Omnisc'IO : une méthode basée sur les grammaires pour prévoir les entrées et sorties de données dans les systèmes parallèles de fichiers.

Thomas Bastien, sous la supervision de Shadi Ibrahim.

Présentation d'un article de Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu et Robert Ross : *Omnisc'IO : A Grammar-Based Approach to Spatial and Temporal I/O Patterns Prediction.* SC'14 - International Conference for High Performance Computing, Networking, Storage and Analysis, Nov 2014, New Orleans, United States. <hal-01025670> [1]

Résumé

Les systèmes parallèles de fichiers - utilisés par les super-ordinateurs lors de leurs simulations - font l'objet de nombreuses optimisations souvent basées sur la prélecture, la mise en cache ou l'ordonnancement. Mais afin de mettre en place de manière réellement efficace ces optimisations, il faut disposer d'un système de prédiction temporel et spatial des prochains accès aux fichiers.

Cet article présente une approche, baptisée Omnisc'IO, basée sur la construction et l'utilisation d'une grammaire en temps réel pour répondre à cette demande.

Mots-clés : *Super-ordinateur ; système parallèle de fichiers ; prédiction ; grammaire ; Omnisc'IO.*

Introduction

Aujourd'hui, ce qui limite les performances des super-ordinateurs n'est pas tant leur puissance de calcul que les problèmes de gestion des données : extraire des données d'un disque dur coûte en général plusieurs milliers de fois le temps d'une opération par un processeur. Cet écart est en particulier important lorsque le calculateur doit traiter des centaines de téra-octets de données comme c'est généralement le cas dans le calcul haute performance.

Pour pallier ce problème, des *systèmes parallèles de fichiers* ont été mis au point : il s'agit d'un système de stockage, où un même fichier est généralement réparti sur plusieurs disques, de manière à paralléliser les accès lorsqu'une application demande une donnée. (Un accès fichier se fait dans un *fichier*, avec un *décalage*, et une *taille* donnée.)

Ces systèmes peuvent encore largement être optimisés, notamment en ce qui concerne la gestion des *conflits* (plusieurs lecture/écriture sur les mêmes octets d'un fichier) : des approches se basant sur la *prélecture des données*, sur leur *mise en cache*, ou sur l'optimisation de l'*ordonnancement des accès* sont possibles. Mais ces méthodes ne peuvent être réellement efficaces que si l'on dispose d'un système de *prédiction spatial* (pour la prélecture et la mise en cache) et *temporel* (pour la prélecture et l'ordonnancement) des prochaines demandes de données des applications.

Malheureusement, les systèmes de prédiction "classiques", basés sur les statistiques sont peu efficaces puisqu'ils ne rendent pas bien compte de la structure des accès qui sont en général nichés dans plusieurs niveaux de boucles au sein du code. L'approche proposée ici consiste alors, à capturer la logique du programme dans une grammaire obtenue grâce à l'algorithme

Sequitur [2], une telle grammaire est en effet connue pour mettre en évidence les structures qui se répètent dans une séquence de symboles, de manière compacte et dynamique (c'est pourquoi Sequitur a notamment été utilisé dans des domaines comme la compression de texte), puis à utiliser cette grammaire afin de prédire les prochaines demandes des applications.

Une première partie présentera le fonctionnement d'Omnisc'IO, et une seconde partie évaluera ses performances.

1 Le fonctionnement d'Omnisc'IO

1.1 Lecture de la pile d'appel.

La grammaire utilisée par Omnisc'IO agit sur les contextes d'exécution des accès fichiers. Ceux-ci sont en fait extraits de la pile d'appels du programme. On associe à chaque contexte rencontré durant l'exécution un *symbole de contexte* (une lettre terminale de la grammaire), que l'on relie à des données spatiales et temporelles sur les accès (taille du paquet, intervalle de temps entre deux accès, décalage entre deux paquets...)

