Exécution de codes irréguliers par migration de tâches

Yvon Jégou

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Exécution de codes irréguliers par migration de tâches

Yvon Jégou

N° 2436

Novembre 1994

PROGRAMME 1

ISSN 0249-6399

INRIA
RENNES

Exécution de codes irréguliers par migration de tâches

Yvon Jégou*

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet CAPS

Rapport de recherche n° 2436 — Novembre 1994 — 56 pages

Résumé : L'exécution de codes irréguliers sur une architecture parallèle à mémoire distribuée est un problème difficile. Plusieurs solutions sont étudiées actuellement. Certaines de ces solutions se basent sur la distribution des données sur les mémoires, et les compilateurs ont à leur charge la génération des ordres de communication. D'autres solutions se basent sur l'existence d'un espace d'adressage partagé par l'ensemble des processeurs, ce qui réduit la charge des compilateurs à l'optimisation des mouvements de pages mémoire et à la gestion de la cohérence des pages recopiées. Dans ce document, nous traitons de l'utilisation des tâches migrantes pour effectuer des calculs sur des données distribuées. Avec ce modèle, l'exécution de chaque itération d'une boucle parallèle est interprétée par une tâche. Lorsqu'une tâche tente d'accéder à une donnée non locale, elle migre sur le processeur qui possède cette donnée, et continue son exécution sur ce processeur. Les données parallèles distribuées ne sont jamais déplacées dans notre modèle. Les tâches migrantes sont produites automatiquement par la compilation de langages classiques comme Fortran. Des expérimentations montrent que l'exécution de codes irréguliers peut être accélérée par ce système, même dans le cas de mauvaise localité des données.

Mots-clé : mémoires distribuées, migration de tâches, codes irréguliers

(Abstract: pto)

*. jegou@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Migrating tasks for irregular code execution

Abstract: Executing irregular codes on distributed memory multiprocessors is a difficult problem. A few solutions have been proposed so far. Some of these solutions are based on data distribution techniques where the compilers are in charge of the generation of communication between the parallel computations. Others solutions implement a shared address space on the parallel processors where the compile time optimizations are concerned with the reduction of virtual memory paging and with coherency. In this paper, we introduce the migrating task model. In this model, the execution of each iteration of a parallel loop generates a task. When a task needs to access a non local data, it migrates on the processing element which owns this data, its execution goes on there. In our model, the parallel data are never moved. The migrating tasks are produced automatically by the compilation process of classical languages. Our experiments show that good performance on irregular codes can be obtained on a distributed memory computer even when the application exhibits poor memory locality.

Key-words: distributed memory, task migration, irregular codes

Table des matières

1 Introduction

Les architectures massivement parallèles à mémoires distribuées offrent un potentiel de performance élevé. Cependant les performances réelles atteintes de nos jours sont limitées par l'absence d'environnement de programmation efficaces pour porter les applications. Pour être viables, les systèmes à mémoires distribuées doivent offrir aux utilisateurs un modèle de programmation classique :

- modèle de programmation simple,
- langages de haut niveau,
- possibilité de débogage,
- réutilisation de codes,
- portabilité des applications.

Plusieurs projets, comme Fortran-D [1, 2], Vienna Fortran [3] ou Pandore [4] ont comme objectif la mise en œuvre de compilateurs qui offrent un style de programmation parallèle au travers de spécifications des distributions des données et au travers d'instructions parallèles. D'autres projets [5, 6] proposent des techniques de compilation plus classiques par l'utilisation d'une mémoire virtuelle partagée à l'exécution. Dans tous ces projets, le déplacement des données constitue le frein aux performances : chaque accès à une donnée non locale entraîne des échanges de messages, des synchronisations de processeurs ou de processus, des migrations de données.

Dans notre approche, les données parallèles sont réparties sur l'ensemble des processeurs et ne sont plus jamais déplacées par la suite. Ce sont les tâches qui accèdent à ces données qui doivent migrer pour accéder aux données. Lors de l'exécution d'une boucle parallèle, chaque itération de la boucle devient une tâche migrante. Une tâche n'est pas liée au processeur qui la crée : elle doit migrer pour *rencontrer ses données*. Ce système est complètement asynchrone. Un processeur n'attend jamais de réponse à un message qu'il émet. L'exécution d'une construction parallèle entraîne la création de tâches parallèles. L'exécution de la construction est terminée lorsque toutes les tâches crées pour son exécution sont terminées. Dans une certaine mesure, la technique des tâches migrantes peut être comparée à certaines techniques d'exécution de langages data-flot également basées sur l'exécution de nombreuses tâches parallèles [7].

La première partie de ce document présente le modèle des tâches migrantes. Pour être réaliste, tout modèle d'exécution parallèle doit pouvoir être utilisé de manière automatique. L'algorithme de génération de code associé au modèle des tâches migrantes est détaillé dans le paragraphe 3. La nature asynchrone de notre modèle pose un certain nombre de problèmes classiques du calcul distribué. C'est le cas de la détection de la terminaison des tâches d'une construction parallèle qui est traité dans le paragraphe 4. La mise en œuvre du modèle des tâches migrantes sur une architecture réelle se base non seulement sur les primitives de communication de cette architecture mais également sur les possibilités d'exécution de plusieurs

flots d'instructions en parallèle. Le paragraphe 5.5 traite du problème de l'adaption du modèle aux possibilités du système visé. Nous avons expérimenté ce modèle dans l'exécution d'un code irrégulier sur l'Intel Paragon X/PS. Les résultats de cette expérimentation sont exposés au paragraphe 6.

2 Les tâches migrantes

Le support d'exécution des tâches migrantes est un modèle d'éxécution SPMD dans lequel l'ensemble des processeurs exécute et échange des tâches. Dans ce modèle, l'exécution d'une boucle parallèle se traduit par la création et l'exécution d'un ensemble de tâches. En général, chacune de ces tâches est chargée de l'exécution d'une itération de la boucle. Une tâche présente sur un processeur est exécutée jusqu'à ce qu'elle soit bloquée par une demande d'accès à une mémoire non locale. Cette tâche migre alors sur le processeur qui possède cette adresse, et y reprend son exécution. Les données distribuées ne sont jamais recopiées, ce qui limite les problèmes de cohérence des données. Les boucles considérées étant parallèles, aucune synchronisation n'est prévue entre les tâches. Contrairement à ce qui se passe avec certains modèles, la présence de mises à jour atomiques sur des variables (mise à jour d'une variable par une valeur dépendant de la valeur précédente de la variable) n'interdit pas la parallélisation d'une boucle.

2.1 Le modèle d'exécution SPMD

Le modèle d'exécution SPMD duplique le code à exécuter sur l'ensemble des processeurs. Lorsque ce code est issu d'un programme séquentiel, ce programme est exécuté sur chaque processeur. Pour ce faire, on distingue trois catégories de données dans les applications : les données distribuées, les données dupliquées et les données locales.

Une donnée distribuée du programme est associée à un seul des processeurs dans le modèle SPMD. Les données distribuées sont en général des tableaux et leurs éléments sont répartis sur l'ensemble des processeurs. Les fonctions de distribution sont globales : chaque processeur doit pouvoir déterminer localement quel est le propriétaire d'un élément. Les fonctions de distribution peuvent être quelconques et ne sont pas limitées aux rangements cycliques ou par blocs. Même lorsque seules des fonctions de répartition simples sont utilisées dans l'allocation des mémoires, l'accès à des portions de tableaux passés en paramètre à des sous-programmes peut être complexe. Mais cette fonction reste évaluable localement aux processeurs et le modèle d'exécution par migration de tâche peut être utilisé.

Une donnée dupliquée est présente sur chaque processeur d'un programme SPMD. C'est en général une donnée scalaire. Sa valeur est la même sur tous les processeurs. Une donnée dupliquée est toujours accessible par une tâche, et ne dépend pas du processeur qui l'exécute.

Les variables locales n'ont de signification que dans la tâche qui les accèdent. Ce sont des variables temporaires. Elles migrent avec les tâches.

2.2 Les tâches migrantes

Dans le modèle d'exécution par tâches migrantes, chaque tâche représente une partie de l'exécution du programme. Des tâches parallèles concernent l'exécution de sections parallèles de code. En général, une tâche correspond à une itération d'une boucle parallèle. Mais des tâches migrantes peuvent également être créées dans l'exécution de sections séquentielles. Ce cas se produit lorsqu'une variable distribuée est accédée dans une portion non parallèle d'un programme. Les entrées/sorties sur des variables distribuées entraînent également la création de tâches migrantes : pour respecter la sémantique des fichiers séquentiels, les instructions de lecture et d'écriture sur un fichier sont exécutées sur un seul processeur. Un fichier est considéré comme un scalaire distribué.

Les tâches migrantes ne se synchronisent ni ne communiquent jamais : elle sont totalement indépendantes. En fait, ce n'est pas une restriction de modèle. Cette indépendance provient des techniques de parallélisation utilisées par les outils automatiques. On peut imaginer d'introduire des capacités de communication dans le modèle pour l'exécution de boucles partiellement indépendantes.

Une tâche migrante est un automate et une mémoire contenant les données locales de l'automate. Le code d'exécution de l'automate est dupliqué par le modèle SPMD, et ne migre pas avec les tâches : seuls l'état courant et les données locales sont transférés. L'état courant de l'automate est le point d'entrée du code où l'exécution est reprise après la migration. Soit la boucle parallèle Fortran suivante.

```
      dopar i= 1, n
        a(i)= b(i)+c(i)
      end do
```

Les tableaux `a`, `b` et `c` de cet exemple sont des variables distribuées, `n` est dupliquée et `i` est une donnée locale à l'itération. Ce corps de boucle peut être interprété par le code suivant.

```
      if(EstNonLocal(b(i))) MigrerSur(Proprietaire(b(i)), L_bi, i);
L_bi: bi= b(i);
      if(EstNonLocal(c(i))) MigrerSur(Proprietaire(c(i)), L_ci, i, bi);
L_ci: ci= c(i);
      bici= bi+ci;
      if(EstNonLocal(a(i))) MigrerSur(Proprietaire(a(i)), L_ai, i, bici);
L_ai: a(i)= bici;
```

Dans ce code, les variables `i`, `bi`, `ci` et `bici` sont locales à la tâche migrante. La fonction `EstNonLocal(x)` permet de tester si une variable `x` est locale au processeur courant. La fonction `Proprietaire(x)` fournit l'identification du processeur propriétaire de la variable `x`. Enfin, la routine `MigrerSur(dest, état, contexte)` transfère la tâche courante sur le processeur `dest`. Après ce transfert, l'exécution est reprise au point d'entrée `état`. `contexte` est une liste de variables locales de la tâche. Seules les variables locales vivantes

sont transférées. La routine `MigrerSur` n'est pas une fonction : après l'appel à cette routine, le processeur courant transfère le contrôle à une autre tâche.

2.3 Optimisations de base des automates

Pour diminuer le nombre de migrations potentielles d'une tâche ainsi que le nombre de tests de localité, deux types d'optimisations peuvent être systématiquement appliqués aux automates.

Réduction du nombre de migrations

Lorsque plusieurs données distribuées doivent être accédées dans une tâche, l'analyse des dépendance permet une certaine liberté dans l'ordre des accès à ces données. La première optimisation consiste à accéder à toutes les données distribuées testées locales au processeur courant avant d'effectuer une migration. Dans l'exemple précédent, il se peut que l'élément `b(i)` ne soit pas local au processeur courant alors que `c(i)` l'est. Lorsque ce cas se produit, `c(i)` doit être lue et rangée dans une variable locale avant la migration. Cette forme d'optimisation se traduit par une multiplication du nombre d'états de l'automate. Le corps de l'automate précédent devient :

```
        if(EstNonLocal(b(i))) goto L_b;
L_b_B:  bi= b(i);
        if(EstNonLocal(c(i))) MigrerSur(Propriétaire(c(i)), L_c_BC, i, bi);
L_c_BC: ci= c(i);
L_BC:   bici= bi+ci;
        if(EstNonLocal(a(i))) MigrerSur(Propriétaire(a(i)), L_a_A, i, bici);
L_a_A:  a(i)= bici;
        exit;
L_b:    if(EstNonLocal(c(i))) MigrerSur(Propriétaire(b(i)), L_b_B, i);
        ci= c(i);
        MigrerSur(Propriétaire(b(i)), L_b_BC, i, ci);
L_b_BC: bi= b(i);
        goto L_BC;
```

Réduction du nombre de tests de localité

Une deuxième optimisation consiste à utiliser des relations connues entre les fonctions de distribution des données pour diminuer le nombre de tests de localité. Les fonctions de distribution ne sont en général pas quelconques. Elles sont souvent choisies par le compilateur et les programmes peuvent contenir des spécifications de rangement ou d'alignement des données. Par exemple, dans le code précédent, le fait de savoir que les vecteurs `b` et `c` ont la même fonction de distribution permet d'éliminer les tests de localité sur l'un des accès. Une forme plus spécifique de ce type d'optimisation consiste à choisir la fonction de distribution des itérations d'une boucle pour éviter la première migration.

2.4 Comportement attendu des tâches migrantes

Dans les techniques classiques d'exploitation des architectures à mémoires distribuées, l'accès à une donnée non locale entraîne des échanges de messages entre plusieurs processeurs et leur synchronisation. Cet échange nécessite une synchronisation de ces requêtes : le processus qui émet une requête d'accès distant doit attendre la réponse. Sur un modèle monoprocessus, cette attente bloque tout calcul. Sur un modèle multiprocessus, le processus est désactivé pour libérer les ressources de calcul. Mais il faut alors mettre en place une structure de données permettant de le réveiller lorsque la réponse à sa requête est présente. Cette structure peut être relativement complexe lorsque plusieurs requêtes distantes sont actives simultanément et que les réponses peuvent revenir dans le désordre.

Dans notre modèle, une migration de tâches ne provoque pas d'émission de message en retour. Le système est entièrement asynchrone. Un processeur n'attend jamais de réponse à une requête. Cet asynchronisme doit permettre un recouvrement des communications et du calcul.

3 Génération automatique de codes de tâches migrantes

La transformation automatique de programmes Fortran classiques vers un modèle à tâches migrantes peut être entièrement automatisée. L'efficacité des programmes qui résultent dépend avant tout du parallélisme potentiel de ces codes, condition nécessaire pour produire de l'activité simultanément sur plusieurs processeurs. L'algorithme de génération de code que nous proposons se base sur des informations présentes dans les compilateurs/optimiseurs : ce sont le graphe de flot/dépendance et les informations sur les alignements des tableaux dans les mémoires.

Les nœuds du graphe de flot sont restructurés de manière à contenir au plus un accès à une donnée distribuée. Durant la génération du code, les nœuds du graphe sont regroupés en nœuds locaux à la tâche courante (ne contenant que des accès à des variables locales à la tâche), en nœuds distribués (contenant un accès à une donnée distribuée), en nœuds locaux au processeur courant (contenant un accès à une donnée distribuée testée locale), et en nœuds non locaux au processeur courant (le test de localité a échoué). Ces regroupements se font par l'analyse des accès mémoire des instructions de ces nœuds. Ces accès peuvent être des lectures, des écritures ou des mises à jour atomiques. La réduction des nœuds locaux (à la tâche ou au processeur) n'entraîne pas de migration et est toujours possible dès lors que les relations sur le flot des données et sur les dépendances sont respectées. Les nœuds non locaux contiennent un accès distribué pour lequel le test de localité a échoué. La réduction d'un nœud non local ne peut être réalisée qu'après migration. Les nœuds distribués contiennent un accès à une donnée distribuée dont l'état de localité n'est pas connu : ces nœuds doivent subir un test de localité avant toute réduction. La technique d'optimisation utilisée pour réduire le nombre de migrations consiste à ne pas générer de migration tant qu'il existe des nœuds distribués non testés dont la réduction n'est pas interdite par les relations de flot ou de dépendance. Tous les nœuds peuvent contenir des accès locaux à la tâche courante. La

figure 2 montre les différents changements de l'état des nœuds au cours des tests de localité ou des migrations.

3.1 Principes de la génération de tâches migrantes

L'algorithme de génération de code simple de la figure 3 cherche à réduire autant de nœuds que possible avant de provoquer une migration de la tâche. Tous les nœuds qui accèdent à des variables distribuées sont initialement dans le groupe distribué. Chaque exécution de cet algorithme produit, soit la réduction d'un nœud (génération du code associé à ce nœud), soit un test de localité sur un nœud distribué, soit une migration sur un nœud non local. Ce même algorithme est appliqué récursivement sur le graphe résultant de la transformation.

Dans les explications sur le génération de code de tâches migrantes, nous considérons le corps de la boucle Fortran

```
do i=1, n
  a(l(i))= a(l(i))+ v(i)*b(c(i))
end do
```

Le graphe de flot/dépendance correspondant au corps de cette boucle est donné par la figure 1. Dans ce graphe, les nœuds 0 et 5 sont locaux à la tâche, ils ne contiennent pas d'accès à des données distribuées. Les autres nœuds sont distribués.

codage d'état

Chaque application de la routine de génération de code commence par le codage et la mémorisation de l'état courant du graphe du programme. Cette phase permet d'éviter la génération de séquences de codes identiques.

réduction d'un nœud

La réduction d'un nœud du graphe consiste à générer le code associé à ce nœud. Si ce nœud correspond à une instruction structurée — boucle, conditionnelle — le code correspondant est produit par un appel récursif à l'algorithme de génération de code sur le graphe associé à ce nœud.

arc vivant

Un arc du graphe de flot/dépendance est dit vivant lorsque son nœud d'origine a été réduit mais pas son nœud de destination.

nœud réductible

Un nœud du graphe flot/dépendance est réductible lorsque tous ses arcs entrants sont vivants.

