RenPar'7,
Actes des 7^{es} Rencontres
Francophones du Parallélisme,
PIP–FPMs Mons, Belgique, 30 mai–2 juin 1995
Dekeyser, Libert et Manneback Editeurs

Programmation évolutionniste parallèle

Mouloud Oussaidène, Bastien Chopard
CUI, Université de Genève
CH-1211 Genève 4, Suisse
E-mail: {mouloud,chopard}@cui.unige.ch

et Marco Tomassini
CSCS CH-6928 Manno et LSL, EPFL
CH-1015 Lausanne-Ecublens, Suisse
marco.tomassini@di.epfl.ch

Résumé

Les algorithmes évolutionnistes sont des méthodes de recherche adaptatives s'inspirant de la sélection biologique et qui peuvent s'exprimer naturellement dans un modèle de programmation parallèle. Dans ce travail nous décrivons cette approche et discutons en détail l'environnement de programmation génétique que nous développons. Nous mentionnons ensuite quelques résultats préliminaires d'applications.

1. INTRODUCTION

Les algorithmes *évolutionnistes*, dont les algorithmes *génétiques* sont un exemple, sont des techniques de recherche adaptatives s'inspirant des mécanismes de sélection naturelle[1]. Il s'agit de procédures itératives qui se déroulent comme suit: on génère une population initiale de manière aléatoire ou heuristique, dans laquelle chaque individu représente une solution possible d'un problème donné à travers un codage opportun conduisant à une représentation en chaîne de longueur finie, par exemple une chaîne binaire. A chaque pas d'itération, appellé également une *génération*, les individus sont évalués en fonction d'un critère d'adaptation fixé.

Dans une première phase, les individus sont choisis avec une probabilité proportionnelle à leur valeur d'adaptation (ou coût), afin de favoriser les meilleurs d'entre-eux. Pour reformer une nouvelle population de taille constante et générer de nouveaux points dans l'espace de solution, on utilise les opérateurs dits génétiques.

Il y a deux types principaux d'opérateurs génétiques: l'hybridation (ou *crossover*) et la mutation. l'hybridation est une recombinaison de deux individus parents produisant deux nouveaux individus. Cette création se fait en prennant dans chacun des deux parents des sous-chaînes complémentaires de leur code "génétique." Il a été montré que, sous certaines conditions[1], cette opération oriente rapidement la recherche vers des régions prometteuses.

La mutation consiste en un changement aléatoire d'un symbole dans la chaîne codant l'individu. La mutation sert à éviter que la population ne se fixe trop rapidement sur des optima locaux, phénomène que l'on nomme *convergence prématurée*. Le cycle d'évaluation/sélection, puis /recombinaison/mutation est répété jusqu'à ce que un critère de fin soit atteint. Ceci peut survenir après un nombre maximal prédéterminé de générations ou par l'obtention d'une solution satisfaisante. Les algorithmes génétiques sont des procédures stochastiques n'offrant pas de garanties de convergence absolue en un temps fini.

Il est à remarquer que les algorithmes évolutionnistes, ont été appliqués avec succès à beaucoup de problemes difficiles en optimisation de fonctions et en optimisation combinatoire, ainsi que dans l'inférence de règles de décision.

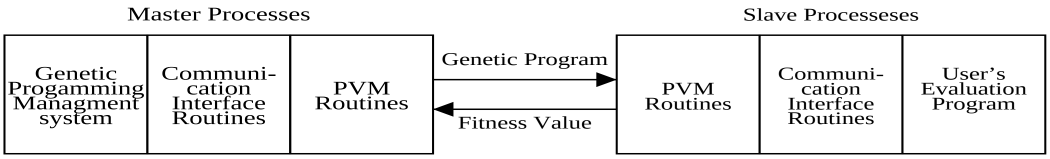

FIG. 1 - . *Structure du système du programmation génétique parallèle.*

2. PROGRAMMATION GÉNÉTIQUE PARALLÈLE

Nous présentons un système de programmation génétique parallèle pour un environnement MIMD. La programmation génétique est l'extension du modèle génétique d'apprentissage décrit plus haut à l'espace des programmes. Le concept peut être perçu comme l'application d'un algorithme génétique à un espace de recherche constitué d'expressions LISP. Dans le contexte de la programmation génétique, un individu est appellé programme génétique (PG). Le principe de la programmation génétique est détaillé dans [2].

