Graphe de flot de données *Athapascan* et ordonnancement

Mathias Doreille, François Galilée, Jean-Louis Roch[a] *

[a]`Mathias.Doreille@imag.fr, Francois.Galilee@imag.fr, Jean-Louis.Roch@imag.fr`,
LMC-IMAG, Institut Fourier BP 53X, 100 rue des mathématiques, 38041 Grenoble Cedex 9.

Mots clef : graphe de flot de données, de dépendances, mémoire partagée, ordonnancement.

Résumé

Nous présentons dans cet article la génération du graphe de flot de données au sein du langage *Athapascan*. Ce graphe, destiné à la gestion des dépendances entre tâches, se transforme aisément en un graphe d'ordonnancement, ce qui permet d'intégrer efficacement les résultats de la théorie d'ordonnancement.

1. Introduction

La mise œuvre d'un langage parallèle permettant de virtualiser le nombre de processeurs nécessite celle d'un mécanisme d'ordonnancement pour placer les calculs et router les données de manière à minimiser le temps d'exécution. La séparation entre l'application parallèle et son ordonnancement (qu'il soit calculé statiquement ou dynamiquement) est alors rendue possible par la construction du graphe de flot de données, central en théorie de l'ordonnancement, qui définit à la fois les dépendances de données et les précédences des calculs [2].

Dans un cadre statique, sous certaines hypothèses, la construction d'un tel graphe est possible [1]. Dans un cadre dynamique, certains environnements manipulent une version restreinte de ce graphe. Ainsi dans Cilk [4], il est possible d'obtenir le graphe des calculs sans les dépendances de données ; Jade [6] permet la construction d'un graphe intégrant les dépendances de données bien que celles-ci ne soient pas prises en compte pour l'ordonnancement.

Le langage *Athapascan* [5] permet la construction *dynamique* et *distribuée* d'un tel graphe. Ce graphe permet de garantir la sémantique du langage lors de l'exécution et peut être utilisé par un algorithme d'ordonnancement. Nous présentons tout d'abord (§ 2) brièvement le langage *Athapascan*, puis expliquons (§ 3) le principe de la construction du graphe de flot de données. Enfin, nous montrons comment un algorithme d'ordonnancement tel PYRROS [3] peut être utilisé en ligne lorsque des informations de coût sont disponibles : une fois l'ordonnancement calculé, l'exécution est effectuée sans autres communications que celles des données de l'application (i.e. sans surcoût lié à la régulation).

2. *Athapascan*

Athapascan [5] est une bibliothèque C++ qui permet d'exprimer le parallélisme par appel de procédure asynchrone à distance : un tel appel est réalisé par la création d'une tâche. Une tâche décrit une procédure (prenant en argument des paramètres formels) ainsi que l'ensemble des paramètres effectifs à passer à cette procédure lors de son appel. Deux modes de passages sont disponibles, l'un par valeur (recopie en mémoire locale) et l'autre par référence (utilisation d'objets en mémoire globale). La mémoire globale (ou partagée) permet de regrouper les communications des paramètres passés en référence, la donnée n'étant émise qu'une seule fois vers un site puis partagée localement. Quatre modes d'accès sont définis pour une telle donnée : *lecture* (`a1_shared_r`), *écriture* (`a1_shared_w`), *lecture/écriture* (`a1_shared_r_w`) et *accumulation* (`a1_shared_cw`). Les trois premiers modes sont classiques [4, 6]. L'écriture par accumulation est effectuée à partir de la valeur existante de l'objet par une incrémentation qui est définie par une fonction binaire `f` (par défaut l'opérateur C++ `+=`) supposée *associative* et *commutative* ; elle permet d'effectuer localement des réductions

*. Ce travail est supporté par le projet CNRS-IMAG-INPG-INRIA APACHE.

indépendamment des sites choisis par l'ordonnanceur et d'appliquer une réduction finale sur les valeurs cumulées sur les différents sites concernés.