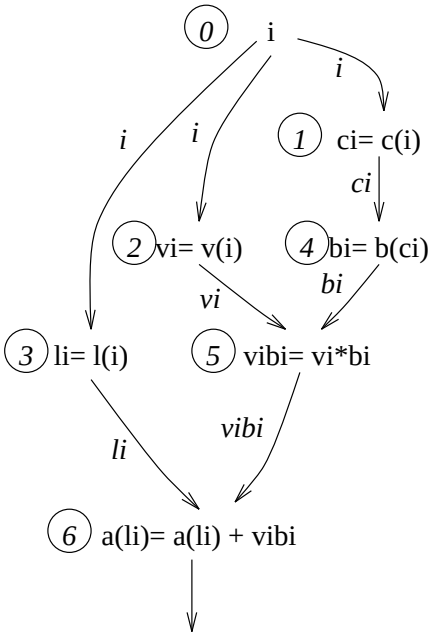

FIG. 1 - *graphe flot/dépendance*

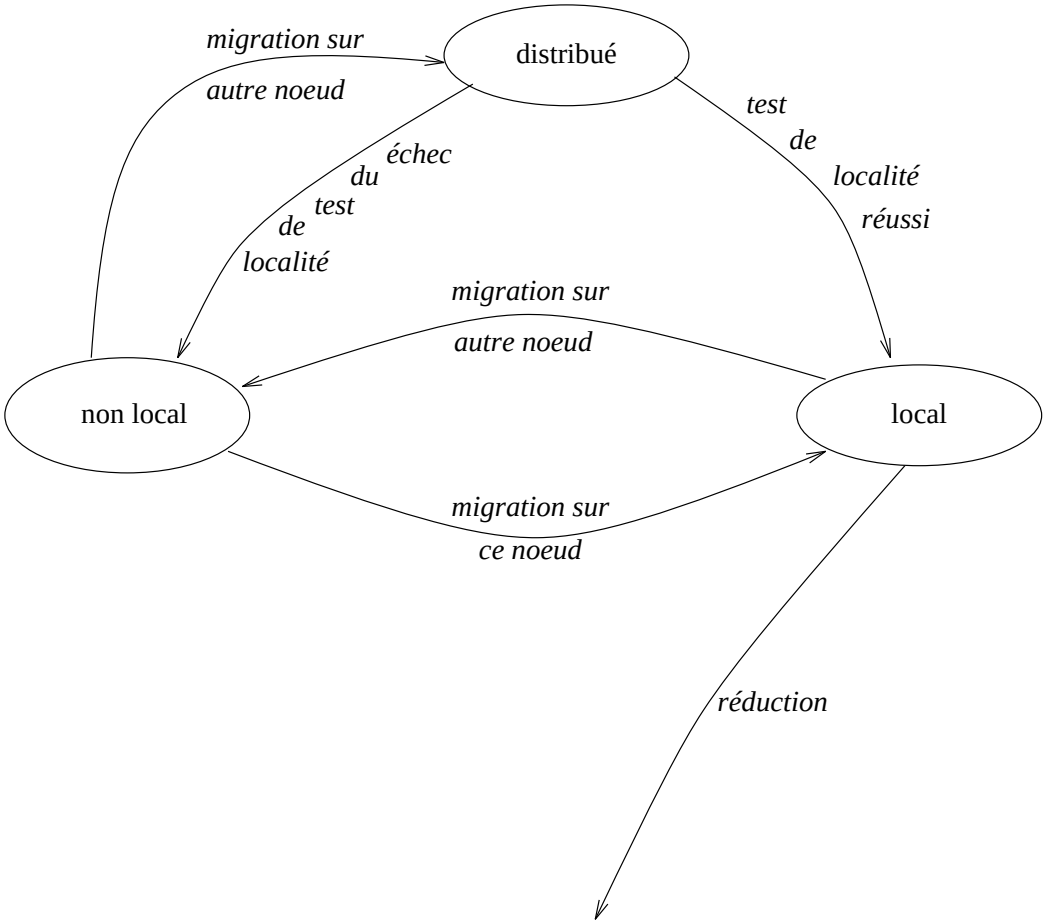

FIG. 2 - *transitions d'états des nœuds du graphe*

```
GénèreCodeTâche(VIVANT, TACHE, LOCAL, NLOCAL, DIST){
    Étiq ← ÉtiquetteÉtat (VIVANT, TACHE, LOCAL, NLOCAL, DIST);
    si (ÉtiquetteExiste(Etiq)) {
        GénèreSaut("goto Etiq;");
        Sortir;
    } sinon GénèreÉtiquette("Étiq:");

    calculer REDUC d'après VIVANT et IN.
    Si (REDUC ∩ TACHE ≠ ∅){
        choisir nœud n dans REDUC ∩ TASK
        GénèreCodeDe(n);
        GénèreCodeTâche (VIVANT − IN_n + OUT_n, TACHE − {n}, LOCAL, NLOCAL, DIST);
    }
    SinonSi (REDUC ∩ LOCAL ≠ ∅){
        choisir nœud n dans REDUC ∩ LOCAL
        GénèreCodeDe(n);
        GénèreCodeTâche (VIVANT − IN_n + OUT_n, TACHE, LOCAL − {n}, NLOCAL, DIST);
    }
    SinonSi (REDUC ∩ DIST) ≠ ∅){
        choisir nœud n dans DIST ∩ REDUC;
        Étiq ← ÉtiquetteÉtat (VIVANT, TACHE, LOCAL + {n}, NLOCAL, DIST − {n})
        GénèreSaut("if (EstLocal(n) goto Étiq;");
        GénèreCodeTâche (VIVANT, TACHE, LOCAL, NLOCAL ∪ {n}, DIST − {n});
        GénèreCodeTâche (VIVANT, TACHE, LOCAL ∪ {n}, NLOCAL, DIST − {n}) };
    }
    SinonSi (NLOCAL ≠ ∅){
        Étiq ← ÉtiquetteÉtat (VIVANT, TACHE, {n}, LOCAL, DIST ∪ NLOCAL);
        GénèreSaut("MigrerSur(Propriétaire(j), Étiq, (∀e ∈ VIVANT, VAL_e)");
        GénèreCodeTâche (VIVANT, TACHE, {n}, LOCAL, DIST);
    }
}
```

FIG. 3 - *Gen1 : algorithme simple de génération de code*

contexte de tâche

Le graphe de flot/dépendance considéré provient de la fusion du graphe de flot et du graphe de dépendance. Une (ou plusieurs) donnée est associée à chaque arc provenant du graphe de flot. Le contexte d'une tâche est constitué de l'ensemble des valeurs associées aux arcs vivants au moment de la migration.

Dans notre exemple, les arcs 0 : 1,0 : 2 et 0 : 3 portent la valeur i.

3.2 Algorithme de génération de code

L'algorithme de génération de code adopté part d'un graphe de flot/dépendance, génère soit une instruction, soit un test de localité, soit un ordre de migration, calcule le nouveau graphe résultant de cette génération et s'applique récursivement sur ce nouveau graphe (figure 3).

Génération d'une instruction

Cette transformation est appliquée lorsqu'un nœud local à la tâche ou un nœud local au processeur est réductible. Ce nœud est enlevé du graphe résultant (réduction), l'ensemble

des arcs vivant et les ensembles $TACHE$ et $LOCAL$ sont mis à jour. Le génération de code suivant est appliquée sur ce graphe résultant. Les paramètres de l'algorithme sont :

$VIVANT$ **:** ensemble des arcs vivants

$TACHE$ ensemble des nœuds non réduits locaux à la tâche

$LOCAL$ ensemble des nœuds non réduits locaux au processeur

$NLOCAL$ ensemble des nœuds non réduits non locaux au processeur

$DIST$ **:** ensemble des nœuds non réduits dont la localité n'a pas été testée

Génération d'un test de localité

La génération d'un test de localité entraîne le calcul de deux nouveaux environnements du graphe : un environnement correspondant au cas où le test réussi et un environnement correspondant au cas où il échoue. Le test de localité est toujours appliqué sur l'un des nœuds réductibles de l'ensemble $DIST$. Dans l'environnement de succès, ce nœud est placé dans l'ensemble $LOCAL$. Dans le cas d'échec, il est déplacé dans l'ensemble $NLOCAL$. La génération de code est appliquée successivement sur ces deux environnements. La génération d'un test de localité ne provoque pas de réduction sur le graphe. Il est en général possible de choisir le sens du test de localité (test si local ou test si non local) en analysant les états déjà rencontrés. Ce choix permet d'éviter de produire des instructions de saut derrière les tests de localité.

Génération d'un ordre de migration

La génération d'un ordre de migration s'applique à l'un des nœuds réductibles de l'ensemble $NLOCAL$. Le contexte de l'ordre de migration (les valeurs qui doivent migrer) est calculé d'après l'ensemble des arcs vivants. Dans l'environnement qui résulte d'une migration, le nœud sélectionné devient le composant unique de l'ensemble $LOCAL$, l'ensemble $NLOCAL$ étant constitué des nœuds qui appartenaient précédemment à l'ensemble $LOCAL$. Les nœuds qui composaient l'ensemble $NLOCAL$ sont remis dans l'ensemble $DIST$. Le code qui sera exécuté après cette migration est généré par application de l'algorithme sur ce nouvel environnement.

Génération de l'exemple

Dans notre exemple, seul le nœud 0 est réductible initialement. C'est le point d'entrée dans le graphe. Après sa réduction, les arcs $0:1, 0:2$ et $0:3$ sont vivants et les nœuds 1, 2 et 3 deviennent réductibles. Mais ces trois nœuds sont dans l'état distribué (ensemble $DIST$, ligne `e1`, figure 4).

$$(VIVANT = \{0:1, 0:2, 0:3\}, TACHE = \{5\}, LOCAL = \emptyset, NLOCAL = \emptyset, DIST = \{1, 2, 3, 4, 6\})$$

Aucun nœud local n'étant réductible, un test de localité est produit sur l'un des nœuds réductibles de l'ensemble $DIST$

```
      if(EstNonLocal(Var)) goto cas_non_local;
        ... code pour le cas ou Var est local
cas_non_local:
        ... code pour le cas ou Var n'est pas local
```

Un test de localité est accompagné de deux appels récursifs à l'algorithme de génération de code. Supposons le choix du nœud 1 pour le test de localité. Dans le cas local (étiqueté `e3`, figure 4), l'environnement de génération est :

$$(VIVANT = \{0:1, 0:2, 0:3\}, TACHE = \{5\}, LOCAL = \{1\}, NLOCAL = \emptyset, DIST = \{2,3,4,6\})$$

Dans le cas non local(étiqueté `e2`, figure 4), cet environnement est :

$$(VIVANT = \{0:1, 0:2, 0:3\}, TACHE = \{5\}, LOCAL = \emptyset, NLOCAL = \{1\}, DIST = \{2,3,4,6\})$$

Au cours de l'appel récursif de traitement local, le nœud 1 est dans l'état réductible local, ce qui entraîne sa réduction immédiate. Dans notre exemple, après la réduction de ce nœud, les nœuds 2, 3 et 4 sont réductibles distribués et l'un de ces nœuds est choisi pour un nouveau test de localité et ainsi de suite.

Un ordre de migration est produit lorsque les seuls nœuds réductibles sont dans l'ensemble $NLOCAL$. C'est ce qui se produit pour notre exemple dans les états `e15`, `e11`, `e21`... des figures 4 et 5.

Un ordre de migration contient le destinataire de la tâche, son état et son contexte. L'état de la tâche est une adresse dans le code où son exécution sera reprise. Le code correspondant à cette adresse est produit par l'appel récursif qui suit la production de l'ordre de migration. Le contexte de la tâche est constitué de l'ensemble des valeurs portées par les arcs vivants au moment de la migration.

Dans notre exemple, à la ligne `e18`, l'algorithme de génération se retrouve dans la situation où les seuls nœuds 3 et 4 sont réductibles. Mais ces nœuds appartiennent à l'ensemble $NLOCAL$. En ce point, le contexte est défini par les arcs vivants $0:3, 1:4, 2:5$, ce qui correspond aux valeurs i, ci et vi. L'un des nœuds réductibles, le nœud 2 dans notre exemple, est choisi comme cible pour une migration de la tâche avec les valeur i, ci et vi, et une reprise de l'exécution à l'étiquette `e9`. En `e9`, le nœud choisi 3 est placé dans l'ensemble $LOCAL$, ce qui permet sa réduction (voir diagramme figure 2).

3.3 Prise en compte des alignements

Lorsque suffisamment d'informations peuvent être réunies sur le rangement des données pendant la phase de compilation, il est possible de déduire la réponse aux tests de localité de certains nœuds des réponses de certains autres nœuds. Deux nœuds sont dits alignés lorsqu'ils produisent toujours la même réponse à un test de localité. Les extensions HPF de Fortran90 prévoient des directives de distribution et d'alignement des données.

Ces informations sont traduites dans l'algorithme de génération de code en terme de groupes d'alignement.

	vivant	tache	local	nlocal	dist	Code généré
e0	{:0}	{5}	{0}	∅	{1,2,3,4,6}	
e1	{0:1,0:2,0:3}	{5}	∅	∅	{1,2,3,4,6}	`if(EstNonLocal(c(i))) AllerA e2;`
e3	{0:1,0:2,0:3}	{5}	{1}	∅	{2,3,4,6}	`ci= c(i);`
e4	{0:2,0:3,1:4}	{5}	∅	∅	{2,3,4,6}	`if(EstNonLocal(v(i))) AllerA e5;`
e6	{0:2,0:3,1:4}	{5}	{2}	∅	{3,4,6}	`vi= v(i);`
e7	{0:3,1:4,2:5}	{5}	∅	∅	{3,4,6}	`if(EstNonLocal(l(i))) AllerA e8;`
e9	{0:3,1:4,2:5}	{5}	{3}	∅	{4,6}	`li= l(i);`
e10	{1:4,2:5,3:6}	{5}	∅	∅	{4,6}	`if(EstNonLocal(b(ci))) AllerA e11;`
e12	{1:4,2:5,3:6}	{5}	{4}	∅	{6}	`bi= b(ci);`
e13	{2:5,3:6,4:5}	{5}	∅	∅	{6}	`vibi= vi*bi;`
e14	{3:6,5:6}	∅	∅	∅	{6}	`if(EstNonLocal(a(li))) AllerA e15;`
e16	{3:6,5:6}	∅	{6}	∅	∅	`a(li)= a(li)+vibi;`
e17	{6:}	∅	∅	∅	∅	`Termine;`
e15	{3:6,5:6}	∅	∅	{6}	∅	`MigrerSur(Prop(a(li)), e16, li, vibi );`
e11	{1:4,2:5,3:6}	{5}	∅	{4}	{6}	`MigrerSur(Prop(b(ci)), e12, ci, vi, li );`
e8	{0:3,1:4,2:5}	{5}	∅	{3}	{4,6}	`if(EstNonLocal(b(ci))) AllerA e18;`
e19	{0:3,1:4,2:5}	{5}	{4}	{3}	{6}	`bi= b(ci);`
e20	{0:3,2:5,4:5}	{5}	∅	{3}	{6}	`vibi= vi*bi;`
e21	{0:3,5:6}	∅	∅	{3}	{6}	`MigrerSur(Prop(l(i)), e22, i, vibi );`
e22	{0:3,5:6}	∅	{3}	∅	{6}	`li= l(i);`
	{3:6,5:6}	∅	∅	∅	{6}	`AllerA e14;`
e18	{0:3,1:4,2:5}	{5}	∅	{3,4}	{6}	`MigrerSur(Prop(l(i)), e9, i, ci, vi );`
e5	{0:2,0:3,1:4}	{5}	∅	{2}	{3,4,6}	`if(EstNonLocal(l(i))) AllerA e23;`
e24	{0:2,0:3,1:4}	{5}	{3}	{2}	{4,6}	`li= l(i);`
e25	{0:2,1:4,3:6}	{5}	∅	{2}	{4,6}	`if(EstNonLocal(b(ci))) AllerA e26;`
e27	{0:2,1:4,3:6}	{5}	{4}	{2}	{6}	`bi= b(ci);`
e28	{0:2,3:6,4:5}	{5}	∅	{2}	{6}	`MigrerSur(Prop(v(i)), e29, i, li, bi );`
e29	{0:2,3:6,4:5}	{5}	{2}	∅	{6}	`vi= v(i);`
	{2:5,3:6,4:5}	{5}	∅	∅	{6}	`AllerA e13;`
e26	{0:2,1:4,3:6}	{5}	∅	{2,4}	{6}	`MigrerSur(Prop(v(i)), e30, i, ci, li );`
e30	{0:2,1:4,3:6}	{5}	{2}	∅	{4,6}	`vi= v(i);`
	{1:4,2:5,3:6}	{5}	∅	∅	{4,6}	`AllerA e10;`
e23	{0:2,0:3,1:4}	{5}	∅	{2,3}	{4,6}	`if(EstNonLocal(b(ci))) AllerA e31;`
e32	{0:2,0:3,1:4}	{5}	{4}	{2,3}	{6}	`bi= b(ci);`
e33	{0:2,0:3,4:5}	{5}	∅	{2,3}	{6}	`MigrerSur(Prop(v(i)), e34, i, bi );`
e34	{0:2,0:3,4:5}	{5}	{2}	∅	{3,6}	`vi= v(i);`
e35	{0:3,2:5,4:5}	{5}	∅	∅	{3,6}	`vibi= vi*bi;`
e36	{0:3,5:6}	∅	∅	∅	{3,6}	`if(EstNonLocal(l(i))) AllerA e21;`
	{0:3,5:6}	∅	{3}	∅	{6}	`AllerA e22;`
e31	{0:2,0:3,1:4}	{5}	∅	{2,3,4}	{6}	`MigrerSur(Prop(v(i)), e6, i, ci );`
e2	{0:1,0:2,0:3}	{5}	∅	{1}	{2,3,4,6}	`if(EstNonLocal(v(i))) AllerA e37;`
e38	{0:1,0:2,0:3}	{5}	{2}	{1}	{3,4,6}	`vi= v(i);`
e39	{0:1,0:3,2:5}	{5}	∅	{1}	{3,4,6}	`if(EstNonLocal(l(i))) AllerA e40;`
e41	{0:1,0:3,2:5}	{5}	{3}	{1}	{4,6}	`li= l(i);`
e42	{0:1,2:5,3:6}	{5}	∅	{1}	{4,6}	`MigrerSur(Prop(c(i)), e43, i, vi, li );`
e43	{0:1,2:5,3:6}	{5}	{1}	∅	{4,6}	`ci= c(i);`

