delta d <= 8$ processeurs. A un Δd assez grand correspond un bénéfice approximatif de 50-60% pour $8 < \Delta d <= 16$, et approximativement de 70-100% pour $32 <= \Delta d <= 64$. Ceci est dû à la réduction des temps de communication et de répartition (= 0 si les deux opérations utilisent la même fonction de répartition, le même attribut de répartition et le même nombre de processeurs).

Dans le cas où d1<d2 (figure IV.27 droite), nous distinguons 3 cas :

1- pour $1 <= \Delta d <= 16$ (avec d1 = 128), le temps de communication et de répartition gagné par la propagation est supérieur au temps effectif perdu à cause de la réduction du nombre de processeurs,

2- pour $16 < \Delta d < 512$ (avec d1 = 128), la perte du temps effectif est plus importante que le temps gagné par la propagation. Dans ce cas, on ne doit pas exécuter le processus propagation, comme illustré dans la figure IV.28 ;

3- lorsque d1 est assez grand > 200 processeurs, le temps effectif perdu en raison de la propagation est toujours largement compensé par la réduction du temps de communication et de répartition.

La figure IV.28 résume les différents cas où la propagation est efficace ($\Omega > 1$). De plus, si nous tenons compte des messages de terminaison lors du calcul des temps de réponse locaux, alors les intervalles de propagation aug-

mentent. Par exemple, si d1 = 128, alors la propagation pour d1 < d2 devient efficace jusqu'à d2 = 260 processeurs.

d2 \ d1	32	64	128	>= 256
0 <= d2 <= 34	> 1	> 1	> 1	> 1
34 < d2 <= 70	< 1	> 1	> 1	> 1
70 < d2 <= 160	< 1	< 1	> 1	> 1
d2 >= 160	< 1	< 1	< 1	> 1

Figure IV.28 : Intervalles d'efficacité du processus de propagation

7 Conclusion

Nous avons présenté deux approches de parallélisation des opérations d'une requête SQL : l'approche mono-phase et l'approche à deux phases. La présentation des stratégies de parallélisation fondées sur l'approche mono-phase nous a permis de montrer l'impact de la nature de l'espace de recherche sur la stratégie d'exécution. La stratégie associée à un arbre linéaire droit, basée sur l'utilisation de l'algorithme de jointure par hachage, est la plus adaptée pour exploiter au mieux le parallélisme des SGBD parallèles comportant un grand nombre de processeurs [SCH 90]. Mais, cette structure n'est plus la meilleure quand la mémoire est limitée. En effet, dans ce cas, si l'on élargit l'espace de recherche aux arbres droits segmentés [CHE 92b], le nombre de phases du plan d'exécution peut être diminué. De plus, il est possible, avec les arbres zig-zag [ZIA 93], de réduire le nombre d'entrées/sorties en évitant de stocker sur disque les relations temporaires de chaque sous-arbre. Le choix de l'algorithme d'une jointure influe également sur le choix d'une structure arborescente. Par exemple, si l'on choisit l'algorithme de jointure par tri-fusion, la meilleure structure est l'arbre bushy. En effet, les arbres bushy offrent plus d'opportunités de parallélisme puisque des fusions peuvent s'exécuter en parallèle, alors que dans les arbres linéaires gauches et les arbres linéaires droits, toutes les fusions s'exécutent de manière séquentielle.

Quant aux stratégies de parallélisation basées sur l'approche à deux phases, deux méthodes ont été décrites :

1- la méthode de parallélisation intra-opération et inter-opération des requêtes, implantée dans le système XPRS. Celle-ci est fondée sur l'exploitation maximale des ressources (i.e. bande passante en entrées/sorties, et capacité des processeurs) en exécutant en parallèle deux tâches à leur point

d'équilibre I/O-CPU. Pour rentabiliser les ressources du système parallèle, l'auteur propose une méthode d'ajustement dynamique du degré de parallélisme des tâches ;

2- la méthode d'ordonnancement parallèle PSA d'un plan d'exécution contenant des opérateurs relationnels et opérateurs de fermeture externe ou de fermeture transitive. Le principe de PSA s'appuie sur les méthodes sérielles (i.e. recherche de chemin critique) dont la règle de priorité appliquée est celle des dates de fin au plus tard croissantes, et tient compte du nombre de processeurs.

