Modèle de Compilation d'HPF pour la Machine MIMD à Bancs Mémoire et Réseau Distribué Programmable Phénix

François IRIGOIN* Ronan KERYELL* Pierre FIORINI†

2 février 1996

Résumé

Nous décrivons une approche basée sur l'algèbre linéaire de compilation du langage parallèle HPF pour le projet PHÉNIX de co-conception matérielle et logicielle. Cette machine MIMD a la particularité d'avoir une mémoire distribuée avec plusieurs bancs mémoire programmables par processeur et un accès réseau par banc mémoire afin d'augmenter le débit données et adresses de la mémoire et du réseau. Notre approche, plutôt ciblée sur les applications de type gros traitement du signal, consiste à définir les ensembles de données distantes, locales et dans le cache sous forme de systèmes linéaires et à résoudre ces systèmes pour en déduire les codes de communication, de transfers mémoire, d'accès à la mémoire cache programmable et de calcul tout en pipelinant accès mémoire et calculs.

1 Introduction

Les améliorations classiques du modèle de style Von NEUMANN consistent à essayer d'augmenter le débit de calcul, le débit mémoire (la « quantité ») et enfin le débit d'addresse mémoire (la « qualité »), point souvent oublié. La solution actuelle classique au problème de débit d'adresse est basée sur l'usage de mémoire cache. Malheureusement, de nombreuses applications sont handicapées par la nécessité pratique d'accéder à la mémoire avec une certaine cohérence spatio-temporelle. D'autres, telles que les méthodes itératives de type échiquier, gaspillent de la bande passante mémoire en ne lisant qu'un élément sur deux et en n'écrivant aussi qu'un élément sur deux.

Dans le cas de la machine PHÉNIX en cours de développement l'ETCA [1] représentée sur la figure 1, le débit de calcul est atteint par du parallélisme MIMD à mémoire distribuée à partir de processeurs du commerce (PE avec de la mémoire scalaire MS). Afin d'améliorer le débit d'adresse au niveau de la mémoire, la mémoire de calcul de chaque processeur élémentaire est banquée (les bancs M) et afin de palier au faible débit d'adressage du processeur,

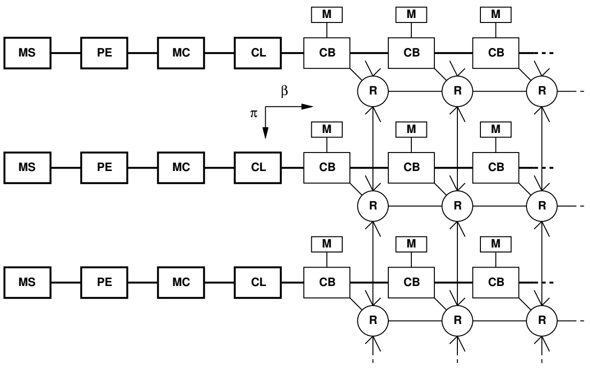


FIG. 1 – *Synoptique de la machine* PHÉNIX.

un processeur d'adressage (contrôleur de banc CB) est rajouté au niveau de chaque banc mémoire M et permet d'exécuter du code localement. Un contrôleur de ligne CL permet de faire des échanges entre les bancs mémoire et la mémoire cache programmable MC. Dans la suite, nous considérons deux types de CL, un simple avec des possibilités d'adressages minimales, un plus complexe cabable d'écrire ou de lire n'importe où dans la MC. Enfin, pour compenser le caractère très distribué de la mémoire, les bancs sont reliés entre eux par un réseau de type grilles par des contrôleurs de réseau R et la machine complète contient plusieurs plans de ce type interconnectés par des réseaux de type Omega en parallèle (non représentés ici) [Fio94].

Plutôt que de définir un nouveau langage, nous préférons partir du langage parallèle classique HPF qui permet bien d'exprimer des applications de type traitement du signal et, en nous basant sur l'expérience acquise [ACIK93, ACIK95], de compiler ce langage pour PHÉNIX. Par rapport aux machines classiques, une dimension de parallélisme est ajoutée : celle des bancs mémoire et du réseau. En adaptant le modèle d'exécution d'HPF qui a la caractéristique de n'être presque pas spécifié [For93, pages 2,3, 10–19] on peut choisir qu'une dimension de processeurs HPF est en fait une dimension de processeurs virtuels dont la mémoire est représentée par chaque banc mémoire.