La sémantique des accès en mémoire partagée [2] est telle que chaque lecture voit la dernière écriture effectuée selon l'ordre séquentiel de création des tâches (profondeur d'abord). Dans l'implémentation actuelle d'*Athapascan* les conflits d'accès entre tâches sur un même objet en mémoire partagée sont résolus par des synchronisations sur les dates de début d'exécution : une tâche est dite "exécutable" lorsque toutes les données qu'elle accède en lecture ne sont plus accédées en écriture par les tâches précédentes (selon l'ordre séquentiel de leur création).

Exemple. Considérons le calcul par morceaux (Newton-Cotes) de l'intégrale d'une fonction f sur un intervalle $[a, b]$, connaissant la fonction g telle que pour tout $|x_b - x_a| < h$, $g(x_a, x_b) = \int_{x_a}^{x_b} f dx$. Nous présentons ici un programme *Athapascan* effectuant une découpe récursive jusqu'à atteindre le pas h désiré puis accumulant les intégrales de chaque morceau :

```
task compute(float xa, float xb,
             a1_shared_cw<float> res) {
  if((xb-xa) < h) {
    res += g(xa, xb);
  } else {
    new_task(compute, xa, (xb-xa)/2, res);
    new_task(compute, (xb-xa)/2, xb, res);
  }
}
```

La tâche de calcul prend en paramètres les bornes (par valeurs) et un objet en mode accumulation pour stocker le résultat. Si l'intervalle est assez petit, il y a accumulation de la valeur de l'intégrale sur le morceau, sinon il y a création de deux tâches sur une moitié d'intervalle chacune.

```
task print (a1_shared_r<float> res) {
  cout << res;
}
```

La tâche d'affichage du résultat demande la donnée en lecture.

```
main () {
  a1_shared_r_w<float> res;
  float a, b;

  new_task(compute, a, b, res);
  new_task(print, res);
}
```

La tâche principale crée les objets puis lance, en séquence, une tâche de calcul, puis une tâche d'affichage dont l'exécution sera retardée jusqu'à ce que plus aucune tâche ne possède la donnée `res` dans le mode d'accumulation.

3. Construction du graphe de flot de données

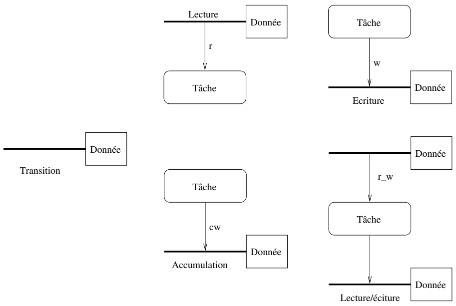
Il n'existe pas en *Athapascan* d'effet de bord entre les tâches : le seul moyen de modifier une donnée partagée est de l'avoir soit reçue en paramètre, soit déclarée. Il est alors possible de suivre chaque donnée à travers le graphe de création des tâches, et donc de construire le graphe des accès aux données. Ce graphe est un graphe orienté sans cycle (DAG) et peut être représenté sous la forme d'un réseau de Petri sans cycle dans lequel les *transitions* correspondent aux données en mémoire partagée (orientées d'écriture vers lecture) et les *places* aux exécutions des tâches. Plusieurs transitions peuvent être associées à une même donnée : chaque transition correspond à une assignation unique (éventuellement concurrente lors d'une accumulation) de la donnée.

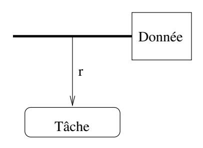
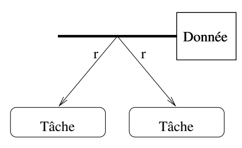
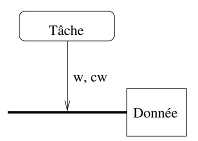
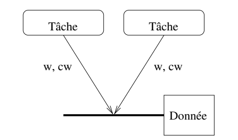
Chaque transition est nommée par le nom de l'objet partagé concerné. Toutes les transitions associées à un même objet ont donc le même nom, ce qui permet de réaliser des calculs en place (sans recopie locale sur un même site). Nous donnons dans la figure suivante la représentation d'une transition ainsi que les quatre éléments de base du graphe qui correspondent aux quatre modes d'accès en mémoire partagée : une lecture provient d'une transition de la donnée à lire, une écriture (simple ou cummulative) aboutit à une transition sur la donnée écrite et un accès en lecture/écriture combine les deux précédents.