Notre système est basé sur un algorithme parallèle obéissant à l'architecture Maître-Esclave. Le processus Maître accomplit les tâches de création de la population initiale, sélection des individus (PG) selon leur coût (ou leur valeur d'adaptation) et l'application des opérateurs génétiques. Le coût d'un individu est calculé par un processus Esclave. En l'occurence, le coût d'un PG est donné par son évaluation sur un ensemble de données. Les étapes principales des processus Maître et Esclave sont les suivantes:

<u>Processus Maître</u>:
(1) Création de la population initiale; (2) Répartition des individus sur les processus Esclave; (3) Réception des coûts des individus; (4) Sélection des individus selon leurs coûts; (5) Reproduction; (6) Réitérer les phases 2-5 jusqu'à ce qu'un "bon" PG soit trouvé.

<u>Processus Esclave</u>:
(1) Réception d'un programme génétique; (2) Evaluation du programme reçu; (3) Envoyer au processus Maître le résultat de l'évaluation (coût); (4) Réitérer les phases 1-3 jusqu'à récéption d'un signal de terminaison.

La figure 1 illustre l'architecture parallèle du système. Plusieurs processus Esclaves peuvent être reliés de la même façon au processus Maître.
Nombreuses sont les applications où le temps cpu d'évaluation d'un individu est considérable. Nous citons comme exemple l'application de la programmation génétique au marché financier où un PG est évalué sur des séries de prix dont la taille dépasse cent-mille données[3]. Par conséquent, il est judicieux d'isoler le processus d'évaluation du reste de l'algorithme et d'utiliser le parallélisme à ce niveau plutôt qu'au niveau de la maniplulation des individus[4]. Cela justifie notre choix de l'architecture Maître-Esclave.

Le présent environnement à été développé sur une IBM SP2 comprenant 14 processeurs. Les communications inter-processeurs se font par l'intermédiaire de la bibliothèque d' échange de messages PVM[5]. Le code est écrit en C++[6, 7] sur la base d'un package séquentiel du domaine publique[8].

2.1. Construction automatique d'un programme génétique

Un programme génétique peut être représenté par un arbre dont un noeud parent contient les opérations à effectuer sur les noeuds enfants. Les feuilles de l'arbre sont constituées par les données.

En fait, une telle structure arborescente n'est rien d'autre qu'une expression LISP (S-expression) constituée d'opérateurs et d'opérandes. Un opérateur est nécessairement une fonction. Un opérande peut être lui-même une fonction ou simplement un terminal. Une fonction est caractérisée par un identificateur alpha-numérique et le nombre d'arguments. Un terminal est caractérisé par sa valeur (s'il est numérique) ou un identificateur alpha-numérique (par exemple si c'est un nombre aléatoire). L'ensemble des fonctions est appellé *Function-Set* et l'ensemble des terminaux est appellé *Terminal-Set*. A la phase de création, des fonctions et des terminaux sont choisis aléatoirement tout en respectant la contrainte que toute fonction est appellée avec exactement le nombre d'arguments qui lui correspond. Ceci nous garantie la validité syntaxique (mais pas forcément sémantique) des PG. L'exemple ci-dessous illustre ce mécanisme de fonctionnement :

$$\text{Function-Set} = \{+, -, *, /\} \qquad \text{Terminal-Set} = \{x, N\}$$

où x est une variable quelconque et N un nombre aléatoire balayant l'ensemble des entiers naturels. Le système peut générer l'individu suivant:

$$\texttt{PG} = (* \;\texttt{(+ x 1) (/ x 0)})$$

Ce programme est syntaxiquement valide, mais la division par zéro le rend sémantiquement incorrecte. Pour pallier à ce genre d'exception, la solution standard consiste à redéfinir la division par zéro afin qu'elle retourne, par exemple, la valeur 1.