Concrètement, la fonction `backtrace` de la libc permet de récupérer la pile des appels du programme, lorsque cette fonction est appelée dans une fonction `f`, elle retourne un tableau de pointeurs `void*` caractérisant les différents appels de fonctions imbriqués depuis celui de `main`. Omnisc'IO appelle cette fonction dans les fonctions de bas niveau comme `read` ou `write` et associe à chaque tableau ainsi obtenu un entier : un symbole de contexte.

On observe alors qu'à un symbole correspond souvent une taille d'accès fichier particulière, et qu'à une transition entre deux symboles est associé un changement de décalage ainsi qu'un intervalle de temps particulier.

1.2 Construction de la grammaire.

À partir des symboles ainsi construits, on peut utiliser l'algorithme Sequitur pour générer dynamiquement une grammaire compacte qui traduit l'organisation de la structure des accès. Une grammaire $G = (\Sigma, \mathbb{V}, \mathcal{R}, S)$ ainsi construite est particulière puisqu'elle ne permet de générer que le mot x donné en entrée de l'algorithme, elle respecte de plus les deux règles suivantes :

Unicité des paires : Une séquence de deux symboles $ab \in (\Sigma \cup \mathbb{V})^2$ ne peut apparaître qu'une seule fois dans toutes les règles de la grammaire.

Utilité des règles : Chaque règle doit être utilisée au moins deux fois.

On peut générer dynamiquement une grammaire respectant ces contraintes de la manière suivante : La grammaire commence avec une seule règle S, à laquelle on ajoute une à une les lettres de x, en faisant en sorte que les règles continuent à être respectées.

— Si la séquence ab apparaît plus d'une fois (non respect de l'*Unicité des paires*), alors une règle $R \to ab$ est ajoutée à la grammaire, et on remplace par R toutes les instances de ab. Les règles sont réappliquées récursivement.

— Si une règle n'apparaît qu'une seule fois (non respect de l'*Utilité des règles*), elle est effacée et son instance est remplacée par son contenu. Les règles sont réappliquées récursivement.

Par exemple, pour $x = abcdbc$

— $S \to aAdA$

— $A \to bc$

Et pour $x = abcdbcabcdbc$

— $S \to AA$

— $A \to aBdB$
— $B \to bc$

1.3 Ajout des prédicteurs

Mais cela n'est pas suffisant puisque le modèle ne peut pas encore faire de prédictions : pour cela, on ajoute de nouveaux symboles : des *prédicteurs*. Il s'agit en fait de lettres terminales marquées comme étant des possibilités pour les prochains accès fichiers du programme. On étend ces marqueurs aux lettres non terminales grâce aux deux règles suivantes :

Prédicteurs intérieurs : Un symbole non terminal est un prédicteur que si l'un des symbole de la règle associée l'est aussi.

Utilité des prédicteurs : Si un symbole x (terminal ou non) est un prédicteur dans une règle Y, alors il doit exister au moins une règle Z dans laquelle une instance de Y est un prédicteur.

Les prédicteurs vont alors devoir "avancer" dans la grammaire au cours de l'exécution de l'application. En effet, si un prédicteur a bien prédit le dernier accès fichier, il est fort probable que le prochain accès soit celui prédit par le symbole qui suit directement ce prédicteur dans le mot x généré par la grammaire. Concrètement, on *met à jour* les prédicteurs après chaque accès de la manière suivante :

Mise à jour des prédicteurs : La *mise à jour des prédicteurs* commence par enlever les marques de tous les prédicteurs qui n'ont pas correctement prédit le dernier accès fichier en faisant en sorte que les contraintes soient encore respectées. On *incrémente* ensuite les prédicteurs restants : on enlève les marques sur les prédicteurs terminaux et on marque leurs successeurs, si le c'est un non-terminal, on marque aussi la première lettre de la règle associée, et si le prédicteur était au bout d'une règle, on marque les successeurs de toutes les instances du non terminal marqué associé à la règle.

Découverte de prédicteurs : Si tous les prédicteurs ont été supprimés, il faut en générer de nouveaux, pour cela, on marque toutes les lettres accessible qui correspondent au dernier symbole vu, on fait en sorte que les contraintes soient respectées, puis on met à jour les prédicteurs.