La mise en oeuvre des traitements parallèles d'une requête SQL nécessite l'initialisation de plusieurs processus sur différents processeurs et requiert des communications de données entre ces processus. Le principal problème qui se pose au niveau des modèles d'exécution parallèle est de trouver le point de compromis traitement-communication permettant d'exploiter au maximum les capacités du système parallèle. Dans cette perspective, plusieurs travaux ont été menés pour réduire les coûts de communication inter-processeur, en particulier pour éviter la redistribution des données et minimiser le transfert de messages [GRA 88, PIR 90, ENG 95, HAM 93b, HAS 95]. Une étude expérimentale effectue avec le SGBD parallèle commercial NonStop SQL/MP de Tandem Computers [ENG 95] a permis de montrer que le coût de la redistribution peut excéder largement le coût de traitement d'une opération relationnelle. Ceci confirme la nécessité de développer des modèles et des algorithmes efficaces tenant compte des coûts de communications dus au parallélisme.

8 Références bibliographiques

[AND 88] F. André, J. L. Pazat, " Le placement sur des architectures parallèles ", Technique et Science Informatiques TSI, Vol. 7, No. 4, 1988, pp. 385-401.

[BEE 87] C. Beeri, R. Ramakrishnan, " On the Power of Magic ", Proc. ACM SIGMOD Symposium on Principles of Database Systems, 1987, pp.269-283.

[BIT 83] D. Bitton et al., " Benchmarking Database Systems - a Systematic Approach ", Proc. of the 1983 VLDB Conf., Octobre 1983, pp. 8-19.

[BOU 90] M. Bouzeghoub et al., " Systèmes de bases de données : des techniques d'implantation à la conception de schémas ", Ed. Eyrolles, 1990.

[BOS 91] P. Borla-Salamet et al., " Compiling Control into Database Queries for Parallel Execution Management ", First Intl. Conf. on Parallel Distributed Information Systems, December 1991, Florida, pp. 271-279.

[CHE 89] J.P. Cheiney, C. De Maindreville, " A Parallel Transitive Closure Algorithm Using Hash-Based Clustering ", IWDM'89, Sixth Intl. Workshop on Database Machines Deauville, France, june 1989.

[CHE 92a] M.S. Chen et al., " Scheduling and Processor Allocation for Parallel Execution of Multi-join Queries ", Proc. 8th Intl. Conf. Data Eng., IEEE CS Press, Los Alamitos, Californie, 1992, pp. 58-67.

[CHE 92b] M.S. Chen et al., " Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins ", Proc. of the 18th VLDB Conf., Vancouver, Aug. 1992, pp. 15-26.

[CHE 95] C. Chekuri et al., "Scheduling Problem in Parallel Query Optimization", Symposium in Principles of Database Systems, PODS'95, 1995.

[CHR 92] P. Chretienne, " Ordonnancement et parallélisme ", 20th spring school of LITP, Sables-d'Or-les Pins, FRANCE, may 1992, pp. 297-312.

[CON 67] R.W. Conway et al., " The Theory of Scheduling ", Addition-Wesley, 1967.

[COS 93] M. Cosnard, D. Trystram, " Algorithmes et architectures parallèles ", InterEditions, 1993.

[DAH 92] E. Dahlhaus et al., " Complexity of Multiway Cuts ", 24th Annual ACM Symposium on the Theory of Computing, 1992, pp. 241-251.

[DEE 90] S. M. Deen et al., " Multi-Join on Parallel Processors ", Proc. Second Intl. Symp. Databases in Parallel and Distributed Systems, IEEE CS Press, Los Alamitos, Calif. 1990, pp. 92-102.

[DEW 84] D. Dewitt et al., " Implementation Techniques for Main Memory Database Systems ", Proc. of the 1984 SIGMOD Conf., Boston, Mass., June, 1984, pp. 1-8.

[DEW 92] D.J. Dewitt, J. Gray, " Parallel Database Systems: The Future of High Performance Database Systems ", Com. of the ACM, Vol. 35, No. 6, June 1992, pp. 85-98.