2.2. Opérateurs de reproduction

Notre système utilise principalement deux opérateurs de reproduction: crossover et mutation. Dans le cas de l'opérateur crossover, on commence par sélectionner deux PG et choisir un noeud sur chacun d'eux. Chaque noeud est, par définition, la racine d'un sous-arbre. Les deux sous-arbres sont extraits et interchangés. Ainsi, si on choisit les noeuds 4 et 2 respectivement sur les PG suivants (les noeuds sont comptés selon l'ordre de la S-expression)

$$(+ \; 2 \; (* \; 3 \; (/ \; 5 \; 2))) \qquad \text{et} \qquad (+ \; (* \; 4 \; 5) \; 10)$$

on aura, comme nouveaux individus

$$(+ \; 2 \; (* \; (* \; 4 \; 5) \; (/ \; 5 \; 2))) \qquad \text{et} \qquad (+ \; 3 \; 10)$$

Dans le cas de l'opérateur mutation, un noeud terminal est échangé avec un autre noeud terminal choisi aléatoirement dans le Terminal-Set, tandis qu'un noeud fonction n'est interchangeable qu'avec une autre fonction ayant le même nombre d'arguments.

2.3. Représentation mémoire d'un PG et interface de communication

Chaque noeud de l'arbre représentant un PG est une structure possédant un identificateur numérique, un pointeur gauche et un pointeur droite. Un noeud dont le pointeur gauche n'est pas nul est nécessairement un noeud fonction (dans ce cas, le pointeur gauche est le premier argument de la fonction). Le pointeur droite est utilisé pour chaîner avec l'argument suivant (s'il y en a un). Ainsi, cette représentation nous permet d'utiliser dans un même GP plusieurs fonctions n'ayant pas forcément le même nombre d'opérandes. L'évaluation du programme génétique se

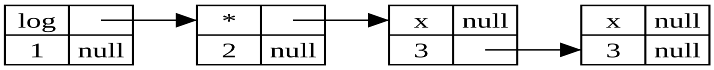

FIG. 2 - . *Représentation mémoire de* $\log(x^2)$ *par la S-expression (log (* x x)). Les pointeurs "gauche" sont representés par les flêches supérieures et les pointeurs "droite" par les flêches inférieures. Les valeurs indiquées dans le coin inférieur gauche indique le code de l'opérande dans le Function-Set ou le Terminal-Set.*

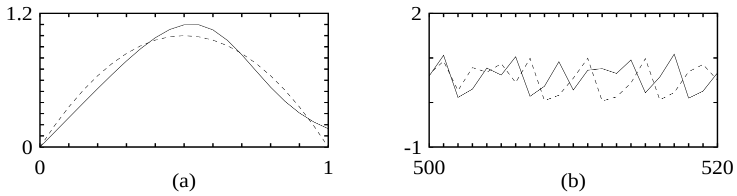

FIG. 3 - . *(a):Comparaison entre la logistic-map (en pointillé) et la fonction* $GP(x)$ *(ligne continue); (b): Prédiction "out-sample" de* $x_{i+1} = F(x_i)$ *pour* $i = 500, 501, .., 520$ *selon* $PG(x)$ *(ligne continue) et comparaison avec le résultat attendu de la logistic-map (ligne pointillé).*

fait alors récursivement de la racine jusqu'aux noeuds terminaux. La figure 2 donne un exemple de représentation mémoire de l'expression (log (* x x)).

Afin de pouvoir communiquer les programmes génétiques aux processus esclaves qui vont les évaluer, la structure en pointeur est transformée en une S-expression, sous la forme d'une chaîne de caractères qui peut être aisément transmise grâce aux routines PVM d'échange de message.

3. APPLICATION

A titre d'illustration de la méthode, nous avons étudié le problème de la prédiction des séries temporelles générées par la fonction dite de *logistic-map*. La série est donnée par:

$$x_{i+1} = F(x_i) \qquad \text{où} \qquad F(x) = 4x(1-x) \tag{1}$$

et donne lieu à un comportement chaotique déterministe dans l'intervalle $0 \leq x_i \leq 1$. En partant de la suite $x_1, x_2, ..., x_N$, où $N = 500$ et $x_1 = 0.2$, on cherche à déterminer un programme GP capable de prédire l'élément x_{i+1} de la suite en connaissant l'élément x_i.