2 Modèle d'exécution

Nous nous intéressons pour l'instant aux nids de boucles parfaitement imbriquées avec des bornes de boucles et des fonctions d'accès aux éléments de

*CRI-ENSMP, 35, rue Saint-Honoré,
77305 FONTAINEBLEAU CEDEX,
{irigoin,keryell}@cri.ensmp.fr,
http://www.cri.ensmp.fr/pips.
†DSIT/DAPSA/PTO, Cité de l'Air, 2bis, rue Lucien Boussoutrot, 00460 ARMÉE.

1. Établissement Technique Central de l'Armement.

```
!HPF$INDEPENDENT(i)
     forall(Li ≤ b0(n))
          X(SXi + aX0(n)) = f(Y(SYi + aY0(n)),
                              Z(SZi + aZ0(n)),
                              ...)
```

FIG. 2 – *Boucle générique.*

tableaux de type `!HPF$independent` (c'est-à-dire sans dépendances entre itérations) car

- ils permettent une compilation plus simple ;
- ils représentent souvent une grande partie du parallélisme disponibles dans les programmes qui nous intéressent ;
- souvent on peut se ramener à ce cas par des transformations de programme.

D'autre part, on utilise la « *owner-compute rule* » pour distribuer les calculs, sachant qu'on peut simplement choisir une autre règle (voir l'introduction de [ACIK95]).

Le concept de bancs mémoire n'existe pas en HPF mais il est intéressant de contrôler leur utilisation de la même manière que les processeurs d'HPF puisque c'est un problème clé pour éviter les conflicts dans les mémoires banquées. Plutôt que d'introduire une nouvelle directive de distribution, on décide qu'une dimension de processeur correspond en fait à la dimension des B bancs mémoire. Cette dimension peut aussi être vue microscopiquement comme une dimension de processeurs virtuels.

Les fonctionnalités offertes par cette équivalence permettent d'exprimer des distributions de données sous forme cyclique ou sous forme de bloc via la directive `!HPF$distribute`, des alignements de variables ou encore des répartitions de tableaux décalées pour éviter des conflicts sur le bus de données via la directive `!HPF$align`. Notons qu'on peut répliquer un tableau sur tous les bancs, ce qui peut être utile lorsqu'on fait certaines lectures avec des motifs irréguliers.

Considérons le nid de boucle générique de la figure 2 à exécuter sur PHÉNIX.

Le modèle d'exécution choisi suit de près l'architecture matérielle et étend le modèle de [ACIK95] en introduisant la notion de banc mémoire β et de processeur élémentaire de ligne π et les projections (cannoniques pour simplifier) $\beta = \boldsymbol{\beta}(p)$ et $\pi = \boldsymbol{\pi}(p)$.

Les directives HPF sont modélisées par le système d'équation 1 [ACIK95] et permettent par exemple de représenter figure 5(a) le « *template* » `T` et le tableau `A` de l'exemple figure 4 (les ronds représentent les éléments de tableau et les ronds noirs ou grisés les éléments accédés).

$$\left.\begin{array}{rr} \text{Alignement} & Rt = Aa + s_0 \\ \text{Distribution} & \Pi t = CPc + Cp + \ell \\ \text{Référence affine} & a = Si + a_0(n) \end{array}\right\} \quad (1)$$

Tout tableau `U` distribué au sens HPF se retrouve sous la forme d'un tableau local `U'` sur tous les bancs mémoire mais aussi sous forme d'un tableau `U''` dans la mémoire cache de chaque processeur élémentaire de ligne. Le modèle d'exécution consiste pour chaque itération i sur un processeur virtuel (c'est-à-dire en fait un banc mémoire) p :

1° à envoyer les données nécessaires $\mathit{Send}_{\mathtt{U}}(p, p')$ aux autres processeurs p' pour chaque tableau `U` ;

2° à recevoir sur chaque processeur les données nécessaires pour le calcul local $\mathit{Receive}_{\mathtt{U}}(p, p')$;

