RenPar'8,
Bordeaux, France, 20–24 mai 1996.

Régulation dynamique en Athapascan : exemple d'un tri parallèle probabiliste optimal

Mathias Doreille[a†], Gerson Cavalheiro[a†*], Jean-Louis Roch [a†]

[a]{Mathias.Doreille, Gerson.Cavalheiro, Jean-Louis.Roch}@imag.fr
LMC-IMAG, Institut Fourier, BP 53 - 100, rue des Mathématiques - 38041 GRENOBLE

Résumé

Nous présentons deux mécanismes de régulation dynamique de charge d'un ensemble de tâches indépendantes et de durées inconnues. Ces deux mécanismes sont appliqués et comparés théoriquement (modèle PRAM) et expérimentalement (machine IBM SP2 32 processeurs) à un algorithme de tri parallèle probabiliste optimal.

1. Introduction

De nombreux algorithmes mettent en jeux des tâches parallèles indépendantes, de durée inconnue avant leur terminaison et créées dynamiquement en cours d'exécution. L'implémentation efficace nécessite alors un mécanisme de régulation adaptée. Cet article présente la mise en œuvre théorique (PRAM) et expérimentale (mesures sur SP-2 32 processeurs) de deux techniques de régulation dynamique avec l'environnement ATHAPASCAN[3] et correspondant à deux cas spécifiques qui apparaissent dans différentes applications étudiées dans le cadre de STRATAGÈME[4] :

- lors de la création d'une tâche, une estimation de son coût est connue. C'est par exemple le cas de l'algorithme d'élimination de Gauss avec pivot total ou de différents problèmes en traitement d'images [4]. Un placement peut alors être réalisé à partir de cette estimation de façon à équilibrer la charge entre les processeurs.

- seul un encadrement de la durée d'une tâche est connu lors de sa création, le coût exact n'étant déterminé que lorsqu'elle se termine. L'ordonnancement des tâches se fait alors par l'utilisation des fins d'exécution des tâches comme indication de charge [1, 3].

Dans la suite, on considère l'implémentation distribuée d'un algorithme parallèle probabiliste optimal de tri de complexité temporelle $O(\log n)$ avec n processeurs d'une CREW-PRAM [5] (§2). Cet algorithme est intéressant de par sa faible complexité de communication, mais nécessite l'ordonnancement de tâches de durée inconnue. Le calcul d'un ordonnancement pour ce problème est étudié de manière théorique (§3) et pratique (§4) en comparant expérimentalement deux techniques de régulation (§4.1 et §4.2).

$*$. Boursier du gouvernement brésilien CAPES/COFECUB.

†. Ce travail de recherche s'est déroulé au sein du projet APACHE, financé par le CNRS, l'INRIA, l'INPG et UJF.

3. environnement de programmation parallèle développé au sein du projet APACHE.

4. Projet Inter-PRC, terminé en 95.

2. Adaptation distribuée d'un algorithme de quicksort généralisé

De par la faible localité du problème du tri ($O(\log n)$) [6], les communications induites par un algorithme parallèle distribué de tri sont inévitablement importantes. Nous avons choisi de mettre en œuvre un algorithme parallèle probabiliste (Las Vegas) dû à Reischuk [7], qui présente différents avantages :

– faible volume de communication de l'ordre de $O(n)$.

– complexité parallèle optimale : temps $O(\log n)$ sur n processeurs d'une CREW PRAM.

– la constante du travail ($O(n \log n)$) est proche du quicksort séquentiel.

Cet algorithme peut être vu comme une généralisation du quicksort [5]. Il consiste à prendre simultanément $k-1$ pivots ($k = \sqrt{n}$) et à les trier (par sélection). Soient $p_0 = -\infty, p_1, \ldots, p_{k-1}, p_k = +\infty$ les pivots triés : deux pivots consécutifs (p_i, p_{i+1}) définissent alors un seau B_i, $0 \leq i < k$. Le tableau à trier est alors restructuré, en plaçant chacun de ses éléments dans le seau qui lui correspond. Le tri peut alors être terminé en appliquant le même algorithme récursivement à chacun des seaux B_i.

L'implémentation distribuée nécessite d'augmenter la granularité en fonction du nombre de processeurs. Lorsque le nombre d'éléments est inférieur à un seuil (ici $\sqrt{n/p}$), on procède à un tri séquentiel par un algorithme de quicksort de durée moyenne $O(m \log m)$.

Entrée : Un tableau A de taille n distribué sur p processeurs.

Sortie : Le tableau trié découpé en k sous-tableaux (B_i) et distribué.