Quelques exemples :

— Un exemple de mise à jour :
  — $S \to \underline{a}bA\underline{A}e$
  — $A \to \underline{c}d$
  devient après la lecture de c :
  — $S \to abA\underline{A}e$
  — $A \to c\underline{d}$
— Un exemple de découverte de nouveaux prédicteurs :
  — $S \to abAAeb$
  — $A \to cdb$
  devient après marquage des instances de b :
  — $S \to a\underline{b}AAe\underline{b}$
  — $A \to cd\underline{b}$
  et après mise à jour :
  — $S \to ab\underline{AA}e\underline{b}$
  — $A \to \underline{c}db$

1.4 Prédictions par Omnisc'IO

La prédiction par Omnisc'IO est alors simplement l'ensemble des symboles terminaux prédicteurs de la grammaire. Il y a donc en général plusieurs prédictions possibles, on choisira par la suite une probabilité uniforme parmi ces dernières. Il reste encore à relier les comportements des accès aux symboles. Différentes procédures sont utilisées selon la donnée à mémoriser :

- **Taille des accès :** Le plus souvent, à chaque contexte est associée une unique taille d'accès, auquel cas associer le contexte et la taille de l'accès est trivial. Lorsque plusieurs tailles d'accès sont toutefois associées à un même symbole, on associe à ce symbole une nouvelle *grammaire locale* générée exactement de la même manière que la grammaire globale, en remplaçant les symboles de contexte par les tailles des accès fichiers. Ces grammaires locales ne sont mises à jour que lorsque le symbole associé apparaît.
  Si le nombre d'accès devient trop important (expérimentalement, s'il dépasse $N = 24$), alors on considère que la taille de ces accès est aléatoire et on ne retient que la moyenne, la taille minimale et la taille maximale des accès.
- **Décalage des accès :** Les modifications de décalage des accès sont classées en trois catégories :
  - Les *transformations contiguës* sont celles qui placent le décalage juste après la fin de la précédente lecture. (Beaucoup de systèmes de prélecture supposent automatiquement une telle configuration.)
  - Les *transformations absolues* sont celles qui réinitialisent le décalage à une valeur spécifique.
  - Les *transformations relatives* modifient la valeur du décalage en fonction de sa valeur précédente.

  Au niveau auquel opère Omnisc'IO, il est impossible de distinguer les transformations absolues des transformations relatives. Les transformations absolues ne seront alors que les transformations qui mettent le décalage à zéro (fonctions `open` et `close`) les autres seront considérées comme étant relatives.
  Omnisc'IO associe à chaque transition entre deux symboles a et b la transformation du décalage qui lui est associé, si plusieurs décalages sont associés à la même transition, le même type de grammaire locale que pour la prédiction des accès est créée. Si un trop grand nombre de décalages est associé à la même transition, Omnisc'IO supposera des accès contigus.
- **Pointeurs des fichiers :** Pour que la prédiction des décalages soit utile, il faut savoir dans quel fichier l'accès se fait. Pour cela Omnisc'IO associe aussi à chaque transition de symboles les changements éventuels de pointeurs de fichiers.
- **Intervalle de temps entre les accès :** Omnisc'IO enregistre et met à jour plusieurs données sur les intervalles de temps entre les accès : La moyenne, le maximum, le minimum, la médiane, ainsi qu'une moyenne pondérée. Ces données sont suffisantes pour donner une approximation de la durée avant le prochain accès ainsi que de la confiance que l'on peut avoir envers cette prédiction. La moyenne pondérée place un poids deux fois plus important au dernier accès , quatre fois moins à l'avant dernier etc. Afin de prendre en compte les changements qui peuvent survenir au cours de l'évolution du programme.

La prédiction des prochains accès peut alors se faire en récupérant les prédicteurs de la grammaire et en construisant pour chaque prédicteur un triplet *(taille, décalage, date)*. Il est même possible de prévoir plusieurs étapes en avance, mais cela n'a pas été fait ici.