3° à envoyer les données $\mathit{View}_{\mathtt{U}}(p)$ nécessaires au calcul pour chaque ligne π à la mémoire cache ;

4° à recevoir dans la mémoire cache les données des bancs mémoires $\mathit{Fill}_{\mathtt{U}}(\pi)$;

5° à exécuter les calculs $\mathit{Compute}(\pi)$ sur le processeur physique π ;

6° à envoyer les données $\mathit{Empty}_{\mathtt{U}}(\pi)$ de la mémoire cache vers les bancs mémoire ;

7° à écrire les données $\mathit{Write}_{\mathtt{X}}(p)$ dans les bancs.

Il est plus formellement représenté sur la figure 3.

3 Modèle de compilation

Le modèle d'exécution laisse de nombreux choix dans l'énumération des ensembles et de nombreuses optimisations sont possibles. Certains choix et certaines optimisations sont exposés ici.

3.1 Allocation du cache à partir des bancs mémoire

Les articles [ACIK93, ACIK95] présentent des mécanismes d'allocation des tableaux et des variables temporaires qu'il faut étendre ici au cache. La figure 7 résume le procédé : on part de l'allocation bidimensionnelle dans la base (c, ℓ) d'un tableau `X`, on extrait via une transformation de HERMITE une base (u_1, u_2) du cristal HPF et on termine par une division entière pour éliminer encore des trous dans la base (u'_1, u'_2) si on est prêt à payer un surcoût d'adressage pour économiser de l'espace mémoire, ce qui revient par exemple à allouer les colonnes compactées sur les figures 7 et 6 pour le résultat global.

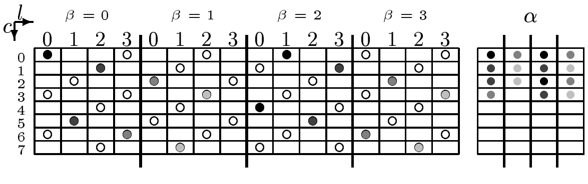
Il s'agit ici de trouver en plus une bijection entre les ensembles *View* et *Fill* (les données nécessaires à

```
array X'((c,ℓ) ∈ Own_X(p)),
      Y'((c,ℓ) ∈ View_Y(p)),
      Z'((c,ℓ) ∈ View_Z(p))
array X''(α ∈ Empty_X(p)),
      Y''(α ∈ Fill_Y(p)),
      Z''(α ∈ Fill_Z(p))
forall(U ∈ {Y',Z',...})                                  1°
      forall((p,p'), p ≠ p', Send_U(p,p') ≠ ∅)
            forall((ℓ,c) ∈ Send_U(p,p'))
                  send(p', U(ℓ,c))
forall(U ∈ {Y',Z',...})                                  2°
      forall((p,p'), p ≠ p', Receive_U(p,p') ≠ ∅)
            forall((ℓ,c) ∈ Receive_U(p,p'))
                  U(ℓ,c) = receive(p')
forall(U ∈ {Y',Z',...})                                  3°
      forall((ℓ,c) ∈ View_U(p))
            send_to_cache(U(ℓ,c))
forall(U ∈ {Y'',Z'',...})                                4°
      forall(α ∈ Fill_U(π))
            receive_from_memory(U(α))
forall(α ∈ Compute(π))                                   5°
      X''(S_X''(α)) = f(Y''(S_Y''(α)),
                        Z''(S_Z''(α)),...)
forall(α ∈ Empty_X(π))                                   6°
      send_to_memory(X''(α))
forall((ℓ,c) ∈ Write_X(p))                               7°
      receive_from_cache(X'(ℓ,c))
```

FIG. 3 – *Modèle d'exécution de la boucle générique figure 2.*

```
      real A(0:42), B(0:42)
!HPF$ template T(0:127)
!HPF$ processors P(0:3)
!HPF$ align A(i) with T(3*i)
!HPF$ align B(i) with A(i)
!HPF$ distribute T(cyclic(4)) onto P
      A(0:U:3) = A(0:U:3) + B(1:U+1:3)
```