FIG. 4 - *exemple génération de code, première partie*

	vivant	tache	local	nlocal	dist	Code généré
	{1:4,2:5,3:6}	{5}	∅	∅	{4,6}	`AllerA e10;`
e40	{0:1,0:3,2:5}	{5}	∅	{1,3}	{4,6}	`MigrerSur(Prop(c(i)), e44, i, vi );`
e44	{0:1,0:3,2:5}	{5}	{1}	∅	{3,4,6}	`ci= c(i);`
	{0:3,1:4,2:5}	{5}	∅	∅	{3,4,6}	`AllerA e7;`
e37	{0:1,0:2,0:3}	{5}	∅	{1,2}	{3,4,6}	`if(EstNonLocal(l(i))) AllerA e45;`
e46	{0:1,0:2,0:3}	{5}	{3}	{1,2}	{4,6}	`li= l(i);`
e47	{0:1,0:2,3:6}	{5}	∅	{1,2}	{4,6}	`MigrerSur(Prop(c(i)), e48, i, li );`
e48	{0:1,0:2,3:6}	{5}	{1}	∅	{2,4,6}	`ci= c(i);`
e49	{0:2,1:4,3:6}	{5}	∅	∅	{2,4,6}	`if(EstNonLocal(v(i))) AllerA e25;`
	{0:2,1:4,3:6}	{5}	{2}	∅	{4,6}	`AllerA e30;`
e45	{0:1,0:2,0:3}	{5}	∅	{1,2,3}	{4,6}	`MigrerSur(Prop(c(i)), e3, i );`

FIG. 5 - *exemple génération de code, dernière partie*

Les groupes d'alignement

Un groupe d'alignement est un ensemble de nœuds distribués qui fournissent la même réponse aux tests de localité. Les groupes d'alignement *initiaux* sont construits par analyse des déclarations des tableaux et des directives d'alignement du programme. Les groupes d'alignement peuvent être construits dynamiquement par fusion de groupes à la suite des tests de localité. Les groupes d'alignement sont disjoints. Un nœud donné appartient à un seul groupe d'alignement. L'ensemble $LOCAL$ est un groupe d'alignement particulier.

Les ensembles incompatibles

A chaque groupe d'alignement est associé un ensemble de nœuds qui ne sont jamais locaux aux nœuds du groupe d'alignement : l'ensemble incompatible du groupe d'alignement. En général, les ensembles incompatibles initiaux sont vides. Les extensions de type HPF n'en prévoient pas la spécification. Ces ensembles sont construits dynamiquement pendant le processus de compilation. L'ensemble des nœuds $NLOCAL$ constitue l'ensemble incompatible du groupe $LOCAL$. Un nœud peut appartenir à plusieurs ensembles incompatibles. Les ensembles incompatibles sont cohérents si, lorsqu'un nœud e_i appartient à l'ensemble incompatible du groupe d'alignement d'un nœud e_j, ce nœud e_j appartient à l'ensemble incompatible du nœud e_i. Cette cohérence est maintenue par construction des groupes.

Construction des groupes d'alignement

Les groupes d'alignement et leurs ensembles incompatibles sont construits après chaque test de localité et après chaque migration.

Sur la branche "succès" d'un test de localité, le groupe auquel appartient le nœud sur lequel porte le test est fusionné avec le groupe $LOCAL$. Son ensemble incompatible est fusionné avec l'ensemble $NLOCAL$. Cette opération réduit le nombre de groupes de une unité.

Sur la branche "échec" d'un test de localité, les nœuds du groupe d'alignement sur lequel porte le test sont ajoutés à l'ensemble $NLOCAL$. Le groupe d'alignement est conservé.

	vivant	tache	local	nlocal	dist	Code généré
e0	{:0}	{5}	{0}	∅	{1,2,3,4,6}	
e1	{0:1,0:2,0:3}	{5}	∅	∅	{1,2,3,4,6}	`if(EstNonLocal(c(i))) AllerA e2;`
e3	{0:1,0:2,0:3}	{5}	{1,3}	∅	{2,4,6}	`ci= c(i);`
e4	{0:2,0:3,1:4}	{5}	{3}	∅	{2,4,6}	`li= l(i);`
e5	{0:2,1:4,3:6}	{5}	∅	∅	{2,4,6}	`if(EstNonLocal(v(i))) AllerA e6;`
e7	{0:2,1:4,3:6}	{5}	{2}	∅	{4,6}	`vi= v(i);`
e8	{1:4,2:5,3:6}	{5}	∅	∅	{4,6}	`if(EstNonLocal(b(ci))) AllerA e9;`
e10	{1:4,2:5,3:6}	{5}	{4}	∅	{6}	`bi= b(ci);`
e11	{2:5,3:6,4:5}	{5}	∅	∅	{6}	`vibi= vi*bi;`
e12	{3:6,5:6}	∅	∅	∅	{6}	`if(EstNonLocal(a(li))) AllerA e13;`
e14	{3:6,5:6}	∅	{6}	∅	∅	`a(li)= a(li)+vibi;`
e15	{6:}	∅	∅	∅	∅	`Termine;`
e13	{3:6,5:6}	∅	∅	{6}	∅	`MigrerSur(Prop(a(li)), e14, li, vibi );`
e9	{1:4,2:5,3:6}	{5}	∅	{4}	{6}	`MigrerSur(Prop(b(ci)), e10, ci, vi, li );`
e6	{0:2,1:4,3:6}	{5}	∅	{2}	{4,6}	`if(EstNonLocal(b(ci))) AllerA e16;`
e17	{0:2,1:4,3:6}	{5}	{4}	{2}	{6}	`bi= b(ci);`
e18	{0:2,3:6,4:5}	{5}	∅	{2}	{6}	`MigrerSur(Prop(v(i)), e19, i, li, bi );`
e19	{0:2,3:6,4:5}	{5}	{2}	∅	{6}	`vi= v(i);`
	{2:5,3:6,4:5}	{5}	∅	∅	{6}	`AllerA e11;`
e16	{0:2,1:4,3:6}	{5}	∅	{2,4}	{6}	`MigrerSur(Prop(v(i)), e7, i, ci, li );`
e2	{0:1,0:2,0:3}	{5}	∅	{1,3}	{2,4,6}	`if(EstNonLocal(v(i))) AllerA e20;`
e21	{0:1,0:2,0:3}	{5}	{2}	{1,3}	{4,6}	`vi= v(i);`
e22	{0:1,0:3,2:5}	{5}	∅	{1,3}	{4,6}	`MigrerSur(Prop(c(i)), e23, i, vi );`
e23	{0:1,0:3,2:5}	{5}	{1,3}	∅	{4,6}	`ci= c(i);`
e24	{0:3,1:4,2:5}	{5}	{3}	∅	{4,6}	`li= l(i);`
	{1:4,2:5,3:6}	{5}	∅	∅	{4,6}	`AllerA e8;`
e20	{0:1,0:2,0:3}	{5}	∅	{1,2,3}	{4,6}	`MigrerSur(Prop(c(i)), e3, i );`

FIG. 6 - *génération de code avec alignement de* `c(i)` *et* `l(i)`

Après une migration, le groupe $LOCAL$ devient l'un des groupes d'alignement et l'ensemble $NLOCAL$ son ensemble incompatible. Le groupe sur lequel porte la migration devient le groupe $LOCAL$. L'ensemble $NLOCAL$ est constitué des éléments de l'ensemble incompatible du groupe choisi et des éléments de l'ancien groupe $LOCAL$.

Après toute modification de l'ensemble $NLOCAL$, les ensembles incompatibles des groupes d'alignement des nœuds appartenant à cet ensemble sont mis à jour pour refléter les relations d'incompatibilité entre l'ensemble $LOCAL$ et l'ensemble $NLOCAL$.

Exemple de prise en compte des alignements

La figure 6 montre le résultat de la génération de code de l'exemple précédent en considérant un alignement des éléments `c(i)` et `l(i)`. Le nombre d'états de l'automate est fortement réduit. La construction dynamique des groupes d'alignement n'a cependant aucun effet sur la génération de code des corps de boucles simples. Les résultats ne sont visibles que pour les cas où des nœuds qui ne peuvent pas être réduits simultanément sont alignés. Un tel exemple est présenté dans les figures 29 à 33.

3.4 Traitement des sorties multiples

Il est fréquent qu'un corps de boucle parallèle se termine par deux instructions d'affectation ou plus. Ceci se traduit dans le graphe de flot/dépendance par la présence de sorties multiples. En principe, le présence de sorties multiples du graphe signifie qu'elles peuvent être traitées indépendamment. Dès que le graphe en cours de réduction se présente sous la forme de deux sous-graphes disjoints, il est possible de traiter ces deux sous-graphes par deux tâches indépendantes. L'effet global escompté est une diminution du nombre total de migrations. Cependant, cette séparation ne paraît souhaitable que lorsque ces deux tâches doivent migrer vers deux processeurs distincts.

Algorithme de traitement des sorties multiples

Le traitement des sorties multiples se fait en trois étapes au cours de la production d'un ordre de migration. Si l'intersection des ensembles de nœuds prédécesseurs de chaque sortie du graphe est vide, on est en présence de deux (ou plus) sous-graphes disjoints. Le traitement de ces sous-graphes est séparé si leurs nœuds réductibles ne sont pas liés par des groupes d'alignement.

La création d'une sous-tâche se traduit par la production d'un ordre `DetacheSur(dest, etiq, contexte...)`. Cet ordre provoque la création d'une nouvelle tâche sur le processeur `dest`, s'exécutant à l'adresse `etiq`, l'exécution de la tâche courante se poursuivant en séquence. Les figures 29 à 33 contiennent des exemples de création de sous-tâches (en `e64` et `e113`).

3.5 Limitation de la complexité du code généré

Le nombre d'états de l'automate produit, et donc le volume du code généré augmente très vite lorsque le nombre de nœuds réductibles dans l'état distribué augmente. L'explosion du code peut être évitée en limitant le nombre de tests de localité en échec avant une migration. La figure 7 montre le code généré pour l'exemple des figures 4 et 5 lorsque le nombre d'échecs sur les tests de localité est limité à deux. Cette technique permet de maîtriser le volume du code produit.

3.6 Distribution initiale des tâches migrantes

La création des tâches migrantes d'une construction parallèle est distribuée sur l'ensemble des processeurs : chaque itération doit être exécutée une et une seule fois. Pour éviter la première migration des tâches, il est généralement possible de calquer la distribution des itérations sur la distribution mémoire de l'un des accès entrants à un objet distribué dans le corps de boucle. Un accès entrant est défini comme un accès à une donnée distribuée qui ne dépend pas (au sens contrôle, flot ou dépendance) d'un accès à une autre donnée distribuée. Cet accès peut être choisi comme cible favorite de la première migration. Il faut noter que le choix de cette distribution initiale n'a aucune relation avec une recherche d'équilibrage

	vivant	tache	local	nlocal	dist	Code généré
e0	{:0}	{5}	{0}	∅	{1,2,3,4,6}	
e1	{0:1,0:2,0:3}	{5}	∅	∅	{1,2,3,4,6}	`if(EstNonLocal(c(i))) AllerA e2;`
e3	{0:1,0:2,0:3}	{5}	{1}	∅	{2,3,4,6}	`ci= c(i);`
e4	{0:2,0:3,1:4}	{5}	∅	∅	{2,3,4,6}	`if(EstNonLocal(v(i))) AllerA e5;`
e6	{0:2,0:3,1:4}	{5}	{2}	∅	{3,4,6}	`vi= v(i);`
e7	{0:3,1:4,2:5}	{5}	∅	∅	{3,4,6}	`if(EstNonLocal(l(i))) AllerA e8;`
e9	{0:3,1:4,2:5}	{5}	{3}	∅	{4,6}	`li= l(i);`
e10	{1:4,2:5,3:6}	{5}	∅	∅	{4,6}	`if(EstNonLocal(b(ci))) AllerA e11;`
e12	{1:4,2:5,3:6}	{5}	{4}	∅	{6}	`bi= b(ci);`
e13	{2:5,3:6,4:5}	{5}	∅	∅	{6}	`vibi= vi*bi;`
e14	{3:6,5:6}	∅	∅	∅	{6}	`if(EstNonLocal(a(li))) AllerA e15;`
e16	{3:6,5:6}	∅	{6}	∅	∅	`a(li)= a(li)+vibi;`
e17	{6:}	∅	∅	∅	∅	`Termine;`
e15	{3:6,5:6}	∅	∅	{6}	∅	`MigrerSur(Prop(a(li)), e16, li, vibi );`
e11	{1:4,2:5,3:6}	{5}	∅	{4}	{6}	`MigrerSur(Prop(b(ci)), e12, ci, vi, li );`
e8	{0:3,1:4,2:5}	{5}	∅	{3}	{4,6}	`if(EstNonLocal(b(ci))) AllerA e18;`
e19	{0:3,1:4,2:5}	{5}	{4}	{3}	{6}	`bi= b(ci);`
e20	{0:3,2:5,4:5}	{5}	∅	{3}	{6}	`vibi= vi*bi;`
e21	{0:3,5:6}	∅	∅	{3}	{6}	`MigrerSur(Prop(l(i)), e22, i, vibi );`
e22	{0:3,5:6}	∅	{3}	∅	{6}	`li= l(i);`
	{3:6,5:6}	∅	∅	∅	{6}	`AllerA e14;`
e18	{0:3,1:4,2:5}	{5}	∅	{3,4}	{6}	`MigrerSur(Prop(l(i)), e9, i, ci, vi );`
e5	{0:2,0:3,1:4}	{5}	∅	{2}	{3,4,6}	`if(EstNonLocal(l(i))) AllerA e23;`
e24	{0:2,0:3,1:4}	{5}	{3}	{2}	{4,6}	`li= l(i);`
e25	{0:2,1:4,3:6}	{5}	∅	{2}	{4,6}	`if(EstNonLocal(b(ci))) AllerA e26;`
e27	{0:2,1:4,3:6}	{5}	{4}	{2}	{6}	`bi= b(ci);`
e28	{0:2,3:6,4:5}	{5}	∅	{2}	{6}	`MigrerSur(Prop(v(i)), e29, i, li, bi );`
e29	{0:2,3:6,4:5}	{5}	{2}	∅	{6}	`vi= v(i);`
	{2:5,3:6,4:5}	{5}	∅	∅	{6}	`AllerA e13;`
e26	{0:2,1:4,3:6}	{5}	∅	{2,4}	{6}	`MigrerSur(Prop(v(i)), e30, i, ci, li );`
e30	{0:2,1:4,3:6}	{5}	{2}	∅	{4,6}	`vi= v(i);`
	{1:4,2:5,3:6}	{5}	∅	∅	{4,6}	`AllerA e10;`
e23	{0:2,0:3,1:4}	{5}	∅	{2,3}	{4,6}	`MigrerSur(Prop(v(i)), e6, i, ci );`
e2	{0:1,0:2,0:3}	{5}	∅	{1}	{2,3,4,6}	`if(EstNonLocal(v(i))) AllerA e31;`
e32	{0:1,0:2,0:3}	{5}	{2}	{1}	{3,4,6}	`vi= v(i);`
e33	{0:1,0:3,2:5}	{5}	∅	{1}	{3,4,6}	`if(EstNonLocal(l(i))) AllerA e34;`
e35	{0:1,0:3,2:5}	{5}	{3}	{1}	{4,6}	`li= l(i);`
e36	{0:1,2:5,3:6}	{5}	∅	{1}	{4,6}	`MigrerSur(Prop(c(i)), e37, i, vi, li );`
e37	{0:1,2:5,3:6}	{5}	{1}	∅	{4,6}	`ci= c(i);`
	{1:4,2:5,3:6}	{5}	∅	∅	{4,6}	`AllerA e10;`
e34	{0:1,0:3,2:5}	{5}	∅	{1,3}	{4,6}	`MigrerSur(Prop(c(i)), e38, i, vi );`
e38	{0:1,0:3,2:5}	{5}	{1}	∅	{3,4,6}	`ci= c(i);`
	{0:3,1:4,2:5}	{5}	∅	∅	{3,4,6}	`AllerA e7;`
e31	{0:1,0:2,0:3}	{5}	∅	{1,2}	{3,4,6}	`MigrerSur(Prop(c(i)), e3, i );`

FIG. 7 - *limitation du nombre d'échecs*

	vivant	tache	local	nlocal	dist	Code généré
e0	{:0}	{5}	{0,1,3}	∅	{2,4,6}	
e1	{0:1,0:2,0:3}	{5}	{1,3}	∅	{2,4,6}	`ci= c(i);`
e2	{0:2,0:3,1:4}	{5}	{3}	∅	{2,4,6}	`li= l(i);`
e3	{0:2,1:4,3:6}	{5}	∅	∅	{2,4,6}	`if(EstNonLocal(v(i))) AllerA e4;`
e5	{0:2,1:4,3:6}	{5}	{2}	∅	{4,6}	`vi= v(i);`
e6	{1:4,2:5,3:6}	{5}	∅	∅	{4,6}	`if(EstNonLocal(b(ci))) AllerA e7;`
e8	{1:4,2:5,3:6}	{5}	{4}	∅	{6}	`bi= b(ci);`
e9	{2:5,3:6,4:5}	{5}	∅	∅	{6}	`vibi= vi*bi;`
e10	{3:6,5:6}	∅	∅	∅	{6}	`if(EstNonLocal(a(li))) AllerA e11;`
e12	{3:6,5:6}	∅	{6}	∅	∅	`a(li)= a(li)+vibi;`
e13	{6:}	∅	∅	∅	∅	`Termine;`
e11	{3:6,5:6}	∅	∅	{6}	∅	`MigrerSur(Prop(a(li)), e12, li, vibi );`
e7	{1:4,2:5,3:6}	{5}	∅	{4}	{6}	`MigrerSur(Prop(b(ci)), e8, ci, vi, li );`
e4	{0:2,1:4,3:6}	{5}	∅	{2}	{4,6}	`if(EstNonLocal(b(ci))) AllerA e14;`
e15	{0:2,1:4,3:6}	{5}	{4}	{2}	{6}	`bi= b(ci);`
e16	{0:2,3:6,4:5}	{5}	∅	{2}	{6}	`MigrerSur(Prop(v(i)), e17, i, li, bi );`
e17	{0:2,3:6,4:5}	{5}	{2}	∅	{6}	`vi= v(i);`
	{2:5,3:6,4:5}	{5}	∅	∅	{6}	`AllerA e9;`
e14	{0:2,1:4,3:6}	{5}	∅	{2,4}	{6}	`MigrerSur(Prop(v(i)), e5, i, ci, li );`

FIG. 8 - *Distribution des itérations sur* `c(i)`

de charge : une tâche passe nécessairement par tous les processeurs qui possèdent dans leur mémoire l'une des données accédées. Par contre, le choix de l'accès sur lequel la distribution des itération est alignée peut influer sur le nombre total de migrations.

