Nestor : un système de parallélisation automatique

Georges-André Silber

Projet CNRS – ENS Lyon – INRIA *ReMaP*
Laboratoire de l'informatique du parallélisme
École normale supérieure de Lyon
69364 Lyon cedex 07, FRANCE

Résumé

Nestor est à la fois un outil de manipulation de code source pour la grammaire complète d'HPF 2.0 et un système de parallélisation automatique puissant, simple à utiliser et extensible. Nestor se présente sous la forme d'un ensemble de bibliothèques orientées objet écrites en C++. De nombreuses autres bibliothèques permettent de transformer du code source en une représentation intermédiaire nécessaire pour restructurer le code, mais cette représentation intermédiaire est souvent de bas niveau (donc difficile à manipuler) ou représente un sous-ensemble de langage. Nestor supprime ces deux défauts, et grâce aux algorithmes de parallélisation que nous avons implémentés, permet de transformer automatiquement du code Fortran séquentiel en HPF.

Mots-clés : HPF, parallélisation automatique, transformation de code.

---

1. Introduction

La parallélisation automatique consiste à transformer automatiquement un programme séquentiel en un programme parallèle. Avec l'essor actuel de High Performance Fortran (HPF), il serait très utile d'avoir un outil permettant de transformer automatiquement un code écrit en Fortran séquentiel en un code HPF. La communauté scientifique faisant de la recherche en parallélisation automatique a produit de nombreux résultats théoriques prometteurs au cours de cette dernière décennie mais très peu de ces résultats sont réellement intégrés dans de vrais compilateurs. Le but de Nestor est d'offrir un environnement de programmation permettant de construire des systèmes de transformation de programmes. De plus, Nestor intègre d'ores-et-déjà un ensemble d'algorithmes de détection de parallélisme dans les nids de boucles et d'algorithmes de transformation de code séquentiel Fortran en HPF 2.0. Nestor est notre base de développement pour l'implémentation des résultats théoriques de notre équipe et est intégré en tant qu'outil de parallélisation dans TransTool [1].

Nestor peut être utilisé par les chercheurs s'intéressant aux compilateurs-paralléliseurs, aux outils d'analyse de performance ou plus généralement aux optimiseurs de code source. Il existe beaucoup de bibliothèques permettant de transformer du code source en une représentation intermédiaire (SUIF [6], PIPS [4], ...), mais cette représentation intermédiaire est souvent de bas niveau (donc difficile à manipuler) ou représente un sous-ensemble de langage. Par exemple, SUIF n'analyse pas directement le Fortran, mais utilise *f2c*, un utilitaire du domaine public, pour transformer le Fortran en C et ensuite l'analyser. Notre bibliothèque, permettant de manipuler et de restructurer du code, supprime ces deux défauts. En nous fondant sur la partie frontale d'un excellent compilateur HPF du domaine public, Adaptor (écrit par T. Brandes [2]), nous avons construit un système permettant d'obtenir une représentation intermédiaire d'un code écrit en HPF 2.0 sous la forme d'une collection hiérarchique d'objets C++. Une fois sous cette forme, le code peut être analysé, transformé et réécrit en HPF 2.0.

2. Structure de Nestor

La structure globale de Nestor est présentée par la Fig.1. Nestor est éclaté en trois bibliothèques différentes qui communiquent entre elles et qui ont chacune un rôle spécifique : `node`, `base` et `graph`.

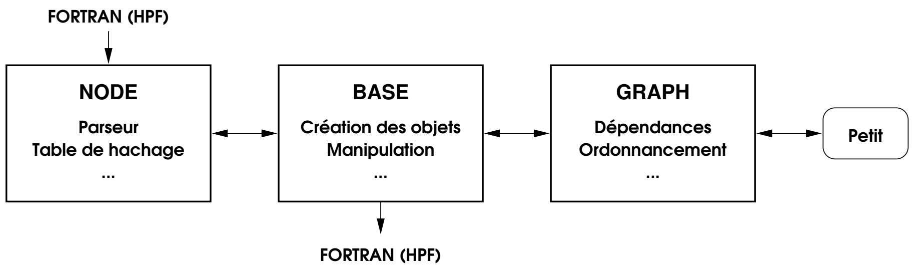

FIG. 1: Structure globale de Nestor.

2.1. Analyse syntaxique et sémantique : `node`

Nestor intègre la partie frontale du compilateur HPF Adaptor [2] qui reconnaît le langage HPF 2.0, c'est-à-dire Fortran 95 plus des directives de compilation. Le fait de pouvoir traiter des codes écrits en « *vrai* » Fortran est d'un intérêt non négligeable puisque l'on peut alors traiter des vrais codes, à la différence de la plupart des outils de parallélisation qui prennent uniquement en compte un sous ensemble de langage ou un langage ad-hoc (Petit [5], SUIF, ...). D'autre part, cette approche permet de conserver la sémantique d'un langage de haut-niveau présente dans un programme Fortran, à la différence d'un programme traduit de Fortran en C.