La génération dynamique du graphe de flot de données correspondant à l'exécution d'un programme *Athapascan* s'effectue à partir de ces éléments. Dans le cadre restreint que nous considérons ici, une tâche possédant une donnée partagée ne peut passer cette donnée à une

2. *Athapascan* [5] offre d'autres types d'accès en mémoire partagée : accès différés permettant une plus grande expression de parallélisme et définition de collections d'objets partagés. Leur implémentation n'est pas décrite ici mais repose sur le principe exposé sur les modes d'accès élémentaires.

tâche fille que si les modes d'accès des deux tâches sont identiques. De plus, une donnée possédée en lecture/écriture ne peut pas être transmise. Le tableau suivant contient les modifications apportées systématiquement au graphe lorsqu'une tâche crée une nouvelle tâche en lui passant un objet partagé en paramètre. Il est à noter qu'aucune supposition n'est faite sur les transitions : il peut y avoir d'autres arêtes (entrantes ou sortantes) que celles représentées.



accès	avant création	après création
`a1_shared_r`		
`a1_shared_w` ou `a1_shared_cw`		

3.1. Un exemple didactique

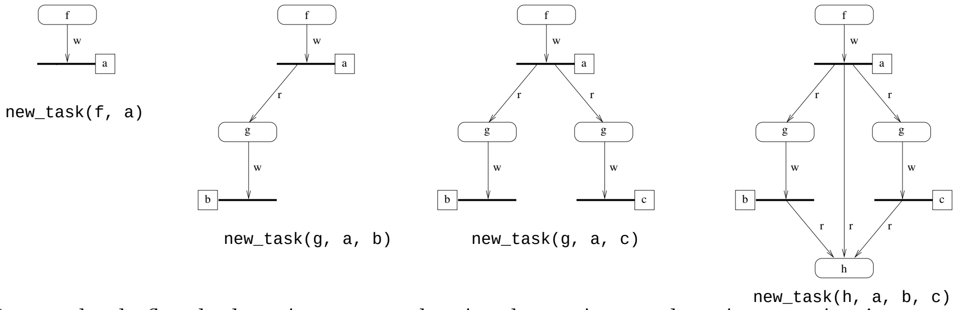
Considérons le programme *Athapascan* suivant :

```
task f(a1_shared_w x) {
  x = ...
}

task g(a1_shared_r x,
       a1_shared_w y) {
  y = x
}
```

```
task h(a1_shared_r x,
       a1_shared_r y,
       a1_shared_r z) {
  ...
}
```

```
main () {
  a1_shared_r_w a, b, c;

  new_task(f, a);
  new_task(g, a, b);
  new_task(g, a, c);
  new_task(h, a, b, c);
}
```

Le graphe de flot de données se construit alors dynamiquement de la manière suivante :



Le graphe de flot de données permet de gérer les accès aux données en mémoire partagée : f doit être exécutée avant les deux tâches g (qui peuvent s'exécuter en parallèle) qui doivent être terminées avant l'exécution de h.

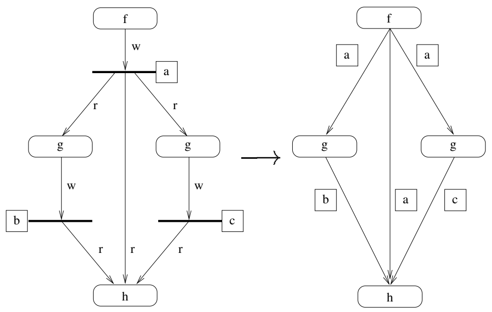
4. Surcoût de création et ordonnancement

La génération du graphe de flot de données se fait dynamiquement et de manière distribuée. A chaque création de tâche prenant k paramètres en mémoire partagée au plus $2k$ arêtes y sont rajoutées (plus précisément, le nombre total d'arêtes du graphe et égal au nombre d'accès aux données partagées, `a1_shared_r`, `a1_shared_w`, `a1_shared_cw` en faisant un et `a1_shared_r_w`, deux). Le graphe est distribué : l'adjonction est effectuée localement (sans communication) lors de l'opération `new_task`. Le surcoût total de création du graphe est en $O(T)$, T représentant le nombre total de tâches (en prenant en compte le nombre d'accès lors du passage de paramètres) générées par l'exécution complète du programme *Athapascan*[3].