[ENG 95] S. Englert et al., " Parallelism and its Price: A Case Study of NonStop SQL/MP ", SIGMOD Record, Vol. 24, No. 4, December 1995, pp. 61-71.

[GAN 92] S. Ganguly et al., " Query Optimization for Parallel Execution ", Proc. ACM SIGMOD Conf. on Management of Data, New York, June 1992, pp. 9-18.

[GRA 88] J. Gray, " The Cost of Messages ", Proc. of the 7th ACM Symposium on Principles of Distributed Computing, Toronto, Canada, August, 1988, pp. 1-7.

[GRA 93] J. Gray, " The Benchmark Handbook for Database and Transaction Processing Systems ", 2nd Edition, Morgan Kaufmann Publishers, Carlifornia, 1993.

[HAM 93a] A. Hameurlain, F. Morvan, " A Parallel Scheduling Method for Efficient Query Processing ", 22nd Intl. Conf. on Parallel Processing, St. Charles, Chicago IL. August 16-20, 1993, Vol. 3, pp. 258-261.

[HAM 93b] A. Hameurlain, F. Morvan, " An Optimization Method of Data Communication and Control for Parallel Execution of SQL Queries ", Intl. Conf. on Database and Expert Systems Applications, LNCS 720, Prague, Sept. 6-9, 1993, pp. 301-312.

[HAM 94a] A. Hameurlain, F. Morvan, " Exploiting Inter-Operation Parallelism for SQL Query Optimization ", Intl. Conf. on Database and Expert Systems Applications DEXA'94, LNCS 856, Greece, 7-9 Sept., 1994, pp. 759-768.

[HAM 94b] A. Hameurlain, F. Morvan, " Scheduling and Parallelism for Extended SQL Query Optimization ", 7th Intl. Conf. on Parallel and Distributed Computing Systems PDCS'94, Las Vegas (Nevada), 5-8 octobre, 1994, pp. 466-471.

[HAM 95b] A. Hameurlain, F. Morvan, " Scheduling and Mapping for Parallel Execution of Extended SQL Queries ", 4th Intl. Conf. on Information and Knowledge Management, ACM Press, Baltimore, Maryland, 28 Nov. - 2 Dec. 1995, pp. 197-204.

[HAS 94] W. Hasan, R. Motwani, "Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism", Proc. of the 20th Intl. Conf. on VLDB, Santiago, Sept. 1994.

[HAS 95] W. Hasan, R. Motwani, " Coloring Away Communication in Parallel Query Optimization ", Proc. of the 21th Intl. Conf. on VLDB, Zurich, Sept. 1995.

[HON 91] W. Hong, " Optimization of Parallel Query Execution Plans in XPRS ", First Intl. Conf. on Parallel Distributed Information Systems, Dec. 1991, Florida, pp. 218-225.

[HON 92] W. Hong, " Exploiting Inter-Operation Parallelism in XPRS ", Proc. ACM SIGMOD Conf. on Management of Data, New York, June 1992, pp. 19-28.

[IOA 90] Y.E. Ioannidis, Y. C. Kong, " Randomized Algorithms for Optimizing Large Join Queries ", Proc. ACM SIGMOD Conf. on Management of Data, Atlantic City, May 1990, pp. 312-321.

[IOA 91] Y.E. Ioannidis, Y. Cha Kong, " Optimizing Linear Trees vs. Optimizing Linear and Bushy ", Proc. ACM SIGMOD Conf. on Management of Data, USA, 1991, pp. 168-177.

[LAN 91] R.S.G. Lanzelotte, P. Valduriez, " Extending the Search Strategy in a Query Optimizer ", Proc. of the 17th Intl. Conf. on VLDB, Barcelona,1991, pp. 363-373.

[LAN 92] R.S.G. Lanzelotte et al., " Measuring the Effectiveness of Optimization Search Strategies ", VIII ième Journée BD Avancées, Trégastel, Sept. 1992, pp. 42-61.

[LAN 93] R.S.G. Lanzelotte et al., " On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces ", Proc. of the 17th Intl. Conf. on VLDB, Dublin, 1993, pp. 413-504.