Le choix de distribution des itérations se traduit dans l'algorithme de génération de code par l'initialisation de l'alignement $LOCAL$ par l'ensemble d'alignement qui contient l'accès initial considéré. L'ensemble $NLOCAL$ est alors initialisé par l'ensemble incompatible correspondant. La figure 8 reprend le cas de la figure 6 en alignant la distribution des itérations sur le groupe d'alignement de la variable `c(i)`. Ce choix élimine la branche de l'automate qui traite la non localité de `c(i)`.

3.7 Structure des programmes transformés

Le schéma d'exécution de base pour les tâches migrantes est un schéma SPMD. Tous les processeurs exécutent le même programme. Entre deux constructions contenant des tâches migrantes, tous les processeurs exécutent exactement les mêmes instructions et affectent les mêmes valeurs aux variables dupliquées.

L'exécution d'une nouvelle construction contenant un accès à une variable distribuée entraîne les opérations suivantes :

- changement de l'identification des tâches
- exécution de la tâche initiale
- déclenchement de l'algorithme de terminaison
- boucle de traitement des messages

Lors d'enchaînements de l'exécution de constructions migrantes, il est possible qu'une tâche appartenant à une construction soit reçue par un processeur qui est toujours en train d'exécuter des tâches de la construction précédente. Ceci est rendu possible par l'asynchronisme des processeurs qui peuvent se trouver dans deux constructions voisines. Il est évident que l'état d'une tâche ne peut pas être interprété en dehors de son contexte. La confusion de contextes d'exécution peut être évitée de deux manières. L'insertion d'une barrière de synchronisation à l'entrée de toute nouvelle construction migrante peut résoudre ce problème à condition que la présence de cette barrière ne bloque pas les réceptions des messages. Une autre solution consiste à numéroter les constructions migrantes. Il est alors possible de placer les messages reçus dans des files d'attente différentes si elles n'appartiennent pas à la même construction. De cette manière, le traitement d'un message peut être différé si son numéro de construction ne correspond pas. La numérotation des construtions évite l'insertion de barrières de synchronisation.

3.8 Programmes cibles des tâches migrantes

La technique de génération de code proposée peut être appliquée à de nombreux programmes parallèles. Les corps des constructions succeptibles de migrer peuvent contenir des instructions de contrôle — boucles parallèles ou séquentielles, conditionnelles, instructions de saut même arrière et provoquant des boucles. La seule véritable contrainte réside dans le fait que, pour pouvoir migrer, chaque tâche doit retrouver le même contexte sur l'ensemble des processeur. Cette contrainte introduit une limitation sur les sous-programmes qui peuvent être appelés par les tâches migrantes: ces sous-programmes ne peuvent pas créer des constructions migrantes. Cette limitation est habituelle sur le modèle de programmation SPMD. Lorsqu'un sous-programme peut créer des constructions migrantes, les appels à ce sous-programme doivent obligatoirement être réalisés dans les sections SPMD à l'exécution. Les boucles parallèles qui pourraient englober ces appels sont exécutées en séquentiel. Lorsque les sous-programmes sont suffisamment simples, il est possible d'insérer leur corps au point d'appel, ce qui permet de lever toutes les restrictions.

4 Terminaison

Pour pouvoir quitter la boucle de traitement d'une construction migrante, un processeur doit détecter la fin de l'exécution de la construction. Dans le modèle des tâches migrantes, un processeur ne sait jamais à l'avance combien de tâches il doit traiter. Un tel calcul exigerait un prétraitement des données avant l'exécution de la construction migrante. De plus, les bibliothèques de communication existantes ne permettent la programmation de barrières de synchronisation interruptible (qui permettraient de relancer l'exécution de la boucle en cas de réception d'un message).

4.1 Algorithme de détection de la terminaison

L'algorithme de détection de la terminaison developpé pour le modèle des tâches migrantes fait appel à une notion de phase de calcul et à une détection des transitions de phases par association d'un poids à chaque message. Cet algorithme est mis en œuvre en faisant circuler des informations dans les messages qui transportent les tâches migrantes.

4.2 Les phases de calcul

Le système de calcul des phases suit les règles suivantes :

1. la phase 0 est la première phase de toute construction migrante
2. toutes les tâches sont crées durant cette phase initiale 0
3. toute tâche doit migrer au moins une fois durant chaque phase, ou doit se terminer
4. lorsque toutes les tâches migrantes ont été créées, le processeur passe dans la phase 1
5. lorsqu'un processeur a traité tous les messages qui lui ont été émis durant une phase i, il passe à la phase suivante $i+1$

Le nombre de migrations d'une tâche étant nécessairement fini, ces règles garantissent que toutes les tâches sont terminées lorsque la phase correspondant au nombre maximum de migrations est atteinte. L'absence de migration de tâches au cours d'une phase est également une preuve de terminaison de l'ensemble des tâches.

4.3 Les changements de phase

Un processeur ne peut passer à la phase suivante que lorsqu'il a traité tous les messages qui lui ont été émis durant la phase courante. Cette détection est réalisée par un système de poids qui suit les règles suivantes :

1. un poids strictement positif est associé à chaque message émis dans le système
2. la somme des poids des messages émis par un processeur p_i vers un même processeur p_j durant une même phase est égale à une valeur prédéfinie M
3. chaque processeur accumule les poids des messages reçus et traités par phase ; lorsque ce poids total atteint la valeur $(p-1)M$ (pour p processeurs) pour la phase courante, tous les messages émis durant cette phase ont été traités et la transition vers la phase suivante est permise
4. chaque changement de phase entraîne l'émission de tous les tampons de sortie en cours de remplissage (toutes les tâches d'un même message doivent être produites durant la même phase). Le poids associé à ce dernier message est calculé de manière à atteindre le total M pour l'ensemble des messages émis vers ce même processeur durant cette phase.

4.4 Détection de la terminaison

Lorsque le corps des tâches migrantes ne contient aucune instruction de contrôle, il est possible de déterminer statiquement le nombre maximum de migration que peut réaliser une tâche. Dans ce cas, la terminaison de la construction migrante est détectée dès que le numéro de la phase courante atteint cette valeur. Ceci est dû au fait qu'une tâche vivante doit migrer au moins une fois au cours de chaque phase.

Dans le cas où le corps de la construction contient des instructions de contrôle (boucle séquentielle, saut arrière etc), il n'est pas possible de calculer statiquement ce nombre maximum de migrations. Dans ce cas, on ajoute au dernier message émis vers chaque destinataire une information indiquant si une tâche a migré durant cette phase. Du coté réception, chaque processeur est capable de déduire, à la fin de chaque phase, si une tâche a migré durant la phase précédente. Si aucune tâche n'a migré, on déduit qu'il n'y a plus aucune tâche vivante et que le calcul est terminé.

Dans le cas de la détection dynamique, la dernière phase est toujours vide (aucune tâche n'ayant migré pendant la phase précédente). Cette phase vide produit l'effet d'une barrière de synchronisation. Par contre, cet algorithme dynamique s'adapte automatiquement au nombre maximum de migrations réelles des tâches. Ce n'est pas le cas de l'algorithme statique qui ne tient pas compte du nombre réel de migrations.

4.5 Commentaires

L'algorithme de terminaison proposé est totalement asynchrone. Chaque transition de phase agit comme une demi-barrière de synchronisation : une transition de phase est permise dès que tous les processeurs ont déclenché une transition sur la phase précédente.

A un instant donné, un processeur peut recevoir des messages émis durant deux phases voisines (asynchronisme). Cet algorithme résiste au désordre dans le traitement des messages : le dernier message ne contient aucun marqueur spécial.

4.6 Coût, complexité

Le coût mémoire de cet algorithme est faible : un compteur des poids émis vers chaque processeur et un compteur pour accumuler les poids reçus par phase. Chaque message contient le numéro de sa phase de production. On peut supposer que le poids implicite des messages est 1. Il suffit de coder le poids restant dans l'un des messages (le dernier en général).

Globalement, sur une architecture à p processeur, le coût en nombre de messages de cet algorithme est $p(p-1)$: chaque processeur doit émettre un message contenant le poids restant vers tout autre processeur. Ce nombre élevé de messages peut devenir un problème lorsque la plupart des messages émis ne contiennent que cette information. Ce cas se produit fréquemment lorsque les accès dans les mémoires sont suffisamment locaux. Il est également relativement fréquent dans les dernières phases, lorsqu'il ne reste que peu de tâches dans le système.

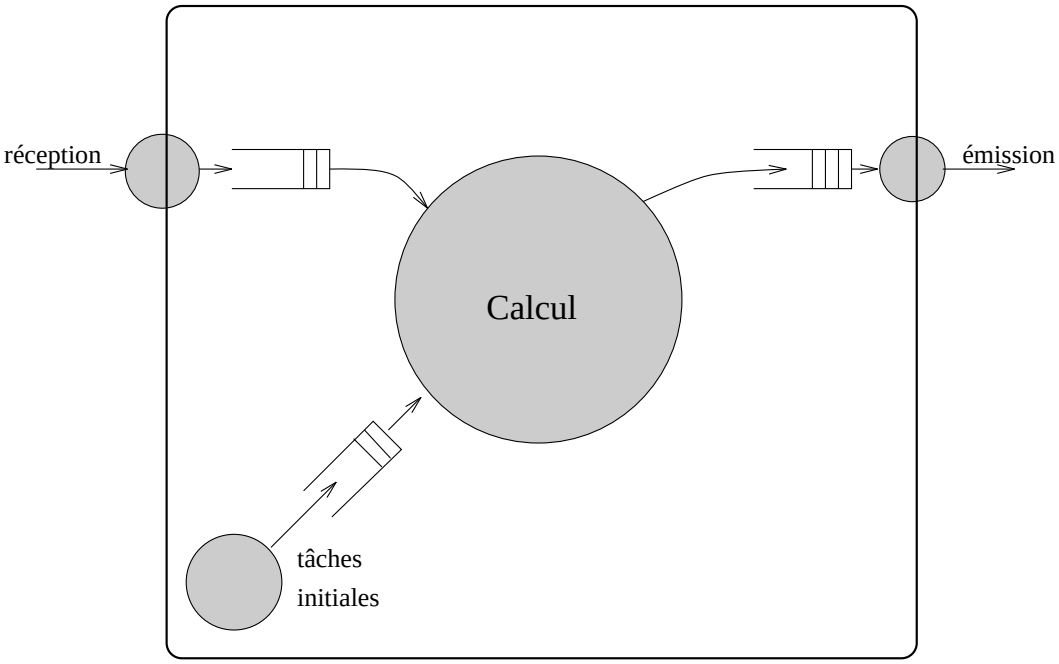

FIG. 9 - *modèle d'exécution*

Le coût de cet algorithme peut être ramené à $p \log p$ messages au total par réduction en arbre des poids en $\log p$ étapes. Durant l'étape i, chaque processeur reçoit du processeur $(p - 2^{i-1})(mod\ p)$ un message contenant, en plus d'éventuelles tâches, des poids restants pour les processeurs $(p+2^i)(mod\ p)$. Ces poids restants sont ajoutés aux poids restants locaux. Avant de passer à l'étape $i+1$, un processeur p émet vers le processeur $(p+2^i)(mod\ p)$ un message contenant les poids restants des processeurs de numéro de la forme $(p + 2^i + 2^{2i}j)(mod\ p)$. Cette version plus économe en nombre de messages permet d'éviter l'émission de "bouffées" de messages lors des changements de phases.

L'algorithme de traitement en arbre de la terminaison est décrit en annexe A.

5 Support d'exécution

Le modèle d'exécution par migration de tâches peut être mis en œuvre par trois familles de processus communiquant par files d'attente sur chaque processeur de l'architecture : un processus d'exécution des tâches, un ou plusieurs processus de réception des messages et un ou plusieurs processus d'émission (figure 9).

5.1 Processus d'émission

Le processus d'émission de tâches reçoit les messages à émettre du processus de calcul dans une file d'attente. Chaque message contient des tâches qui doivent migrer vers le même processeur au cours de la même phase. Pour éviter des problèmes de saturation des mémoires, l'émission d'un message est conditionnée par la présence d'un espace mémoire de réception chez le destinataire (voir 5.4). Le processus d'émission est chargé de la gestion des espaces d'émission.

5.2 Processus de réception

Le processus de réception des messages réalise l'interface entre le support de communication et le processus de calcul. Les messages reçus sont chaînés dans une file d'attente.

5.3 Processus de calcul

Sur chaque nœud de l'architecture, le processus de calcul est chargé de faire évoluer l'état des tâches locales jusqu'à ce qu'elles se terminent ou migrent. Ce processus communique avec les processus d'émission et de réception par des files d'attente. Ce processus est également chargé de la création des tâches initiales. Les tâches initiales sont crées lorsque aucun message reçu n'est en attente. Le fonctionnement du processus de calcul est interrompu dès qu'il a consommé tout l'espace de production de message. Cette exécution reprend lorsque de l'espace est rendu disponible par le processus d'émission.

5.4 Contrôle de flot

Le système d'échange de données étant asynchrone, il est nécessaire de contrôler strictement les espaces des tampons de stockage des messages. Sans contrôle, des messages peuvent être perdus ou peuvent provoquer des phénomènes de pagination qui font chuter les performances. Le contrôle du flot des messages permet d'assujettir une émission à la présence d'un espace tampon chez le destinataire.

Mise en œuvre du contrôle de flot

Le système fonctionne par échange de jetons, chaque jeton représentant un tampon chez le destinataire. L'émission d'un message provoque la consommation d'un jeton. Lorsqu'il n'y a plus de jetons pour un processeur, l'émission de messages à destination de ce processeur est suspendue. Les jetons reviennent dans les messages transmis en direction opposée. Ce système peut entraîner des émissions de messages uniquement pour transporter les jetons : lorsqu'un processeur possède tous les jetons d'un partenaire, il doit lui transmettre un message pour lui rendre ses droits d'émission. De plus, il faut éviter les situations d'interblocage (aucun des deux processeurs n'a le droit d'émettre vers l'autre alors qu'il possède tous ses jetons), par exemple en imposant que l'émission d'un message correspondant à un dernier jeton ne soit permise que si ce même message retourne un jeton au destinataire.

5.5 Implémentation des processus de communication

Lorsque le système d'exploitation des nœuds de l'architecture parallèle ne permet pas une utilisation efficace des processus parallèles, le fonctionnement des processus de communication peut être simulé par un système de traitement de signaux d'exception sur fins d'échanges ou par entrelacement de l'exécution de leurs codes.

Gestion des communications par traitement d'exception

Certains systèmes comme la bibliothèque de communication `nx` de la machine Paragon XP/S permettent de déclencher l'exécution d'une routine d'exception sur la fin d'un échange en entrée ou en sortie. Ce schéma est assez proche du modèle initial. Il permet d'enchaîner des échanges sans imposer à la routine de calcul de venir tester prériodiquement l'état des échanges en cours.

Entrelacement des codes

En général, le noyau système implanté sur chaque nœud d'une architecture massivement parallèle est très réduit et ne permet pas la coexistance de plusieurs processus coopérant (processus lourd de type Unix, ou processus plus léger de type pthread). Le schéma proposé doit alors être simulé en entrelaçant l'exécution du code de calcul, du code de réception et du code d'émission. Ce schéma transformé utilise les routines de communication non bloquantes de la bibliothèque. L'entrelacement de l'exécution est produit dans une boucle qui prend la forme suivante :

```
      i= 1;
      NonTerminee= True
      CalculInitial= True
      call ConstructionNouvelle()

      do while (NonTerminee)
        if (Interruption) then
          call TraiteInterruption()
        else if (MessageRecu&&TamponsOK) then
 .......  traite les tâches du message
        else if (CalculInitial&&TamponsOK) then
Boucle:   continue
          if (i<=n) then
 .........  une iteration de la boucle
             i= i+PasBoucle
             if (MessageRecu||TamponsOK) goto Sortie
             goto Boucle
          else
             CalculInitial= False
```

```
 ..........  Initialise mecanisme de terminaison
             goto Sortie
           end if
         else
           call TachedeFond()
         end if
Sortie: continue
       end do
```

Dans ce segment de code, la boucle principale est chargée d'entrelacer l'exécution des processus de l'exécutif. Le marqueur `Interruption` est positionné lorsqu'une fin d'émission ou de réception de message est signalée par le système de communication. L'évaluation de ce marqueur peut entraîner un appel à la librairie de communication pour analyser l'état des requêtes en cours. La routine `TraiteInterruption` simule le fonctionnement des processus d'émission et de réception. L'appel de cette routine peut provoquer l'émission d'un nouveau message et l'exécution d'une nouvelle commande de réception.