Du graphe de flot de données à son ordonnancement. Le graphe précédemment construit peut alors être utilisé de manière dynamique par un algorithme d'ordonnancement

3. Les expérimentations effectuées sur un IBM-SP1.5 montrent que le coût de l'opération `new_task` pour l'exemple du calcul d'intégrales présenté précédemment est de l'ordre de 5 μs.

tel PYRROS utilisé généralement dans un cadre statique [1]. Lors de la création d'une tâche, des attributs de coût peuvent être fournis à **`new_task`** qui seront évalués lors de l'appel [5]. Ces informations sont ajoutées comme décors sur les nœuds (coût de calcul) et arêtes (taille de donnée) du graphe. Pour permettre l'utilisation de PYRROS, le graphe – construit distribué – est centralisé sur un processeur puis transformé dans le format requis en temps $O(T)$ (parcours des arêtes). Par exemple, pour le programme précédent, la transformation est la suivante :



L'exécution se déroule dans ce cas en trois phases : 1/ construction distribuée du graphe : le programme est exécuté partiellement, les tâches élémentaires (**`task_elem`**, qui ne peuvent exécuter l'opération **`new_task`**) n'étant pas exécutées ; 2/ calcul de l'ordonnancement : centralisation du graphe obtenu, mise sous un format adéquat puis calcul de l'ordonnancement par PYRROS ; 3/ exécution avec l'ordonnancement calculé. Cette dernière phase ne requiert aucune autre communication que celles des données entre processeurs distincts. En effet, d'une part, la dépendance par mémoire partagée permet à deux tâches regoupées sur un même processeur de transférer les données partagées directement par référence sans bufferisation ni copie ; d'autre part, seules des écritures distantes sont requises lorsque deux tâches sont sur des processeurs différents (émission de la tâche qui écrit la donnée, réception par celle qui la lit), les allers-retours, nécessaires dans le cas de lecture à distance, sont ainsi évités.

Remarque : Nous avons détaillé le cas d'une exécution pour un programme lorsque seules les tâches élémentaires sont soumises à ordonnancement. De manière plus générale, dans le cas de tâches quelconques, les trois phases sont entrelacées et un ordonnancement partiel est exécuté.

Les résultats de la théorie d'ordonnancement peuvent donc être directement utilisés en *Athapascan*. Plus particulièrement, les garanties d'efficacité (compétitivité) de certains algorithmes seront respectées, le coût de création du graphe étant supporté par *Athapascan* et celui lié au calcul de l'ordonnancement par l'ordonnanceur utilisé.

5. Conclusion et perspectives

Nous avons montré comment un programme *Athapascan* définit de manière implicite un graphe de flot de données : ce graphe peut être construit de manière explicite lors de l'exécution pour servir au calcul de l'ordonnancement.

Deux implantations sont en cours pour expérimenter cette approche avec deux régulateurs, l'un dynamique basé sur la localité et utilisé dans une version précédente d'*Athapascan* et l'autre statique, PYRROS.

Bibliographie

1. M. Cosnard and M. Loi. Automatic task graph generation techniques. *Parallel Processing Letters*, 5(4):527–538, 1995.
2. T. Gautier and J.-L. Roch. Irregular algorithms and on-line scheduling. In INRIA, editor, *Proc. of Stratagem'96*, pages 17–37, July 1996.
3. A. Gerasoulis and T. Yang. PYRROS : Static Scheduling and Code Generation for Message-Passing Architectures. In *Proc. of the 6th ACM International Conference on Supercomputing*, pages 428–437, July 1993.
4. C.F. Joerg. *The Cilk system for parallel multithreaded computing.* PhD thesis, Massachussets Institute of Technology, january 1996.
5. Jean-Louis Roch Mathias Doreille, François Galilée. Athapascan-1b: Présentation . Technical Report http://navajo.imag.fr/ath1/, Projet APACHE, Grenoble, France, 1996.
6. M.C. Rinard. *The design, implementation and evaluation of Jade : a portable, implicitly parallel programming language.* PhD thesis, Stanford University, september 1994.