2 Évaluation des résultats

Pour évaluer les performances d'Omnisc'IO, quatre applications ont été choisies :

— **CM1** sert à la modélisation du climat, trois variantes sont utilisées ici selon les méthodes de gestion du système parallèle de fichiers.
— **GTC** est utilisé dans la recherche sur la fusion nucléaire.
— **Nek5000** est utilisé pour modéliser la dynamique des fluides.
— **LAMMPS** est utilisé dans le milieu de la dynamique moléculaire.

Les simulations ont été faites sur le site français *Grid'5000*, en utilisant 512 des 736 coeurs disponibles sauf pour Nek5000 qui n'en utilise que 32. Les différents aspects de la prédiction sont évalués de la manière suivante :

— **Prédiction du contexte :** La précision de la prédiction des contextes a été évaluée sur ces quatre applications en attribuant éventuellement un poids uniforme à chaque symbole lorsque plusieurs sont prédits. Pour CM1, GTC et LAMMPS, Omnisc'IO converge vers 100% de prédiction correcte au bout d'une ou deux itérations de la boucle principale de l'application. Seule l'application Nek5000 pose problème avec des baisses de prédictions récurrentes qui semblent être dues à un branchement particulier du code.
— **Prédiction de la taille d'accès :** L'erreur sur les tailles d'accès est mesurée relativement à la taille observée. Pour toutes les applications à l'exception de Nek5000, on observe une erreur égale ou proche de zéro au bout de quelques itérations. Les problèmes observés sur Nek5000 résultent directement des problèmes de prédiction des contextes.
— **Prédiction des décalages :** La précision de la prédiction des décalages est mesurée de la même manière que celle des contextes, en associant un poids relatif à chaque prédiction correcte en fonction du nombre de prédictions proposées par Omnisc'IO. On observe toujours des prédictions plus efficaces que la méthode mise en place habituellement consistant à considérer automatiquement des accès contigus, allant jusqu'à un passage de $47,4\%$ de prédiction correcte si on suppose des accès contigus à $92,2\%$ avec Omnisc'IO.
— **Taux de prédiction :** En combinant la prédiction du décalage et des tailles d'accès, on peut construire une mesure plus utile : le *taux de prédiction*. Son but est de mesurer quelle partie du fichier a effectivement été correctement prédite par Omnisc'IO. En notant $S = [a_d, a_f]$ l'intervalle du fichier prédit par Omnisc'IO et $S_0 = [b_d, b_f]$ l'intervalle réellement accédé, il se calcule ainsi : $T(S|S_0) = \frac{100*|S \cap S_0|}{\max(a_f,b_f) - min(a_d,b_d)}$ Le plus bas taux observé est de $79,5\%$ pour l'une des variantes de CM1, mais ce taux atteint 100% (arrondi au 0.1% le plus proche) pour GTC.
— **Prédiction temporelle :** La moyenne de l'écart entre l'arrivée d'une demande et sa prédiction ne dépasse pas les $0,2s$. Pour certaines applications comme LAMMPS, elle est même inférieure à une milliseconde.

Omnisc'IO présente des résultats très satisfaisants à un moindre coût : la taille de la grammaire ne dépasse pas quelques centaines de kilo-octets. Il n'est toutefois pas infaillible : il lui est impossible de prédire directement les accès d'une application qui n'aurait pas de comportement périodique vis-à-vis du système parallèle de fichiers (certaines applications n'écrivent leurs résultats qu'à la fin de leur exécution), et comme l'a montré le cas de Nek5000, il est de plus sensible à certaines structure du code.

Conclusion

Le problème de l'optimisation des systèmes parallèles de fichiers est large, pourtant, beaucoup de ces optimisations peuvent bénéficier d'un système de prédiction simple, dynamique, polyvalent et précis.

La capacité d'Omnisc'IO à faire des prédictions précises, tant spatiales que temporelles, offre de nouvelles opportunités à des algorithmes de prélecture, de mise en cache ou d'ordonnancement, et offre même la possibilité de combiner ces 3 méthodes tout en partageant un unique système de prédiction.

References

[1] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Robert Ross. Omnisc'io: A grammar-based approach to spatial and temporal i/o patterns prediction. Nov 2014.

[2] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. pages 67–82, 1997.