Le marqueur `MessageRecu` signale la présence de messages dans les files d'attente en entrée. Enfin la variable `CalculInitial` reste positionnée tant que toutes les tâches de la construction migrante n'ont pas été crées. Le mécanisme de terminaison décrit au paragraphe 4 est déclenché après la création de la dernière tâche.

L'exécution des sections de calcul est contrôlée par la variable de garde `TamponsOK`. Cette variable interdit tout calcul lorsqu'il n'y a plus de tampon disponible en sortie (limitation du volume de mémoire occupé par les messages).

Dans ce modèle, l'émission d'un message par la tâche de calcul se traduit, soit par un appel direct à une routine d'émission lorsqu'aucune autre requête n'est en cours, soit par le rangement du message dans une file d'attente.

6 Expérimentations sur la Paragon XP/S

Le modèle de tâches migrantes a été expérimenté sur un algorithme irrégulier : un code d'assemblage d'une partie droite pour un maillage irrégulier [8]. Le code Fortran d'origine a été transformé par l'algorithme décrit au paragraphe 3.

6.1 Programme Fortran

Le comportement du sous-programme `assem` de la figure 10 a été analysé sur la Paragon. Ce sous-programme contient deux boucles parallèles. Compte tenu des alignements des éléments des tableaux accédés, seule la première entraine des accès non locaux dans les mémoires. Dans ce qui suit, seule la génération du code de la première boucle est commentée, l'autre boucle ne subissant aucune transformation — à part la transformation des fonctions d'indiçage.

```
      subroutine assem(n,ns,nt,me,dt,ynm1,yn,ynp1,v1,cq,aire)
      implicit real*8(a-h,o-z)
      integer ns,nt,me(3,nt),i,j
      real*8 ynm1(ns),yn(ns),ynp1(ns),v1(ns)
      real*8 cq(6,nt),aire(nt)
      real*8 gxn,gyn,dt
c
c     boucle en temps
c     ***************
c

      do l=1,n
       do j=1,nt
         gxn= yn(me(1,j))*cq(1,j)+yn(me(2,j))*cq(3,j)+yn(me(3,j))*cq(5,j)
         gyn= yn(me(1,j))*cq(2,j)+yn(me(2,j))*cq(4,j)+yn(me(3,j))*cq(6,j)
         ynp1(me(1,j))=ynp1(me(1,j))+aire(j)*(gxn*cq(1,j)+gyn*cq(2,j))
         ynp1(me(2,j))=ynp1(me(2,j))+aire(j)*(gxn*cq(3,j)+gyn*cq(4,j))
         ynp1(me(3,j))=ynp1(me(3,j))+aire(j)*(gxn*cq(5,j)+gyn*cq(6,j))
       end do
c
       do i=1,ns
        y=yn(i)+dt*v1(i)+0.5*dt*ynp1(i)
        ynm1(i)=yn(i)
        yn(i)=y
        ynp1(i)=0.
       end do

c fin de la boucle sur l
      end do
c
      end
```

FIG. 10 - *programme Fortran source*

Les étapes successives de la génération de code de ce sous-programme Fortran sont détaillées en annexe B.

6.2 L'exécutif

L'exécution des constructions migrantes a été gérée par une boucle entrelaçant le calcul des tâches, les émissions des messages et leur réception. Ces routines utilisent les routines de communication non bloquantes disponibles dans la bibliothèque de communication. L'état des communications en cours est testé périodiquement par des appels à des fonctions de cette bibliothèque. Cette bibliothèque permet également de déclencher l'exécution de routines d'exception sur les fins d'échanges, solution qui permettrait d'éviter la lourdeur des tests périodiques. Mais, dans la version courante du système, l'utilisation de cette fonctionalité ne donne pas de bons résultats.

6.3 Les données de l'expérimentation

Le code expérimental a été exécuté sur deux jeux de données : `fal60k` et `cav200kw`. Le premier maillage contient $ns = 60590$ nœuds et $nt = 119332$ triangles. Le maillage `cav200kw` contient $ns = 207691$ nœuds et $nt = 411380$ triangles. Dans ces jeux de données, les maillages sont structurés par deux types de tableaux. L'un des types de tableaux contient des informations associées aux nœuds, le deuxième type de tableaux contenant des informations sur les triangles. Ces deux structures de données sont reliées par le tableau `me` qui contient l'identification des trois nœuds sommets de chaque triangle. Dans un triangle, l'accès aux données portées par un sommet se fait à travers ce vecteur de nommage, accès qui se traduit pas une indirection (double indiçage) et peut entraîner des lectures ou des écritures sur des mémoires locales. Le nombre de migrations au cours du calcul sur les triangles dépend fortement du nombre d'indirections non locales : du nombre de fois que les sommets d'un triangle ne sont pas décrits sur le même processeur que le triangle. Il est possible de modifier le comportement mémoire du maillage en permutant des triangles et/ou des nœuds par des techniques de renumérotation. Pour chacun des maillages, le comportement du programme a été mesuré sur les données initiales, sur des données restructurées pour améliorer la localité des accès et sur des données renumérotées aléatoirement pour mesurer le comportement du système en cas d'absence de localité. L'amélioration de localité a été obtenu par un algorithme itératif de renumérotation par couche. À chaque itération, cet algorithme renumérote tous les nœuds et les triangles voisins des nœuds et triangles renumérotés au cours de l'itération précédente. La répartition par bloc des nœuds et des triangles entraîne l'allocation de couches successives sur un même processeur. Les seuls accès indirects au cours du balayage des triangles se produisent sur les deux couches externes de chaque bloc. L'avantage de cette renumérotation est qu'elle ne dépend pas du nombre de processeur. Cependant, les triangles qui entraînent un accès non local appartiennent toujours aux deux couches voisines et leur nombre ne dépend pas du nombre de couches d'un processeur. Le nombre d'accès non locaux reste stable lorsque le nombre de processeurs augmente. La figure 11 montre la formation des couches lorsque le maillage du problème `fal` renuméroté est chargé par blocs

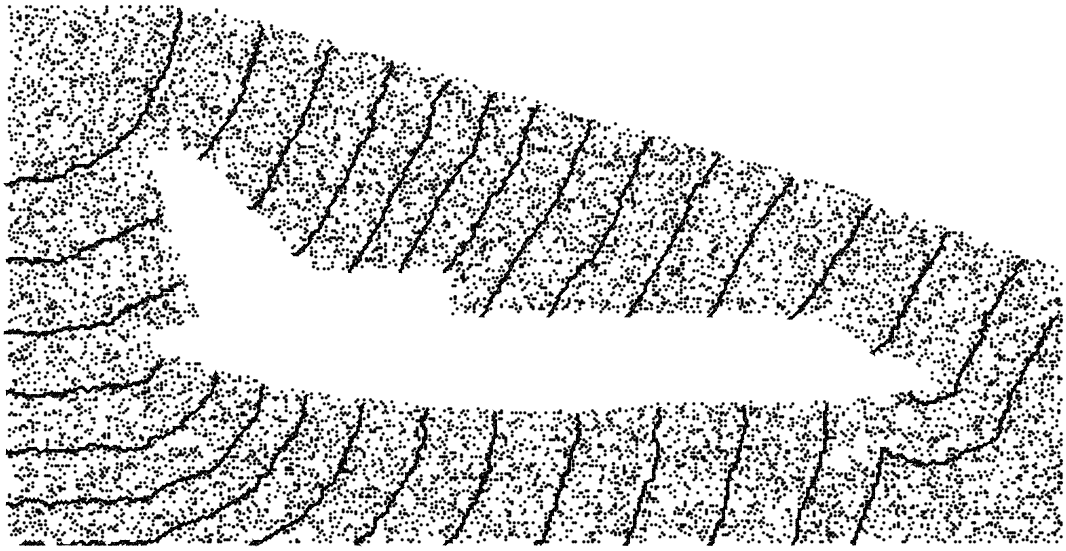

FIG. 11 - *Formation des couches sur* `fal` *renuméroté*

sur seize processeurs. Les lignes qui apparaissent correspondent aux triangles qui entraînent des communications.

Dans les figures qui suivent, les courbes portent les étiquettes suivantes.

fal60k.d : données de départ du maillage fal60k

fal60k.r : maillage fal60k renuméroté par couche

fal60k.m : maillage fal60k renuméroté aléatoirement

cav200kw.d : données de départ du maillage cav200kw

cav200kw.r : maillage cav200kw renuméroté par couche

cav200kw.m : maillage cav200kw renuméroté aléatoirement

6.4 Temps d'exécution de référence

Pour en mesurer l'efficacité, les temps d'exécution du programme transformé doivent être comparés au temps d'exécution du programme fortran de départ sur un nœud de l'architecture parallèle. Or, les seize mega-octets de mémoire centrale de chaque processeur de la Paragon ne permettent pas de charger le programme initial et ses données sans entraîner des paginations. Les temps de référence ont été obtenus sur un processeur de puissance équivalente et doté d'une mémoire plus importante : une station de travail *SPARCstation-10*

	secondes
`fal60k.r`	8.63
`fal60k.d`	9.10
`fal60k.m`	22.80
`cav200kw.r`	29.77
`cav200kw.d`	36.95
`cav200kw.m`	107.48

FIG. 12 - *Temps d'exécution sur SS10/41*

SuperSPARC model 41 dotée d'un SuperCache et d'une mémoire principale de 512 méga octets. La figure 12 rassemble les résultats mesurés. A noter la forte dégradation des performances pour les maillages renumérotés aléatoirement. Dans ce cas, la trop faible localité dans les accès à la mémoire entraîne un rechargement très fréquent des systèmes d'anté-mémoires — cache secondaire, TLB. Un comportement similaire a également pu être constaté sur les processeurs de la Paragon.

6.5 Temps d'exécution sur la Paragon X/PS

Le code transformé a été exécuté sur l'Intel Paragon X/PS de l'IRISA sous le système Mach/OSF1. Le nombre de processeurs alloués varie du minimum permettant une exécution sans pagination (2 processeurs pour fal60k et 6 processeurs pour cav200kw) jusquà 56. Le temps d'exécution du sous-programme `assem` a été mesuré par des appels à la fonction `dclock()` qui fournit le temps écoulé depuis le rechargement du système avec une grande précision.

Le temps d'exécution pour `fal60k.r` (figure 13) décroit rapidement pour descendre en dessous d'une seconde sur une vingtaine de processeurs. Au delà de ce nombre, ce temps est stable ; ceci s'explique par le fait que, pour les maillages renumérotés, le coût en communication pour chaque processeur reste stable et dépend très peu du nombre de triangles alloués.

Pour les versions d'origine et renumérotées aléatoirement, le coût en communication pour un processeur dépend essentiellement du nombre de triangles hébergés par ce processeur. Pour ces exemples, les coûts mesurés en communication et en calcul varient de façon similaire et sont proportionnels au nombre de triangles. Les courbes présentées reflètent cette variation. A noter que le temps d'exécution semble continuer à décroitre au-delà de cinquante six processeurs.

Les courbes 13 et 14 permettent également de comparer les temps d'exécution du code parallélisé avec le temps d'exécution du programme séquentiel de départ. À partir de deux processeurs et pour des données renumérotées, la version parallèle semble plus rapide que le code d'origine. Ce résultat s'explique facilement par la faible importance des communications.

Lorsque les données provoquent de nombreuses migrations, il faut au moins six processeurs pour que le code parallèle prenne l'avantage.

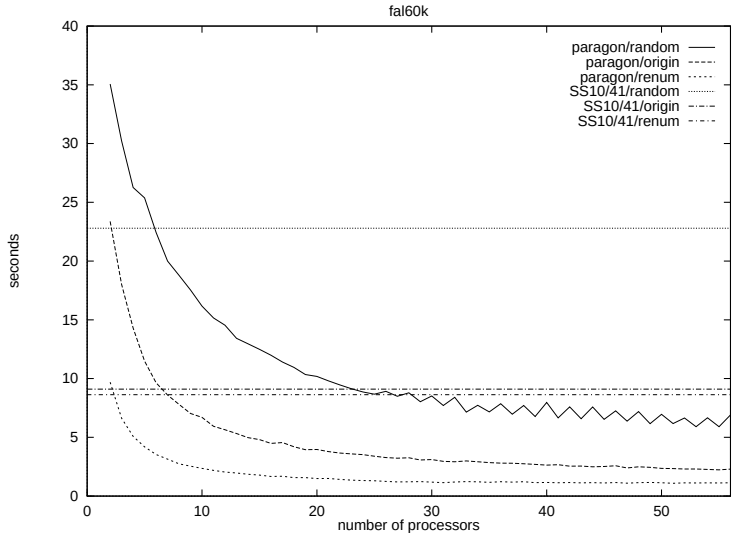

FIG. 13 - *temps d'exécution* `fal`

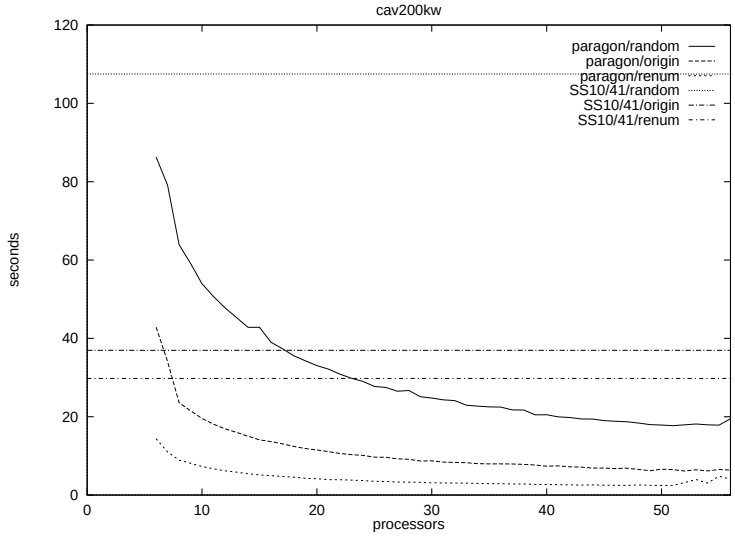

FIG. 14 - *temps d'exécution* `cav`

6.6 Coût des communications

Deux facteurs essentiels sont à prendre en considération dans les communications par messages : le volume des messages échangés et leur nombre.

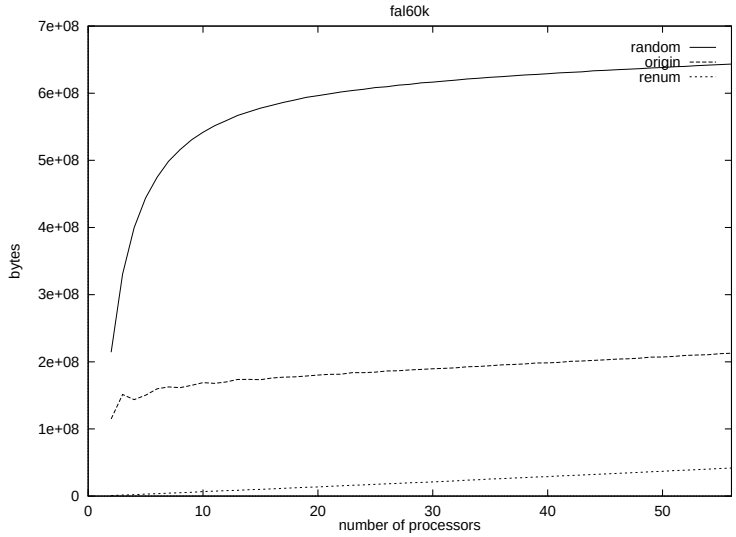

FIG. 15 - *volume total des messages pour* `fal`

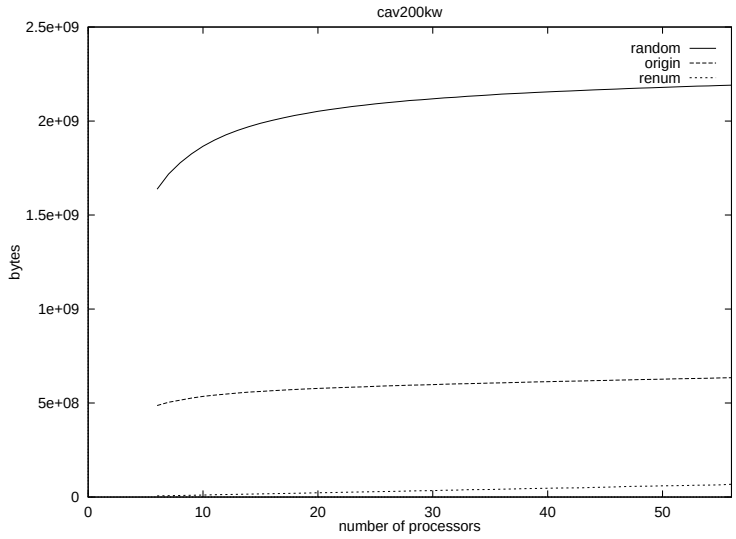

FIG. 16 - *volume total des messages pour* `cav`

Volume de messages échangés

L'effet du volume échangé sur les performances dépend fortement du débit potentiel des échanges sur l'architecture. Mais, pour un volume total de données échangées fixe, d'autre