Le coût C d'un individu GP est pris comme la somme des carrés de la différence entre les valeurs prédites $GP(x_i)$ et les valeurs attendues x_{i+1}

$$C = \sum_{i=1}^{N-1} [x_{i+1} - GP(x_i)]^2 \tag{2}$$

La meilleure solution trouvée par le système, après 100 générations et une population de 20 individus, est montrée sur la figure 3a. Elle a un coût de 3.79 et correspond à l'expression algébrique

$$GP(x) = \frac{x}{(2.7027x^9 + 2x^8 + x^4 + 0.38)} \tag{3}$$

Ce résultat n'est pas vraiment satisfaisant car $GP(x)$ est une fraction rationnelle qui n'a, algébriquement, que peu de points communs avec $F(x)$. Pour mieux capturer l'essence de notre série temporelle (une simple parabole), l'évolution génétique devrait être poursuivie sur des populations plus grandes, sur plus de générations et aussi en considérant des fonctions coût différentes ainsi que d'autres Function-Sets.

Cependant, afin de se faire une meilleure idée des possibilités prédictives de la solution trouvée en (3), il est intéressant de considérer la série temporelle générée par $GP(x)$ à partir de x_N. La figure 3b compare ces prédictions avec le résutat espéré avec F. On constate une prédiction à très court terme qui est bonne, puis un décalage qui s'accentue pour finir en un desaccord complet, ce qui est attendu pour un processus chaotic[9].

4. CONCLUSION

Nous avons présenté un système de programmation génétique parallèle pour un environnement MIMD. L'architecture de l'algorithme a été discutée ainsi qu'une application pour la prédiction des séries temporelles. Une partie de notre futur travail sera axée sur l'optimisation (au sens temps cpu et espace mémoire) des programmes génétiques générés par notre système. Une des optimisations les plus courantes est la réduction des expressions invariantes (constant folding) qui peuvent être détectées avant l'évaluation. Ainsi, il est optimal de réduire l'expression (/ x x) en (1) avant de l'évaluer sur un ensemble de données. La représentation d'une S-expression sous la forme de graphe sans cycle (DAG) nous permet de réaliser ce type d'optimisation.

Ces dernières années, la programmation génétique a suscité une attention particulière dans la communauté scientifique. Plusieurs travaux sont effectués dans les bases de données (optimisation des requêtes), finances (optimisation des modèles de marché), traitement du langage naturel (signification d'un mot selon son contexte), etc... Le parallélisme offre une plateforme très intéressante pour étudier ces problèmes difficiles en raison de son potentiel à calculer sur un grand nombre de données en un temps acceptable. Nos premières expériences nous ont montré l'importance de disposer d'un outil informatique de qualité afin de pouvoir interagir de façon adéquate sur l'évolution génétique artificielle.

Remerciements

Cette recherche est financée par le Fonds National Suisse de la Recherche Scientifique.

Bibliographie

1. G. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, Reading, MA, 1989.
2. J. Koza. *Genetic Programming.* MIT Press, 1992.
3. B.Chopard, M.Oussaidène, O.Pictet, R.Schirru, and M.Tomassini. Evolutionary algorithms for multimodal optimization in financial applications. In *SPP-IF seminar, Zurich*, 1994.
4. M.Oussaidène and B.Chopard. Optimisation génétique massivement parallèle. In *SIPAR-Workshop on Parallel and Distributed Computing.* SIPAR/Université de Fribourg, 1994.
5. *PVM 3 User's Guide and Reference Manual*, 1994.
6. Conor Sexton. *C++ Pocket Book.* NEWNES, 1993.
7. Seth Bergmann. *Compiler Design: Theory, Tools and Examples.* WCB Publishers, 1994.
8. Adam P. Fraser. *Genetic programming in C++*, 1994. Public domain genetic programming system.
9. David Campell. An introduction to nonlinear dynamics. In D.L. Stein, editor, *Lectures in the Sciences of Complexity*, pages 3–105. Addison-Wesley, Reading MA, 1989.