Le partie frontale a été étendue afin d'intégrer de nouvelles directives destinées à donner à l'utilisateur le contrôle sur la parallélisation de son code. Ainsi, il existe une directive `SINGLE` qui placée devant un nid de boucle signifie à Nestor qu'il doit traiter cette boucle[1]. D'autres directives permettent de préciser à Nestor qu'il doit prendre en compte tout le code ou encore une région de code. Ces directives permettent donc un contrôle fin des traitements effectués sur un code : complètement ou semi automatiques.

Une fois cette phase terminée, nous avons un arbre syntaxique abstrait en mémoire. La bibliothèque `node` de Nestor qui encapsule la partie frontale d'Adaptor offre un ensemble de fonctions normalisées écrites en C, permettant de parcourir cet arbre et de récupérer les informations. Ce sont ces fonctions qui vont être utilisées par la bibliothèque `base` de Nestor pour construire les objets qui en quelque sorte paraphrasent l'arbre syntaxique original mais offrent l'expressivité et la souplesse d'utilisation qui lui manquent. L'utilisation de fonctions de parcours normalisées de l'arbre syntaxique abstrait original à plusieurs avantages. Premièrement, la partie frontale peut être écrite dans un langage différent du C++, il suffit juste de fournir ces fonctions. Par exemple, la partie frontale d'Adaptor a été écrite dans un langage de haut niveau avec la boîte à outil Cocktail [2]. Deuxièmement, il est très facile de changer la partie frontale ou d'intégrer une nouvelle version, il suffit de modifier les fonctions normalisées.

2.2. Construction des objets et manipulation : `base`

Un programme source transformé par Nestor en objets C++ peut être vu comme un arbre syntaxique abstrait où chaque nœud est un objet avec des méthodes et des attributs permettant de parcourir et de modifier très facilement cet arbre. Le concept orienté objet facilite le processus de consultation et de restructuration, en fournissant des méthodes claires spécifiques à chaque type d'objet. De plus, la notion d'héritage s'intègre naturellement dans la structure hiérarchique de la syntaxe d'un langage de programmation comme Fortran.

La bibliothèque `base` de Nestor contient la définition de toutes les classes représentant les différents éléments syntaxiques d'un code. Chaque objet possède sa propre fonction de sortie (la redéfinition de l'opérateur << de C++) qui est configurable globalement. C'est ce qui a, par exemple, permis l'écriture d'une sortie dans le langage de Petit.

2.3. Graphes de dépendances : `graph`

L'analyse des dépendances entre les instructions est la première étape avant d'envisager de modifier l'ordre d'exécution dans un programme par une transformation quelconque. Un outil de parallélisation se doit donc de posséder une telle analyse. Le développement d'un analyseur de dépendances puissant et rapide est une tâche ardue sachant que le problème est NP-complet en théorie, si ce n'est indécidable. Il

[1] Cette directive est un commentaire Fortran, le programme n'est donc pas modifié.

peut cependant être résolu de manière suffisamment rapide par un ensemble de méthodes appropriées au cas les plus fréquents. Pour ces raisons, nous avons décidé d'utiliser un logiciel du domaine public, Petit [5], qui est puissant et rapide dans la plupart des cas. Petit prend en entrée un langage restreint, proche de Fortran, mais suffisant dans la plupart des cas courants. La bibliothèque `graph` permet, en s'aidant de Petit, de construire un graphe des dépendances depuis un objet `NstStatement` de la bibliothèque `node`. Chaque dépendance est décrite avec une approximation par un niveau, un vecteur de direction et un polyèdre (du plus au moins précis).

3. Exemples d'utilisation

Nous allons présenter trois exemples permettant d'entrevoir la puissance et la simplicité d'utilisation de Nestor. Ces caractéristiques sont grandement dûes au langage C++ qui permet, notamment grâce aux fonctions virtuelles, une conception claire et robuste des classes.

3.1. Modification d'indices de boucle

Voici un extrait de la définition de la classe `NstStatementDo` représentant la construction `DO` de Fortran :

```
class NstStatementDo : public NstStatement {
  NstExpression* lower_bound ();
  NstExpression* upper_bound ();
  NstExpression* lower_bound (NstExpression& new_expression);
  NstExpression* upper_bound (NstExpression& new_expression);
  ...
};
```

En présence d'un `NstStatementDo` lors du traitement d'un code source, il est très facile de changer les indices de boucle. Soit une boucle `DO I = 1,N` que l'on veut transformer en `DO I = 2,N+1`. Le code suivant effectue cette opération, avec `sd` qui est un `NstStatementDo` :

```
NstInteger deux = 2;
NstExpressionBinary nplusun = NstVariable(N) + 1;
sd->lower_bound (deux);
sd->upper_bound (nplusun);
```