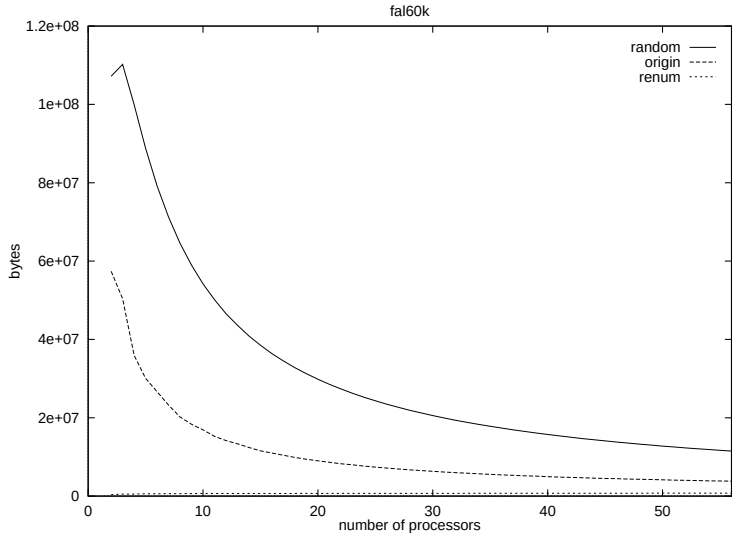

FIG. 17 - *volume de messages échangés par processeur sur* `fal`

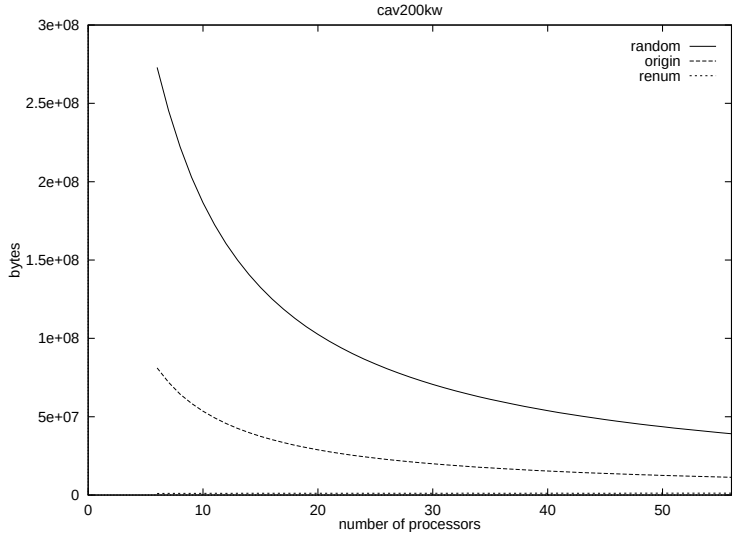

FIG. 18 - *volume de messages échangés par processeur sur* `cav`

facteurs comme le nombre de "voisins" doivent être considérés. En effet, à partir d'un certain seuil, un nombre élevé de voisin tend à produire des messages plus courts par partage des données échangées. Les courbes 15 et 16 montrent une très grande différence sur les volumes

échangés entre la version renumérotée et les autres versions. Cependant, lorsqu'il y a peu de localité, le volume total échangé varie assez peu avec le nombre de processeurs. Une partie de cette croissance faible s'explique par l'augmentation du volume nécessaire dans le calcul de terminaison, croissance comparable sur tous les exemples.

Les courbes 19 et 20 montrent des comportements du débit d'échange des processeurs proches sur les données d'origine et renumérotées aléatoirement, mais très différents sur les données renumérotées pour favoriser les accès locaux. Sur les données non optimisées, le débit demandé par processeur décroit linéairement avec le nombre de processeurs. En fait, un débit stable aurait montré une efficacité maximum de cette technique de parallélisation. Sur les données optimisées, le débit demandé augmente, ce qui s'explique par le fait que le volume émis par chaque processeur augmente légèrement du fait du calcul de terminaison alors que le temps d'exécution diminue.

Nombre de messages échangés

Chaque message échangé entraîne un appel à la bibliothèque d'échange des messages et, en général, au noyau système. Le coût de ces appels ne doit pas être négligé, les ressources processeur consommées par ces appels étant prises sur les capacités de calcul. Les figures 21 et 22 montrent une croissance globale très forte du nombre total de messages émis avec le nombre de processeurs.

Pour les calculs sur les données renumérotées aléatoirement, chaque processeur possède $p-1$ voisin. En supposant que chaque processeur échange un message avec chacun de ses voisins, le nombre total de messages varie en p^2. Mais il faut également tenir compte de l'espace alloué aux messages en mémoire. En effet, le nombre de tâches échangées reste globalement stable (voir la stabilité du volume total en 15 et 16). Lorsque les buffers d'échange sont petits, le nombre de messages émis dépend fortement du nombre de tâches, nombre qui peut être élevé mais qui varie peu. Par contre, lorsqu'ils sont suffisamment grands, à partir d'un certain nombre de processeurs, les buffers sont rarement remplis. Le comportement est alors proche de p^2.

La forme de ces courbes dépend également des conditions de l'expérimentation. Pour ces exécutions, un volume fixe de mémoire a été réservé pour les tampons d'échange. Ce volume fixe est partagé par l'ensemble des tampons qui est proportionnel au nombre de processeurs. L'augmentation du nombre de processeurs entraîne donc une diminution du volume maximum des messages, diminution qui pourrait, dans certains cas, expliquer l'augmentation du nombre de messages.

Dans le cas des données renumérotées, une partie non négligeable des messages échangés concerne uniquement le calcul de terminaison : ce sont les messages échangés avec les processeurs distants d'une puissance de deux alors que les tâches migrent uniquement vers les processeurs distants de un. Ce calcul de terminaison provoque une variation sensible à dix sept et trente trois processeurs, variation visible sur les courbes.

L'analyse des variations du débit en nombre de messages par seconde émis par chaque processeur montre des comportements très proches sur tous les jeux de données (figures 23 et 24). Ce débit semble être proportionnel au nombre de processeurs. La pente des courbes

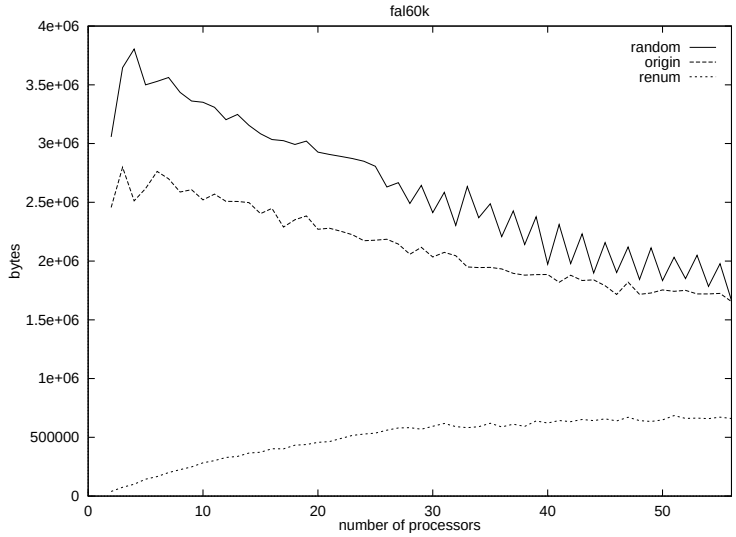

FIG. 19 - *débit d'échange (octets/seconde) par processeur sur* `fal`

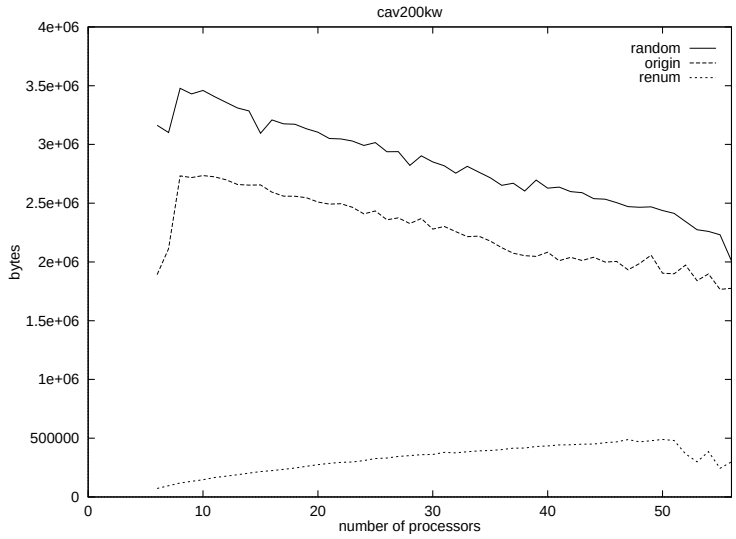

FIG. 20 - *débit d'échange (octets/seconde) par processeur sur* `cav`

correspondant au problème `cav` est plus faible, ce qui peut s'expliquer par une augmentantion du volume de calcul et un meilleur remplissage des messages.

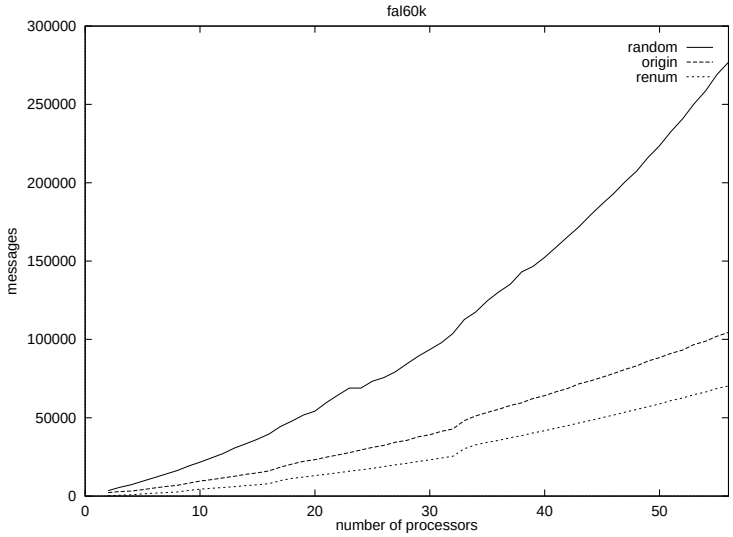

FIG. 21 - *nombre total de messages sur* `fal`

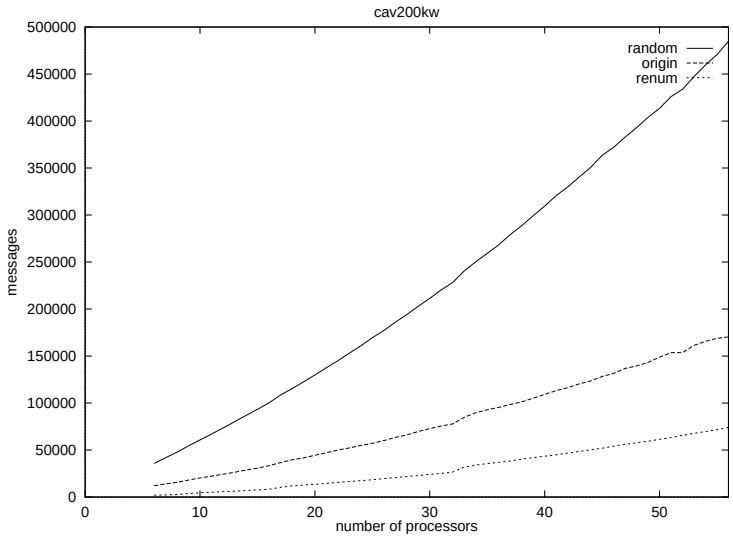

FIG. 22 - *nombre total de messages sur* `cav`

La juxtaposition des courbes de débit en octets par seconde qui décroissent linéairement avec le nombre de processeurs et des courbes de débit de messages qui, elles, croissent indique clairement que ces messages sont de plus en plus courts. La forme de ces courbes

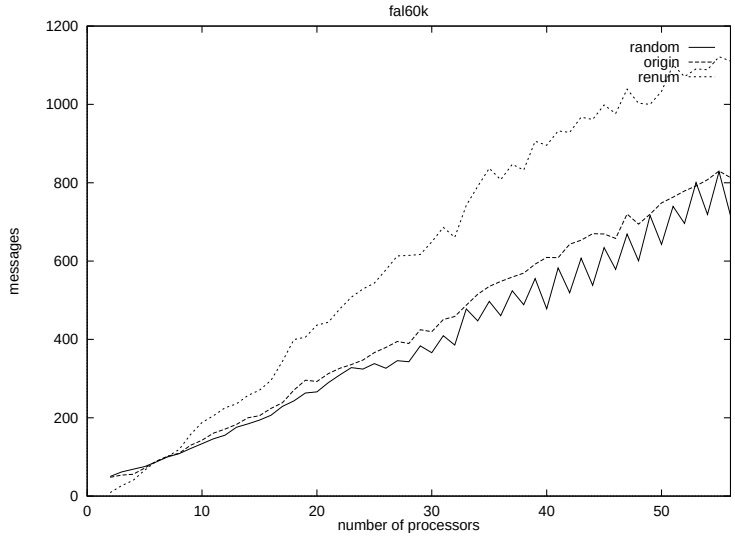

FIG. 23 - *débit d'échange (messages/seconde) par processeur sur* `fal`

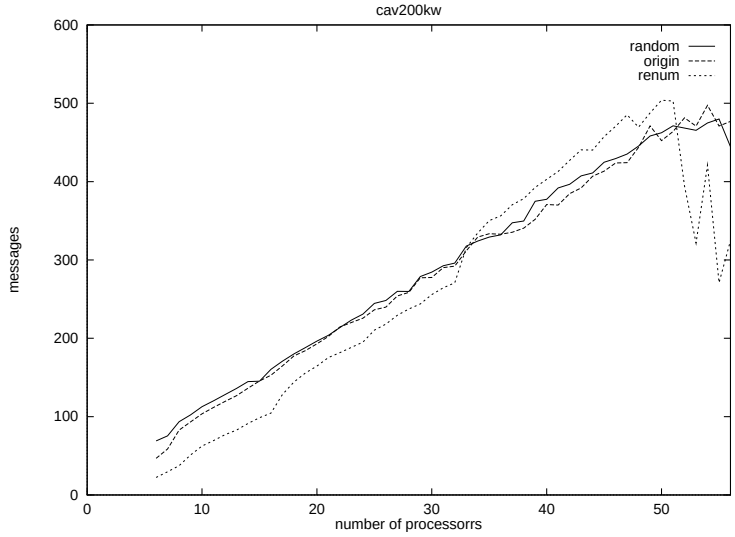

FIG. 24 - *débit d'échange (messages/seconde) par processeur sur* `cav`

donne également des indications sur les paramêtres qu'il faut optimiser sur une architecture massivement parallèle : diminuer le coût (en CPU) des émissions des messages et améliorer les débits d'échange sur les messages courts.

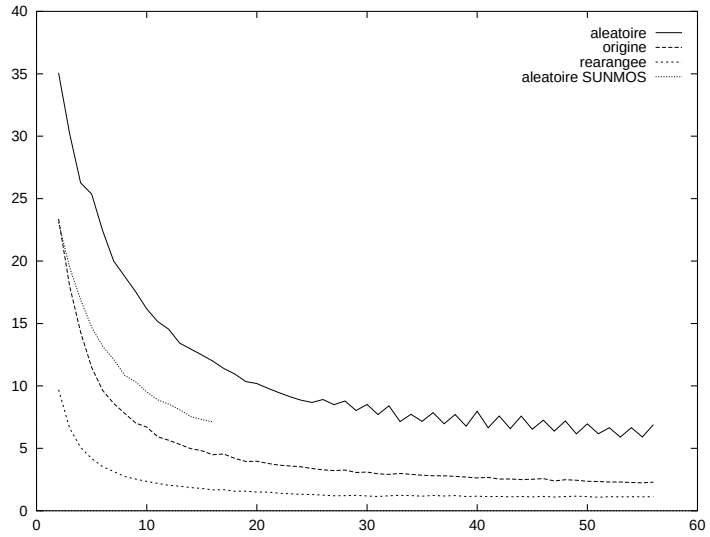

FIG. 25 - *SUNMOS*

6.7 Expérimentations futures

La technique développée n'est pas liée au système de gestion des messages présent sur la Paragon. Les codes ont été exécutés sur la même architecture en utilisant la bibliothèque de communications SUNMOS [9]. Cette bibliothèque offre des routines de communications de plus bas niveau mais plus efficaces (absence de contrôle de flot, pas, ou très peu de mémorisation des messages dans l'espace noyau). Les performances offertes par cette bibliothèque sont plus proches du potentiel de l'architecture. La figure 25 permet de comparer les temps d'exécution obtenus avec SUNMOS sur le problème `fal` renuméroté aléatoirement. Le gain de performance, de l'ordre de 30%, est appréciable.

D'autre tests ont été effectués sur une version plus récente du système d'exploitation de Paragon permettant d'utiliser un deuxième processeur pour les communications. Un gain de l'ordre de 30% semble également possible sur les codes qui nécessitent beaucoup de communications.

7 Conclusion

Le modèle d'exécution basé sur la migration de tâches de calculs développé dans ce document semble être adapté à l'exécution de codes irréguliers sur des architectures massivement parallèles à mémoires distribuées. Ce modèle peut de plus être utilisé pour l'exécution de la plupart des codes parallèles. Sa mise en œuvre automatique à partir de langages standard fait appel à une technique de génération de code spécifique qui est développée dans ce document. L'ensemble des tâches migrantes produites s'exécutent de manière totalement

asynchrone. Cet asynchronisme permet un recouvrement des calculs et des communications dans le temps mais pose des problèmes de contrôle de flot et de détection de terminaison. Les expérimentations sur un code irrégulier montrent l'efficacité du modèle même lorsque de nombreux accès non locaux aux données apparaissent lors de l'exécution. Contrairement à ce qui se passe avec d'autres modèles d'exécution, aucun phénomène d'écroulement des performances ne se produit en cas d'absence de localité dans le code.