FIG. 4 – *Exemple inspiré de* CHATTERJEE et al. *[CGL+ 95].*



(a) Bancs mémoire (b) Cache vectoriel

FIG. 5 – *Exemple de distribution et d'écriture correspondant à la figure 4.*

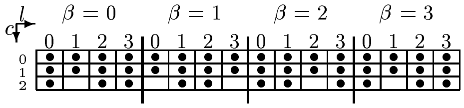


FIG. 6 – *Compactage dans les bancs mémoire et éventuellement dans le cache.*

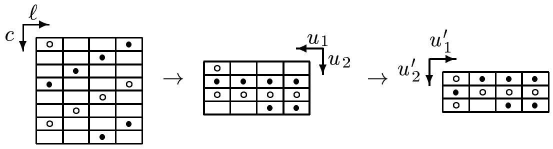


FIG. 7 – *Compactage d'une variable dans un banc mémoire.*

un calcul qu'il faut transférer) d'une part et *Empty* et *Write* (les résultats à réécrire dans X') d'autre part. Cela revient à choisir une allocation des données dans le cache pour les données lues et les résultats. Comme on ne veut réécrire en mémoire que le strict nécessaire du calcul, *Empty* = *Compute*.

3.1.1 Contrôleur de ligne simple

Dans un premier cas, nous considérons que la seule chose que sait faire le CL est de lire ou d'écrire le/dans le cache vers/depuis le bus mémoire pipeliné, avec un entrelacement des bancs mémoires (CB+M). Le cache est alors vu comme une série de registres vectoriels remplis par des processus *Fill* et vidés par des processus *Empty*. En énumérant les *View* et le *Write* dans la même base, α n'est que le numéro d'ordre de l'élément transféré et le PE se comporte comme un coprocesseur vectoriel avec une seule boucle interne sur α.

Une autre possibilité est d'allouer dans le cache de manière CYCLIC(1) ce qui vient de la mémoire et avoir l'allocation de la figure 5(b). Cela permet de différencier facilement ce qui vient de chaque banc. Matériellement, il suffit d'écrire une valeur quelconque dans le cache lorsqu'un banc mémoire n'a plus rien à écrire.

Un problème survient lorsqu'on a besoin de trouver (u'_1, u'_2) en fonction de α, par exemple pour gérer les « *overlaps* » comme dans :

```
A(0:U - 1) = A(0:U - 1) + A(1:U)
```

où il faut éviter de transférer deux parties de A presque identiques. Cela veut dire qu'il faut retrouver rapidement l'élément A(I+1) en fonction de l'élément A(I), ce qui n'est pas le cas dans l'avant-dernière allocation et un peu moins dans la dernière.

Pour éviter ce problème il faut que l'allocation reste proche de celle de la figure 6 car dans ce cas A(I) et A(I+1) sont séparés systématiquement de 3 éléments en mémoire (sauf sur le bord droit). Une allocation identique à une transposition près dans le cache est possible simplement en transmettant toute la variable, y compris ses « trous ». Cela revient à avoir la bijection $\alpha = U_1' B u_2' + B u_1' + \beta$, avec U_1' le nombre d'éléments suivant l'axe u_1' et le code est composé d'une itération dans (u_2, u_1, β) pour optimiser le cache primaire interne au PE. Pour des raisons d'efficacité, on applique un épluchage de boucle (« *loop peeling* ») d'une itération sur β et dans ce cas de 3 sur u_1, gérant ainsi correctement le cycle HPF.

3.1.2 Contrôleur de ligne programmable

Dans le cas de gestion d'*overlap* on peut être amené à transférer plus d'information que nécessaire. Pour se limiter à ce qui est nécessaire il faut rajouter d'autres fonctions d'adressage au CL afin qu'il soit capable d'énumérer *Fill* et *Empty* ou encore suivre des adresses envoyées en plus des données par les CB.

3.2 Diminution des communications

Les points 1° et 2° du schéma de compilation de la figure 3 montre que si $p \neq p'$ mais que $\pi = \pi'$, il y a une communication qui est effectuée alors qu'elle pourrait être remplacée par un placement au vol dans le cache. Ceci n'est possible que dans le cas d'un CL programmable.