1. Restructuration en parallèle du tableau A de taille n en seaux de taille inférieure à $l = \sqrt{n/p}$. Pour cela :
   – Choix de $n/l - 1 \approx \sqrt{np}$ pivots pris dans A (les $n/l - 1$ premiers éléments) et tri séquentiel de ces pivots.
   – Restructuration en parallèle de A en n/l seaux $B_1, \ldots, B_{\frac{n}{l}}$.
   – Restructuration récursive des seaux B_i de taille supérieure à l.

2. En parallèle, tri séquentiel, par un algorithme de quicksort, des k seaux de A obtenus à l'étape 1.

Proposition 1 *Le coût parallèle en moyenne de cet algorithme sur une PRAM-EREW à p processeurs est $O(\sqrt{np} \log np + \frac{n}{p} \log np + \frac{n}{p} \log \frac{n}{p})$. L'algorithme est optimal pour $p \leq \sqrt[3]{n}$.*

La durée d'un quicksort séquentiel de m éléments est inconnue, mais bornée entre $O(m)$ et $O(m^2)$. La difficulté est alors de mettre en œuvre un mécanisme de régulation garantissant un ordonnancement optimal.

3. Régulation théorique sur PRAM

Le problème de l'ordonnancement d'un ensemble de tâches de durée inconnue peut, sous certaines hypothèses, être résolu de manière optimale sur une PRAM [1, 3].

Proposition 2 *[1] Soient k tâches indépendantes de durées $t_1, \ldots, t_k$ inconnues et bornées par t_{max} et soit $T_{seq} = \sum_{i=1}^{n} t_i$. Les k tâches peuvent être exécutées sur p processeurs en temps $O(\frac{T_{seq}}{p})$ pour tout $p \leq \frac{T_{seq}}{Max(t_{max}, \log k)}$.*

Pour le cas $p \leq \frac{T_{seq}}{t_{max} \log n}$, un ordonnancement optimal peut être réalisé par une étape de régulation synchrone tout les $\log n$ pas de calcul de manière à ce que chaque processeur dispose pour les $\log n$ pas suivants de calcul de $\log n$ instructions au moins, ce qui garantit un travail optimal, avec un faible surcoût.

Par cette proposition, l'algorithme de tri proposé peut être ordonnancé optimalement sur les p processeurs, les k tâches à ordonnancer étant de coût borné par $t_{max} = n/p$ (puisque la taille des sous tableaux est borné par $\sqrt{n/p}$) et le coût séquentiel total T_{seq} supérieur à n.

4. Expérimentation

La mise en œuvre a été réalisée en C, à l'aide de la bibliothèque ATHAPASCAN [2]. La fonction de tri séquentielle choisie est la fonction `qsort` de la librairie C standard. Les éléments à trier sont des entiers (32 bits) et la fonction de comparaison est :

```
int IntCompare (int *i, int *j) { return (*i - *j) ; } .
```

L'ordonnancement des tâches de quicksort séquentiel est réalisé de deux manières différentes.

4.1. Placement par prédiction de coût. Si n_i est le nombre d'éléments du seau B_i, le coût du tri séquentiel de ce seau peut être estimé à $n_i \log n_i$, durée moyenne de l'algorithme de quicksort. Un placement équilibré tenant compte de cette information est réalisé de manière centralisée par un processeur au début de la deuxième étape.

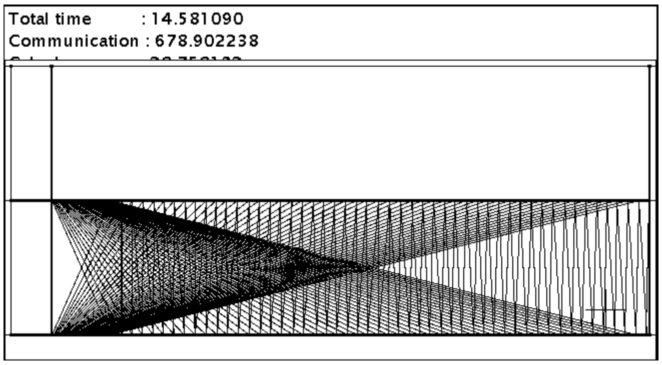

FIG. 1 - . Graphe de l'algorithme exécuté sur deux processeurs. Chaque trait gras représente un processus léger (*thread*) actif, et chaque trait fin une communication.

Le graphe 1 montre une exécution de cette implémentation sur deux processeurs. Une observation qui peut être faite est que le placement des tâches immédiatement après leur création permet, par le lancement des communications nécessaires à la tâche au plus tôt, de recouvrir totalement les communications. La faible variance du coût du quicksort rend cette stratégie très performante en moyenne.