[LU 92] H. Lu, K. Tan, " Dynamic and Load-Balanced Task-Oriented Database Query Processing in Parallel Systems ", Advances in Database Technology, EDBT'92 3rd Intl. Conf. on Extending Database Technology, Vienna, March 1992, pp. 357-372.

[LU 94] H. Lu et al., " Query Processing in Parallel Relational Database Systems ", IEEE Computer Society Press, ISBN 0-8186-5452-X, Los Alamitos, CA, 1994.

[MAU 93] Ph. Mauran et al., " Structures de programmation parallèle : conception, réalisation et utilisation ", Ed. Cépaduès, 1993.

[MEH 93] M. Mehta et al., " Batch Scheduling in Parallel Database Systems ", 9 th Conf. on Data Engineering, Vienna, April, 1993, pp. 400-410.

[ONO 90] K. Ono, G. Lohman, " Measuring the Complexity of Join Enumeration in Query Optimization ", Proc. of the 16th VLDB Conf., Brisbane, 1990, pp. 314-325.

[PIR 90] H. Pirahesh et al., " Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches ", Proc. Second Intl. Symposium on Databases in Parallel and Distributed Systems, Dublin, Ireland, 1990, pp. 4-29.

[SCH 89] D. Schneider, D. Dewitt, " A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment ", Proc. ACM SIGMOD Conf. on Management of Data, Portland, June 1989, pp. 110-121.

[SCH 90] D. Schneider, D. Dewitt, " Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines ", Proc. of the 16th VLDB Conf., Brisbane, 1990, pp. 469-480.

[SEL 79] P. G. Selinger et al., " Access Path Selection in a Relational Database Management System ", Proc. ACM SIGMOD Conf. on Management of Data, Boston, May 1979, pp. 23-34.

[STO 88] M. Stonebraker et al., " The Design of XPRS ", Proc. of the14 th VLDB Conf., San Mateo, Calif., 1988, pp. 318-330.

[SWA 88] A. Swami, A. Gupta," Optimization of Large Join Queries ", Proc. ACM SIGMOD Conf. on Management of Data, Chicago, June 1988, pp. 8-17.

[SWA 89] A. Swami, " Optimization of Large Join Queries: Combining Heuristics an Combinatorial Techniques ", Proc. ACM SIGMOD Conf. on Management of Data, Portland, 1989, pp. 367-376.

[TAN 91] K.L. Tan, H. Lu, " A Note on the Strategy Space of Multiway Join Query Optimization Problem in Parallel Systems ", SIGMOD Record, Vol. 20, No. 4, December 1991, pp. 81-82.

[TAN 92] K. L. Tan, H. Lu, " Processing Multi-Join Query in Parallel Systems ", Symp. Applied Computing, IEEE CS Press, Los Alamitos, Calif., 1992.

[TAN 93] K.l. Tan, H. Lu, " Pipeline Processing of Multi-Way Join Queries in Shared-Memory Systems ", 22nd Intl. Conf. on Parallel Processing, St. Charles, Chicago IL. August 16-20, 1993, Vol. 1, pp. 345-348.

[VAL 84] P. Valduriez, G. Gardarin, " Join and Semi-join Algorithms for a Multiprocessor Database Machine ", ACM TODS, Vol. 9, No. 1, March 1984, pp. 133-161.

[VAL 88b] P. Valduriez, S. Khoshafian, " Parallel Evaluation of the Transitive Closure of a Database Relation ", Intl. Journal of Parallel Programming, Vol. 17, No.1, Feb. 1988, pp. 19-42.

[WOL 91] L. Wolf et al., " An effective Algorithm for Parallelization Hash Join in the Presence of Data Skew ", 7th Intl. Conf. on Data Eng., Japan, April 1991, pp. 200-209.

[ZIA 92] M. Ziane, " Optimisation de requêtes pour un système de gestion de bases de données parallèle ", Doctorat de l'Université de PARIS 6, Février 1992.

[ZIA 93] M. Ziane et al. " Parallel Query Processing in DBS3 ", 2nd Intl. Conf. on Parallel and Distribued Information Systems, San Diego, January 22-24, 1993.