De même, on peut supprimer les composantes horizontales des communications dans le réseau de la figure 1 en les remplaçant par l'adressage adéquat au niveau du CL et éventuellement ne pas réaliser ce réseau dans la machine PHÉNIX.

3.3 Pipeline de l'usage du cache

Si toutes les données logeaient dans le cache programmable, l'architecture n'aurait que peu d'intérêt et on se ramènerait directement à [ACIK95]. Hors, le modèle de la figure 3 implique une taille de cache de même taille que les variables. D'autre part, la séquence des phases 5° , 6° et 7° implique qu'un calcul ne peut commencer avant que tout le cache soit chargé car il n'y a pas de mécanisme matériel permettant d'assurer qu'une donnée est présente dans le cache.

Pour contourner ces deux problèmes, ces phases sont découpées en phases plus petites et pipelinées. Si on prend une allocation du type de la figure 5, un pipeline suivant l'axe des c fournit un bon facteur de forme[2]. Il suffit de rajouter les nouvelles variables c' et c'' avec $c = \Gamma c' + c''$ et $0 \leq c'' < \Gamma$ pour avoir la boucle de pipeline en c'. Γ est choisi en fonction de la majoration du volume des variables nécessaires par bloc de pipeline et de la taille de cache disponible. Le pipeline est ensuite fait en 2 phases travaillant sur 2 zones de caches échangeant leur rôle tour à tour : une phase vidant le cache puis le remplissant tandis que l'autre phase calcule sur la zone de cache précédemment remplie.

2. Tout vecteur dans l'espace d'itération peut convenir puisqu'il n'y a pas de dépendances entre les itérations.

4 Conclusion

Notre modèle de compilation permet d'étendre les distributions du langage à parallélisme de données standard HPF à des machines à mémoire banquée sans aucune modification du langage en unifiant la notion de processeur HPF et de banc mémoire. La génération de code pour les différents sous-systèmes d'une machine telle que PHÉNIX est automatisée en utilisant un formalisme utilisant l'algèbre linéaire pour compiler des nids de boucles à bornes affines. Enfin, en fonction de la complexité de la machine, différentes optimisations sont possibles et une solution permettant de pipeliner calculs et chargement mémoire est proposée.

5 Remerciements

Nous tenons à remercier Sylvain CARTIER et Philippe MICHALLON de l'équipe PHÉNIX et Corinne ANCOURT et Fabien COELHO du projet PIPS pour leur discussions sur le sujet.

Références

[ACIK93] Corinne ANCOURT, Fabien COELHO, François IRIGOIN, et Ronan KERYELL. « A Linear Algebra Framework for Static HPF Code Distribution ». Dans *CPC'93: Fourth Workshop on Compilers for Parallel Computers*, Delft, Hollande, décembre 1993.

[ACIK95] Corinne ANCOURT, Fabien COELHO, François IRIGOIN, et Ronan KERYELL. « A Linear Algebra Framework for Static HPF Code Distribution ». Rapport Technique A-278-CRI, Centre de Recherche en Informatique, École Nationale Supérieure des Mines de Paris, France, novembre 1995. Soumis à « *Scientific Programming* ». `http://www.cri.ensmp.fr/doc/A-278.ps.gz`.

[CGL+95] Siddhartha CHATTERJEE, John R. GILBERT, Fred J. E. LONG, Robert SCHREIBER, et Shang-Hua TENG. « Generating Local Addresses and Communication Sets for Data-Parallel Programs ». *Journal of Parallel and Distributed Computing*, 26(1):72–84, avril 1995.

[Fio94] Pierre FIORINI. « *Réseaux d'Interconnexion pour les Architectures Parallèles. Les Delta-Pyramides et le Projet Phénix* ». Thèse, Université de Paris Sud, décembre 1994.

[For93] High Performance Fortran FORUM. « High Performance Fortran Language Specification ». Version 1.0 CRPC-TR 92225, Center for Research on Parallel Computation, Rice University, Houston, USA, mai 1993. Récupérable par `ftp anonymous` sur la machine `titan.cs.rice.edu` dans le fichier `public/HPFF/draft/hpf-v10-final.ps.Z`.