3.2. Renommage d'identificateur

Plutôt que de dupliquer l'objet `string` correspondant à un identificateur (variable, procédure,...), Nestor stocke celui-ci dans un endroit unique. Par exemple, les objets de type `NstVariable` correspondant à une variable `I` qui se trouve à plusieurs endroits dans la hiérarchie construite par Nestor (chaque fois que la variable `I` est utilisée) possèdent tous en commun un objet `NstObjectVariable` contenant des informations sur cette variable (sa distribution éventuelle, sa déclaration, etc...) et notamment son identificateur. Dès lors, les instructions suivantes :

```
// v est un objet de type NstVariable representant I
NstIdentifier new_id = "TOTO";
NstIdentifier* old_id = v->object()->identifier(new_id);
```

ont comme effet de remplacer dans tout le code l'identificateur représentant la variable `I` par `TOTO` (dans la déclaration de `I` et dans toutes ses utilisations). Il est important de noter qu'une variable qui s'appellerait `I` dans une autre procédure ne serait pas modifiée.

3.3. Un paralléliseur simple

La fonction `mark_parallel_loops()` est un exemple permettant de voir la simplicité d'écriture d'un algorithme de parallélisation avec Nestor.

```
void mark_parallel_loops (NstStatementList& sl)
{
  // Examine chaque instruction de la liste sl
  for (NstStatementList::iterator i = sl.begin(); i != sl.end(); i++)
    {
      NstStatementDo* do_loop = (*i)->isNstStatementDo();
```

```
    if (do_loop && loop_is_parallel (do_loop))
        // L'instruction est un DO et est parallèle
        do_loop->independent = TRUE;
    // Parcourt récursivement les sous-instructions
    NstBranches &b = (*i)->branches;
    for (NstBranches::iterator j = b.begin(); j != b.end(); j++)
      mark_parallel_loops(*(*j));
  }
}
```

Le principe de cette fonction est d'examiner chaque instruction de contrôle (un `NstStatement`) de la liste passée en paramètre. Si cette instruction de contrôle est une boucle `DO`, la fonction `loop_is_parallel()` va tester si la boucle est parallèle sans modification. Ce test est très simple puisqu'il suffit de tester si il n'y a pas de dépendance portée par cette boucle. Dans un deuxième temps, on va examiner récursivement toutes les branches de l'instruction de contrôle que l'on est en train d'examiner (les branches `THEN` et `ELSE` d'un `IF`, le corps d'un `DO`, les différents cas d'un `CASE`, etc...). Après application de cette fonction, toutes les boucles étant parallèles auront, lors de l'écriture du programme, la directive `INDEPENDENT` d'HPF placée devant elles.

4. Parallélisme à grain moyen

Nous terminons actuellement l'implémentation dans Nestor des résultats théoriques issus de [3]. Ces nouveaux résultats théoriques permettent premièrement de caractériser l'espace de recherche des transformations mettant en évidence des parallélisations à grain fin, gros grain ou grain moyen (tiling), et deuxièmement, d'effectivement trouver une transformation dans cet espace. Nous sommes donc capables de paralléliser un nid de boucles et de choisir le grain de la parallélisation en fonction de la machine cible. Il est à noter que pour le grain moyen, la génération effective du code en terme de placement de données pose encore des problèmes théoriques.

5. Conclusion et travaux futurs

Le système que nous venons de présenter a déjà prouvé son utilité puisque nous avons implémenté très rapidement une version modifiée de l'algorithme de parallélisation d'Allen et Kennedy prenant en compte de manière fine les dépendances de contrôle, c'est-à-dire en n'introduisant des temporaires que lorsque c'est réellement nécessaire. La simplicité d'utilisation de Nestor permet de se concentrer uniquement sur l'aspect algorithmique sans perdre de temps avec les aspects de représentation de l'arbre syntaxique. D'autre part, le fait de bénéficier d'une entrée et d'une sortie en HPF permet de paralléliser de vrais codes et d'exécuter effectivement le programme obtenu après compilation par un compilateur HPF.

Bibliographie

1. T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, J.C. Mignot, F. Desprez, and J. Roman. HPFIT : A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran : Part I : HPFIT and the TransTOOL Environment. *Parallel Computing*, 23(1-2) :71–87, 1997.
2. Thomas Brandes. *ADAPTOR Programmer's Guide - Version 5.0.* German National Research Institute for Computer Science, March 1997. `http://www.gmd.de/SCAI/lab/adaptor/`.
3. Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. *Parallel Processing Letters*, 7(4) :379–392, 1997.
4. PIPS Team. PIPS (Interprocedural Parallelizer for Scientific Programs). World Wide Web document, URL : `http://www.cri.ensmp.fr/~pips/index.html`.
5. William Pugh. Release 1.10 of Petit. World Wide Web document, URL : `http://www.cs.umd.edu/projects/omega/`.
6. Stanford Compiler Group. SUIF compiler system. World Wide Web document, URL : `http://suif.stanford.edu/suif/suif.html`.