Références

[1] Chau-Wen Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed Memory Machines.* PhD thesis, Rice University, Jan. 1993.

[2] Reinhard van Hanxleden, Ken Kennedy, and Joel Saltz. *Value-Based Distributions in Fortran D.* Technical Report CRPC-TR93365-S, Rice University, February 1994.

[3] Barbara Chapman, Piyush Mehrotra, Hans Moritsh, and Hans Zima. *Dynamic Data Distribution in Vienna Fortran.* Technical Report 93-92, ICASE/NASA LRC, December 1993.

[4] Olivier Chéron. *Pandore II: un compilateur dirigé par des données.* PhD thesis, IFSIC/Université de Rennes I, July 1993.

[5] Zakaria Lahjomri and Thierry Priol, Koan: a shared virtual memory for an iPSC/2 hupercube. In *CONPAR/VAPP*, Sept. 1992.

[6] Mounir Hahad, Thierry Priol, Jocelyne Erhel, *Irregular Loop Patterns Compilation on Distributed Shared Memory Multiprocessors*, Rapport de recherche INRIA No 2361, Septembre 1994.

[7] R. S. Nikhil, G. M. Papadopoulos, Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proc. 21th Intl. Conf. on Parallel Processing*, pp 156-167, MIT 1992.

[8] Marie Odile Bristeau, Jocelyne Erhel, Philippe Feat, Roland Glowinsky and Jacques Périaux. Solving the helmotz equation at high wave numbers on a parallel computer with a shared virtual memory. *International journal of supercomputer applications and high performance computing*, (9.1), 1995.

[9] Arthur B. Maccabe, Kevin S. McCurley, Rolf Riesen and Stephen R. Wheat, SUNMOS for the Intel Paragon: A Brief User's Guide, *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference*, pp 245-251, June 1994.

[10] High Performance Fortran Forum, *High Performance Fortran Language Specification*, Technical Report revision 1.0, Rice University, May 1993.

A Algorithme de terminaison en arbre

L'algorithme de détection en arbre des transitions de phase se déroule en $\lceil \log p \rceil$ étapes séquentielles. Dans chaque processeur, chacune de ces étapes provoque l'émission d'un message contenant les poids de messages restants pour un certain nombre de processeurs. À l'étape 0, le processeur x émet vers le processeur $x+1$ un message de terminaison contenant les poids restants des processeurs $x+1$, $x+3$, $x+5$, ..., $x+1+2k$, ... (mod p). Au cours de cette étape 0, ce processeur x doit recevoir du processeur $x-1$ un message contenant les poids restants des processeurs x, $x+2$, $x+4$, ..., $x+2k$, ... (mod p). Ces poids sont ajoutés à ceux gérés par le processeur. A la fin ce cette étape, il ne reste plus dans un processeur x que les poids restants des processeurs $x+2k$ (la moitié des poids a été traitée). À l'étape 1, le processeur x émet vers le processeur $x+2$ un message de terminaison contenant les poids restants des processeurs $x+2$, $x+6$, $x+10$, ..., $x+2+4k$, ... (mod p). Au cours de cette étape 1, ce processeur x doit recevoir du processeur $x-2$ un message contenant les poids restants des processeurs x, $x+4$, $x+8$, ..., $x+4k$, ... (mod p) et ajoute ces poids aux poids locaux. Lorsque cette étape est terminée, il ne reste plus dans chaque processeur x que les poids des processeurs $x+4k$. Un processeur x peut réaliser l'étape e dès qu'il a réalisé l'étape $e-1$ et qu'il a reçu le message correspondant à l'étape $e-1$ émis par le processeur $x-2^{e-1}$. Au cours de l'étape e, un processeur x émet vers le processeur $x+2^e$ un message contenant les poids restants à destination des processeurs dont le numéro est de la forme $x+2^e+k2^{e+1}$ (les processeurs distants de 2^{e+1} du processeur $x+2^e$).

Chaque étape divise par deux le nombre de poids gérés par chaque processeur. Ce processus se termine au bout de l'étape $\lceil \log p \rceil$ où chaque processeur ne contient plus que le poids restant pour lui-même. Ce poids peut être retiré du poids qui lui reste à recevoir. Il est possible que ce poids restant à recevoir ne soit pas nul à cet instant, tous les messages de la phase n'étant pas encore nécessairement traités. A noter que p est quelconque, pas nécessairement une puissance de deux.

Mise en œuvre

Pour appliquer l'algorithme de détection en arbre de la terminaison, chaque processeur gère les données suivantes :

- ph_j : phase courante d'émission vers le processeur $x+j$
- E_i^e : état courant de la phase i. L'état de terminaison de chaque phase est décrite par un vecteur de $\lceil \log p \rceil$ booléens. L'élément e de ce vecteur est positionné lorsque l'information concernant l'étape e de l'algorithme de terminaison est reçue pour la phase i
- P_i^j : poids de messages restant à expédier au cours de la phase i vers le processeur $x+j$. Ces poids sont initialisés à M, M étant le poids total émis par un processeur vers un même destinataire au cours d'une phase

- R_i : poids de messages restant à recevoir durant la phase i. Initialisés à $M \times (p-1)$, M étant le poids total émis par un processeur vers un même destinataire au cours d'une phase et p le nombre de processeurs

Chaque message contient une partie gestion (contrôle de flot principalement), une partie tâches migrantes et se termine par un marqueur de fin. Il y a deux types de marqueurs de fin :

- les marqueurs simples indiquent une fin du message. Un marqueur simple contient le numéro de phase de ce message. Le poids d'un message terminé par un marqueur simple est implicitement 1.
- les marqueurs de terminaison utilisés dans les changements de phases. En plus d'indiquer la fin du message, ce type de marqueur transporte l'information permettant de détecter les conditions de terminaisons : le numéro de phase i du message, le numéro d'étape e de traitement de la transition et une suite de $\lceil p/2^e \rceil$ poids.

Le traitement d'un message se termine toujours par l'interprétation de son marqueur de fin.

Traitement d'un marqueur simple de phase i

$R_i \Leftarrow R_i - 1$;
si $(R_i == 0)$ *DébutTerminaison*$(i+1)$;
Le poids (implicitement 1) du message est retiré du poids restant à recevoir pour la phase indiquée. Si le poids restant devient nul, plus aucun message ne sera reçu pour cette phase. L'appel à la routine *DébutTerminaison* amorce le traitement de la transition vers la phase suivante.

Traitement d'un marqueur de terminaison de phase i

Un tel marqueur contient un numéro d'étape e et une suite de $\lceil p/2^e \rceil$ poids $W_k, k = 0..\lceil p/2^e \rceil - 1$. Le poids W_0 concerne le processeur courant x. Le poids W_k concerne le processeur $(x + k2^e)(mod\ p)$.
$R_i \Leftarrow R_i - W_0$;
pour k **de** 1 **jusqu'à** $\lceil p/2^e \rceil - 1$ {
  $j = k2^e$;
  $P_i^j \Leftarrow P_i^j + W_k$;
}
$E_i^e \Leftarrow OK$;
ÉtapeTerminaison$(e+1, i)$;
si $(R_i == 0)$ *DébutTerminaison*$(i+1)$;
Le poids W_0 est retiré du poids restant à recevoir pour cette phase. Si ce poids restant devient nul, le dernier message de cette phase a été traité, le calcul de la terminaison peut

être déclenché pour la phase suivante. Les autres poids restants contenus dans le marqueur sont rajoutés aux poids restants gérés localement. La prise en compte de ce marqueur est mémorisée dans l'état de la phase. L'exécution de la routine $\acute{E}tapeTerminaison(e+1, i)$ lance l'étape suivante si les conditions sont réunies. Les messages contenant ces marqueurs de terminaison peuvent être reçus dans un ordre quelconque.

Exécution de l'étape e dans une phase i

$\acute{E}tapeTerminaison(e, i)$ {
  Exécutable = $E_i^k == \text{OK}, \forall k \in [0, e]$;
  si (Exécutable) {
    $MesTerminaison(e+1,\ i,\ x+2^e,\ P_i^{x+2^e},\ P_i^{x+2^e+2^{e+1}},\ \ldots,$
    $P_i^{x+2^e+k2^{e+1}})$;
    $ph_{x+2^e} \Leftarrow i+1$;
    si $(e < \lceil \log p \rceil$ $\acute{E}tapeTerminaison(e+1, i)$;
  }
}

L'étape e est exécutable lorsque tous les messages de terminaison des étapes précédentes ont été reçus. La routine *MesTerminaison* pour une étape e place une marque de ternimaison dans le buffer à destination du processeur $x + 2^e$ et expédie ce message. Ce message est le dernier émis vers ce processeur dans la phase i.

Exécution de la première étape 0 dans une phase i

La première étape de terminaison d'une phase est déclenchée, soit lorsque toutes les tâches initiales d'une construction migrante ont été créées pour la première phase, soit lorsque le dernier message d'une phase a été traité.

$DébutTerminaison(i)$ {
  $E_i^0 \Leftarrow OK$;
  pour_tout k **de** 1 **jusqu_à** p-1 {
    si $(NonPuissance2(k))$ {
      si $(BufferOccupé(k))$ {
        $ÉmettreBuffer(k)$;
        $P_i^k = P_i^k - 1$;
      }
    $ph_k = ph_k + 1$;
    }
  $ÉtapeTerminaison(1, i)$;
  }

Le traitement d'une transition de phase par un processeur entraîne l'émission de $\lceil \log p \rceil$ messages vers les $\lceil \log p \rceil$ processeurs distants d'une puissance de deux. Pour les $p - \lceil \log p \rceil$ restants (répondant au test $(NonPuissance2(k))$, il n'y a émission de message que si une

tâche doit y migrer au cours de la phase. La routine *BufferOccupé*(k) teste l'état du tampon de messages à destination du processeur $x + k$. Si ce buffer est vide, il reste en place pour la phase suivante. Sinon, il est expédié avec une marque simple de terminaison par l'appel à *ÉmettreBuffer*. Le poids restant P_i^k pour ce processeur est mis à jour pour tenir compte de cette expédition.

Commentaires

Le nombre total de messages émis à chaque transition de phase est au minimum $p\lceil \log p \rceil$.

L'algorithme de traitement des transitions se déroule en parallèle avec le traitement des tâches migrantes: la fin de l'exécution de l'algorithme pour une phase déclenche immédiatement l'algorithme de transition de la phase suivante. A un instant donné, les messages émis ne portent pas toujours le même numéro de phase. Par exemple, à la suite de la première étape de l'algorithme, les messages émis vers les processeurs distants d'une puissance de deux pourtent un certain numéro n alors que les messages émis vers les autres processeurs portent le numéro de phase suivante $n + 1$. Ce décalage peut provoquer un recul du numéro de phase associé à une tâche migrante: elle peut arriver sur un certain processeur dans un message portant le numéro de phase f puis migrer dans un message portant le numéro $f - 1$. Mais cette possibilité ne transgresse pas les règles sur les migrations définies en 4.2. A tout instant les messages émis par l'ensemble des processeurs appartiennent à deux phases voisines.

B Génération du code de la boucle Fortran

B.1 Transformation

Le programme original subit un certain nombre de transformations classiques avant la génération de code. Dans le sous-programme qui nous intéresse, certains éléments de tableaux sont lus plusieurs fois. Pour diminuer le nombre des accès à la mémoire, ces sous-expressions communes sont factorisées. Ensuite, les expressions complexes sont découpées jusqu'à ce qu'il ne reste que quatre types d'instructions simples :

- les instructions de lecture en mémoire distribuée. La donnée lue est rangée dans une variable locale. L'indice éventuel du tableau doit être une variable locale.
- les instructions d'écriture en mémoire distribuée. La donnée doit être dans une variable locale. L'indice éventuel doit également être une valeur locale.
- les instructions de mise à jour d'une variable en mémoire distribuée. Ce type d'instruction contient une lecture et une écriture à une même adresse. La valeur écrite est calculée à partir de la valeur lue et de valeurs locales éventuelles.
- les instructions de calcul. Les instructions qui ne sont ni des lecture ni des écritures en mémoire distribuée ne font référence qu'à des variables locales.

La figure 26 contient la nouvelle formulation de la première boucle du sous-programme à la suite de ces deux transformations. Chaque instruction d'affectation de ce corps de boucle est un nœud du graphe de dépendance de la figure 27 considéré dans la génération de code. Dans cet exemple, tous les arcs du graphe sont de type *flot* (correspondent à un échange de valeur). Ce graphe possède trois sorties par les nœuds 19, 20 et 21.

B.2 Prise en compte des directives d'alignement/distribution

Pour ce code, nous avons considéré les directives de type HPF [10] suivantes :

```
!HPF$ Template triangles(nt)
!HPF$ Template nodes(ns)
!HPF$ Align me(*,:) with triangles(:)
!HPF$ Align (:) with nodes(:) :: ynm1, yn, ynp1, v1
!HPF$ Align cq(*,:) with triangles(:)
!HPF$ Distribute (Block) onto Processors :: triangles, nodes
```

Ces directives provoquent une distribution par blocs des directions indicées par des indices de nœuds ou de triangles. De plus, les directives d'alignement sont telles que les éléments de tableaux indicés par un même numéro de nœud ou de triangle sont localisés dans la même mémoire. Après traitement de ces directives, les groupes d'alignement suivants de

```
      do j=1,nt
1        me1= me(1,j)
2        me2= me(2,j)
3        me3= me(3,j)
4        cq1= cq(1,j)
5        cq2= cq(2,j)
6        cq3= cq(3,j)
7        cq4= cq(4,j)
8        cq5= cq(5,j)
9        cq6= cq(6,j)
10       yn1= yn(me1)
11       yn2= yn(me2)
12       yn3= yn(me3)
13       gxn= yn1*cq1 + yn2*cq3 + yn3*cq5
14       gyn= yn1*cq2 + yn2*cq4 + yn3*cq6
15       Aire= aire(j)
16       up1= Aire*(gxn*cq1+gyn*cq2)
17       up2= Aire*(gxn*cq3+gyn*cq4)
18       up3= Aire*(gxn*cq5+gyn*cq6)
19       ynp1(me1)+= up1
20       ynp1(me2)+= up2
21       ynp1(me3)+= up3
      end do
```

FIG. 26 - *première boucle après réécriture*

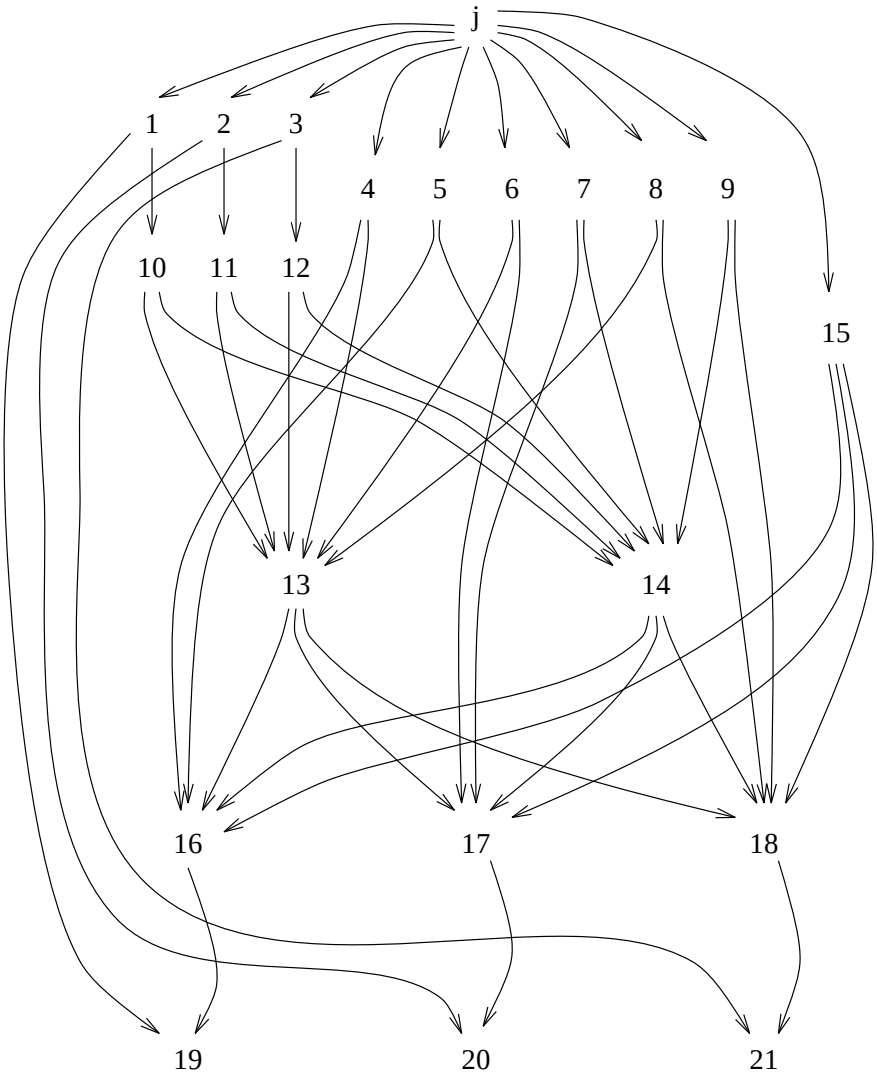

FIG. 27 - *graphe flot/dépendance de l'exemple*

la figure 28 sont formés sur les numéros des instructions du corps de boucle. Leurs ensembles incompatibles sont vides.

$$\begin{array}{ll} A_a & \{1, 2, 3, 4, 5, 6, 7, 8, 9, 15\} \\ A_b & \{10, 19\} \\ A_c & \{11, 20\} \\ A_d & \{12, 21\} \end{array}$$

Le premier groupe d'alignement référencé dans ce corps de boucle est le groupe A_a. Il suffit donc d'utiliser la distribution par bloc des éléments de ce groupe comme distribution des itérations de la boucle pour éviter la première migration (voir paragraphe 3.6). Ce choix pour la distribution des itérations se traduit par l'initialisation du groupe d'alignement $LOCAL$ par les éléments de A_a dans l'algorithme de génération du paragraphe 3.