4.2. Ordonnancement de tâches de durée inconnue. Les tâches sont ordonnancées par une stratégie à seuil centralisée. Les tâches, regroupées sur un processeur maître, sont distribuées à la demande aux autres processeurs. Lorsque le nombre r_i de tâches restantes sur un processeur est inférieur à un seuil fixé $seuil_{min}$, ce processeur demande $seuil_{max} - r_i$ tâches au maître.

Le choix des seuils est important dans l'efficacité de l'algorithme. Si $seuil_{min}$ est trop faible, le processeur épuise sa réserve avant que sa requête ait été satisfaite. Si $seuil_{max}$

est trop élevé, les temps d'inactivité en fin d'exécution sont trop importants et affectent alors l'efficacité.

La figure 2 montre les résultats obtenus en utilisant cette stratégie de régulation sur l'exemple du tri. Cette approche donne de bons résultats sur cet algorithme, 10 millions de mots de 32 bits sont triés en 156 secondes en séquentiel et en 8 secondes sur 29 processeurs, soit une efficacité de 67%.

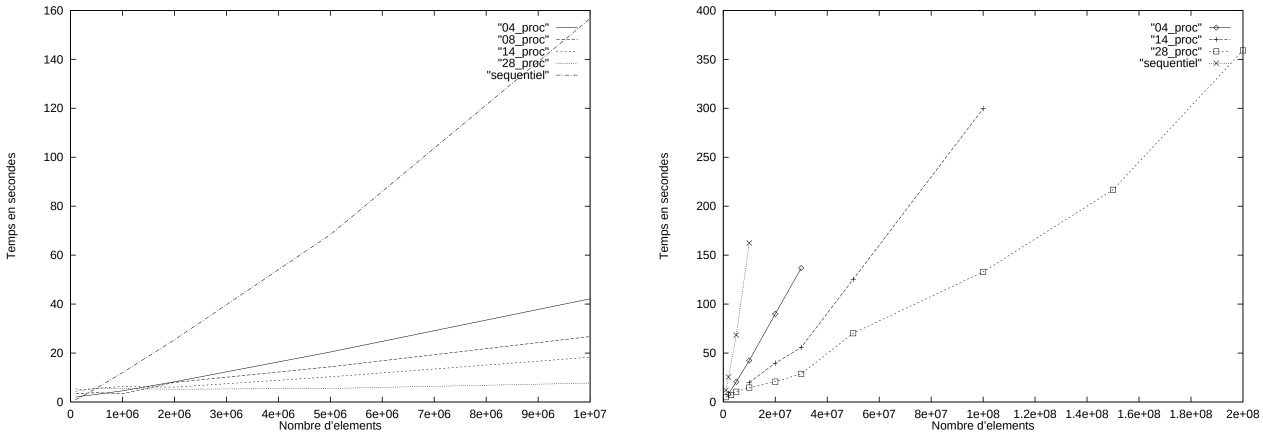

FIG. 2 - . Tri d'un tableau d'entier réalisé séquentiellement (fonction c qsort) et en parallèle.

5. Conclusion

Nous avons expérimenté deux stratégies de régulation sur un exemple d'application. La stratégie "durée inconnue" est très prometteuse car elle ne se base que sur la fin des tâches tout en assurant l'optimalité théorique. Elle ouvre donc des perspectives sur la régulation de charge dans un environnement multi-utilisateur interactif comme les réseaux de station de travail, puisqu'elle peut réagir dynamiquement à la surcharge de certains processeurs.

Bibliographie

1. R. Cole and U. Vishkin. Approximate parallel sheduling. Part I: The basic technique whith applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.*, 17(1):128–142, February 1988.
2. M. Cristaller. ATHAPASCAN-0A sur PVM3: Définition et mode d'emploi. Technical Report 11, Equipe APACHE, LMC-IMAG, June 1994.
3. J. L. Roch G. Villard et C. Roucairol. Algorithmes irréguliers et ordonnancement. In *Parallélisme et applications irrégulières*, chapter 4, pages 71–87. Hermes, 1995.
4. F. Feschet S. Miguet et L. Perroton. Parlist: Une structure de données parallèle pour l'équilibrage des charges. In *Parallélisme et applications irrégulières*, chapter 8, pages 177–201. Hermès, 1995.
5. J. Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.
6. A. Ranade. A framework for analyzing locality and portability issues in parallel computing. In *Parallel Architectures and their Efficient Use*, number 678 in LNCS, pages 185–194, 1993.
7. R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM J. Computing*, 14(2):396–409, 1985.