Les codes qui suivent ont été obtenus en appliquant systématiquement l'algorithme de génération automatique avec calcul dynamique des groupes d'alignement et traitement des sorties multiples. Les instructions produites sont, soit des instructions locales (n'accédant que des variables locales), soit des instructions d'accès aux données distribuées (lecture, écriture ou mise à jour), soit des instructions de test de localité, soit des instructions de migration.

Les figures 29, 30, 31, 32, et 33 montrent le code généré. Certaines instructions de ces figures sont précédées par l'état des groupes d'alignement avant la génération. Chaque groupe est composé de deux parties séparées par le signe / : les nœuds du groupe et les nœuds de l'ensemble incompatible associé. Le premier élément de la liste est toujours composé du groupe local (nœuds testés locaux) et de son ensemble incompatible (nœuds testés non locaux).

Les vingt cinq premières instructions de la séquence produite correspondent au cas où tous les accès sont locaux. Ce bloc contient trois tests de localité. À part ces tests, c'est le code produit pour une architecture séquentielle. Chaque test de localité de ce premier bloc porte sur un groupe d'alignement différent.

Les optimisations issues du calcul des groupes d'alignement sont visibles dans ce code produit. La gestion dynamique des groupes d'alignement permet, par exemple, d'éviter tout test de localité pour accéder aux variables `ynp1(me1)`, `ynp1(me2)` et `ynp1(me3)`. Par exemple, le test de localité sur la variable `yn(me3)` provoque un transfert à l'instruction `e18` en cas d'échec. En `e18`, deux groupes d'alignement ont été formés: le groupe local issu de la fusion des groupes d'alignement de `ynp1(me1)` et de `ynp1(me2)` et le groupe d'alignement de `yn(me3)` qui contient également `ynp1(me3)`. Suite à l'échec du test de localité sur `yn(me3)`, les nœuds de son groupe (`yn(me3)` et `ynp1(me3)`) sont rajoutés à l'ensemble non local en `e18`. Pour respecter la cohérence des groupes, les nœuds du groupe local ont été ajoutés à l'ensemble incompatible du groupe de `yn(me3)`.

La connaissance de ces deux groupes permet, après migration vers le propriétaire de `yn(me3)`, de générer la mise à jour de `ynp1(me3)` sans avoir à vérifier que cette adresse est locale, puis de générer directement l'ordre de migration vers le propriétaire de `ynp1(me1)`, toujours sans test de localité. En `e37`, cible de cette migration, les deux variables `ynp1(me1)`

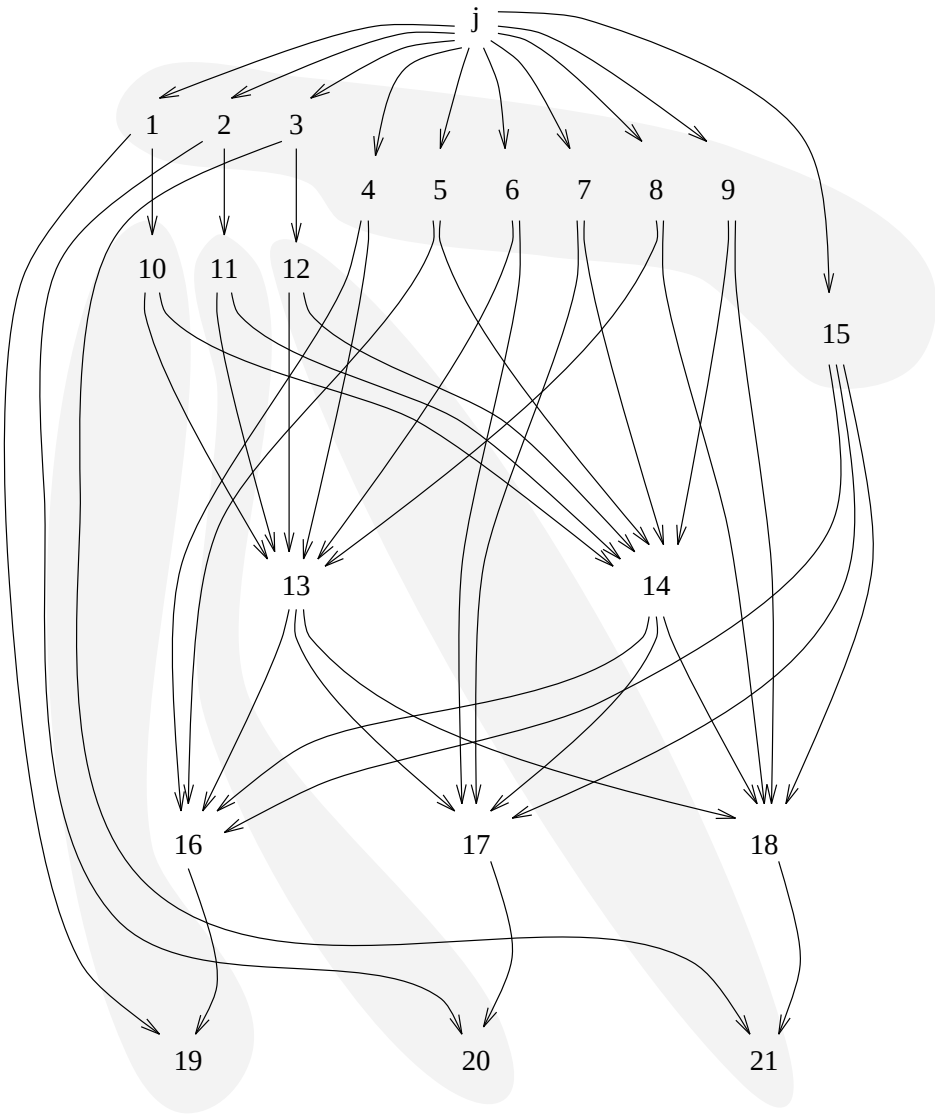

FIG. 28 - *les quatre groupes d'alignement*

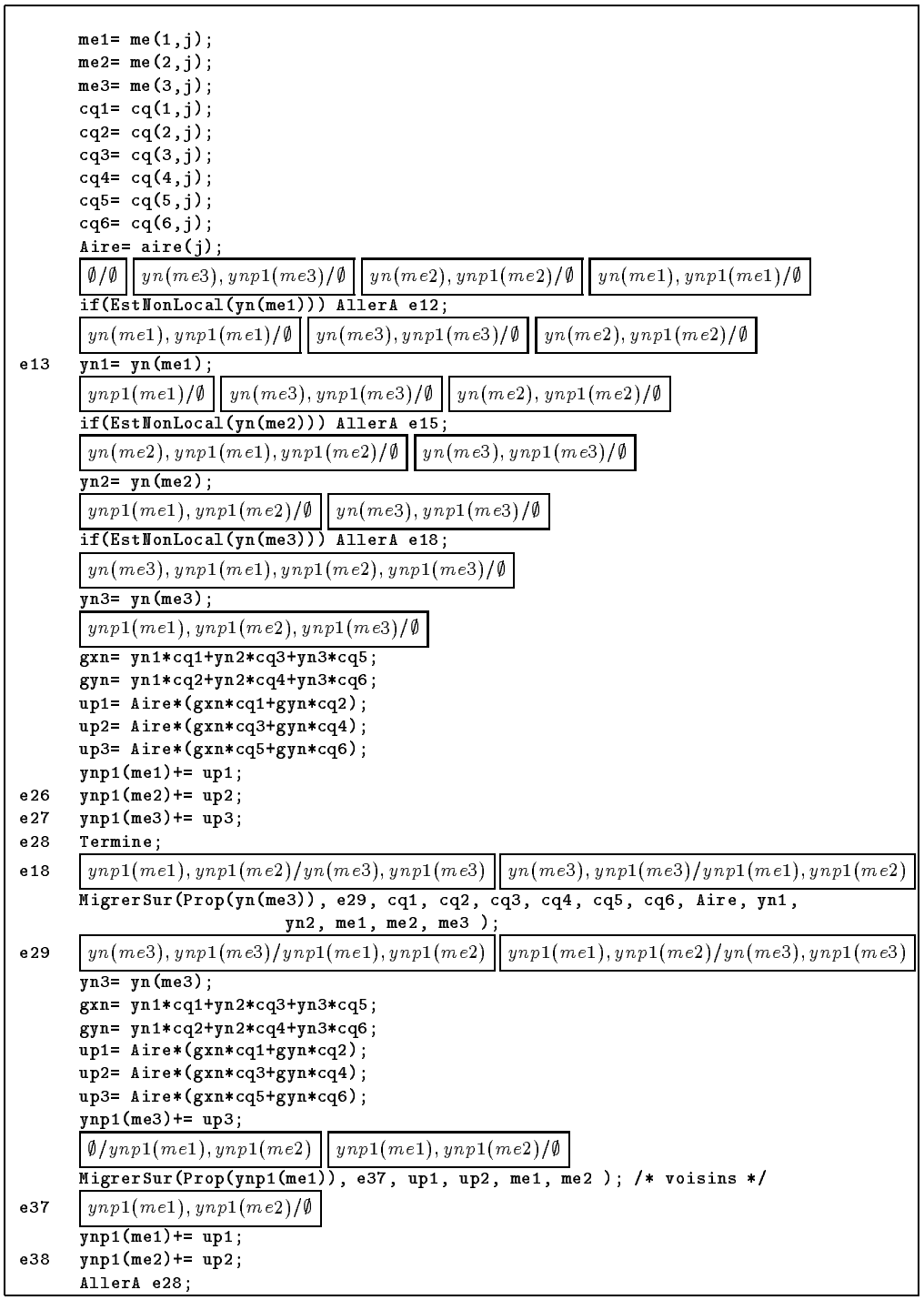

FIG. 29 - *code généré, partie 1*

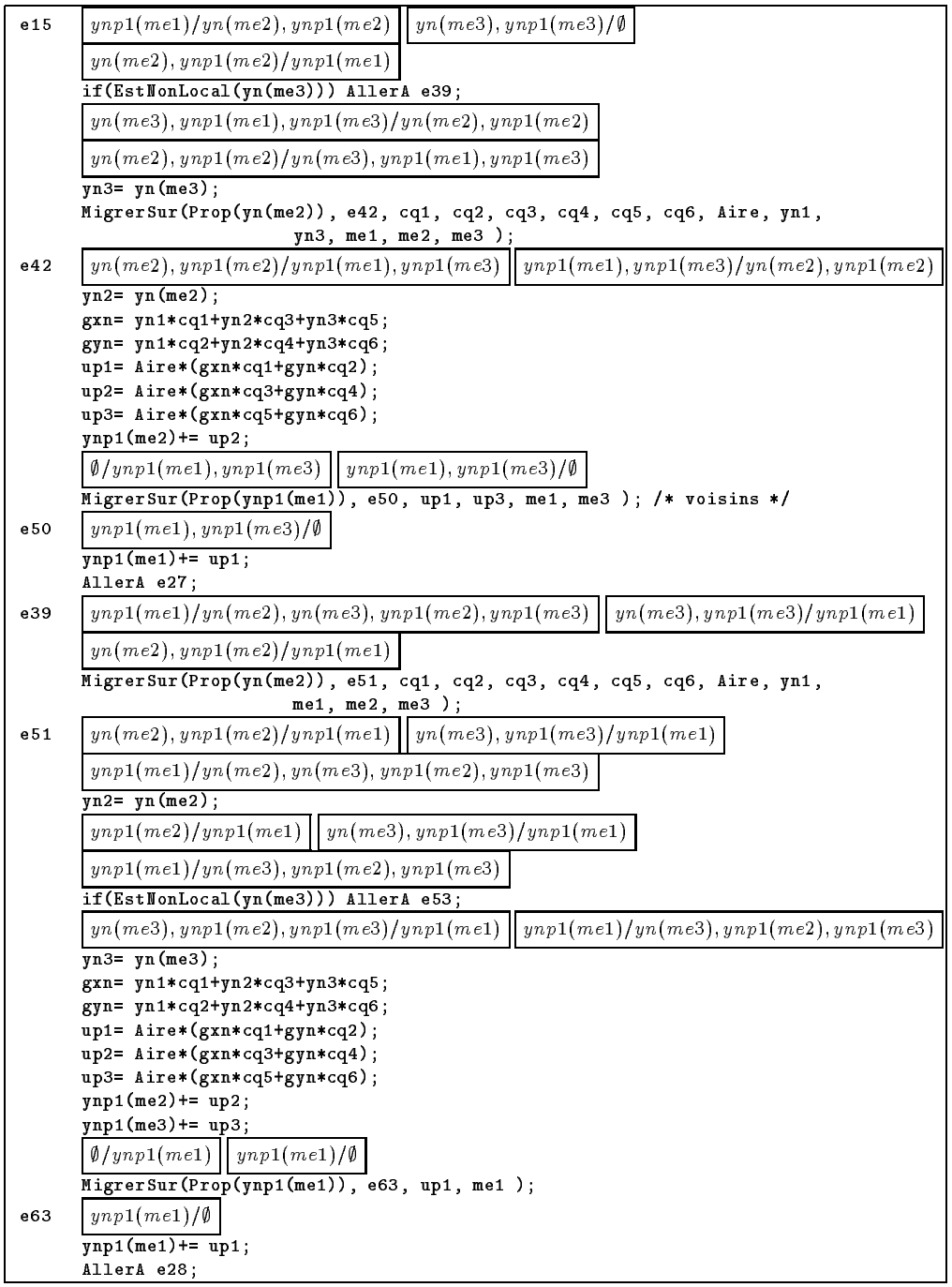

FIG. 30 - *code généré, partie 2*

```
e53   ynp1(me2)/yn(me3),ynp1(me1),ynp1(me3) | yn(me3),ynp1(me3)/ynp1(me1),ynp1(me2)
      ynp1(me1)/yn(me3),ynp1(me2),ynp1(me3)
      MigrerSur(Prop(yn(me3)), e64, cq1, cq2, cq3, cq4, cq5, cq6, Aire, yn1,
                          yn2, me1, me2, me3 );
e64   yn(me3),ynp1(me3)/ynp1(me1),ynp1(me2) | ynp1(me2)/yn(me3),ynp1(me1),ynp1(me3)
      ynp1(me1)/yn(me3),ynp1(me2),ynp1(me3)
      yn3= yn(me3);
      gxn= yn1*cq1+yn2*cq3+yn3*cq5;
      gyn= yn1*cq2+yn2*cq4+yn3*cq6;
      up1= Aire*(gxn*cq1+gyn*cq2);
      up2= Aire*(gxn*cq3+gyn*cq4);
      up3= Aire*(gxn*cq5+gyn*cq6);
      ynp1(me3)+= up3;
      ∅/ynp1(me1),ynp1(me2) | ynp1(me2)/ynp1(me1) | ynp1(me1)/ynp1(me2)
      DetacheSur(Prop(ynp1(me1)), e63, up1, me1 );
e72   ∅/ynp1(me2) | ynp1(me2)/∅
      MigrerSur(Prop(ynp1(me2)), e38, up2, me2 );
e12   ∅/yn(me1),ynp1(me1) | yn(me3),ynp1(me3)/∅ | yn(me2),ynp1(me2)/∅
      yn(me1),ynp1(me1)/∅
      if(EstNonLocal(yn(me2))) AllerA e73;
      yn(me2),ynp1(me2)/yn(me1),ynp1(me1) | yn(me3),ynp1(me3)/∅
      yn(me1),ynp1(me1)/yn(me2),ynp1(me2)
      yn2= yn(me2);
      ynp1(me2)/yn(me1),ynp1(me1) | yn(me3),ynp1(me3)/∅
      yn(me1),ynp1(me1)/ynp1(me2)
      if(EstNonLocal(yn(me3))) AllerA e76;
      yn(me3),ynp1(me2),ynp1(me3)/yn(me1),ynp1(me1)
      yn(me1),ynp1(me1)/yn(me3),ynp1(me2),ynp1(me3)
      yn3= yn(me3);
      ynp1(me2),ynp1(me3)/yn(me1),ynp1(me1) | yn(me1),ynp1(me1)/ynp1(me2),ynp1(me3)
      MigrerSur(Prop(yn(me1)), e79, cq1, cq2, cq3, cq4, cq5, cq6, Aire, yn2,
                          yn3, me1, me2, me3 );
e79   yn(me1),ynp1(me1)/ynp1(me2),ynp1(me3) | ynp1(me2),ynp1(me3)/yn(me1),ynp1(me1)
      yn1= yn(me1);
      gxn= yn1*cq1+yn2*cq3+yn3*cq5;
      gyn= yn1*cq2+yn2*cq4+yn3*cq6;
      up1= Aire*(gxn*cq1+gyn*cq2);
      up2= Aire*(gxn*cq3+gyn*cq4);
      up3= Aire*(gxn*cq5+gyn*cq6);
      ynp1(me1)+= up1;
      ∅/ynp1(me2),ynp1(me3) | ynp1(me2),ynp1(me3)/∅
      MigrerSur(Prop(ynp1(me2)), e26, up2, up3, me2, me3 ); /* voisins */
e76   ynp1(me2)/yn(me1),yn(me3),ynp1(me1),ynp1(me3) | yn(me3),ynp1(me3)/ynp1(me2)
      yn(me1),ynp1(me1)/ynp1(me2)
      MigrerSur(Prop(yn(me1)), e87, cq1, cq2, cq3, cq4, cq5, cq6, Aire, yn2,
                          me1, me2, me3 );
```

FIG. 31 - *code généré, partie 3*

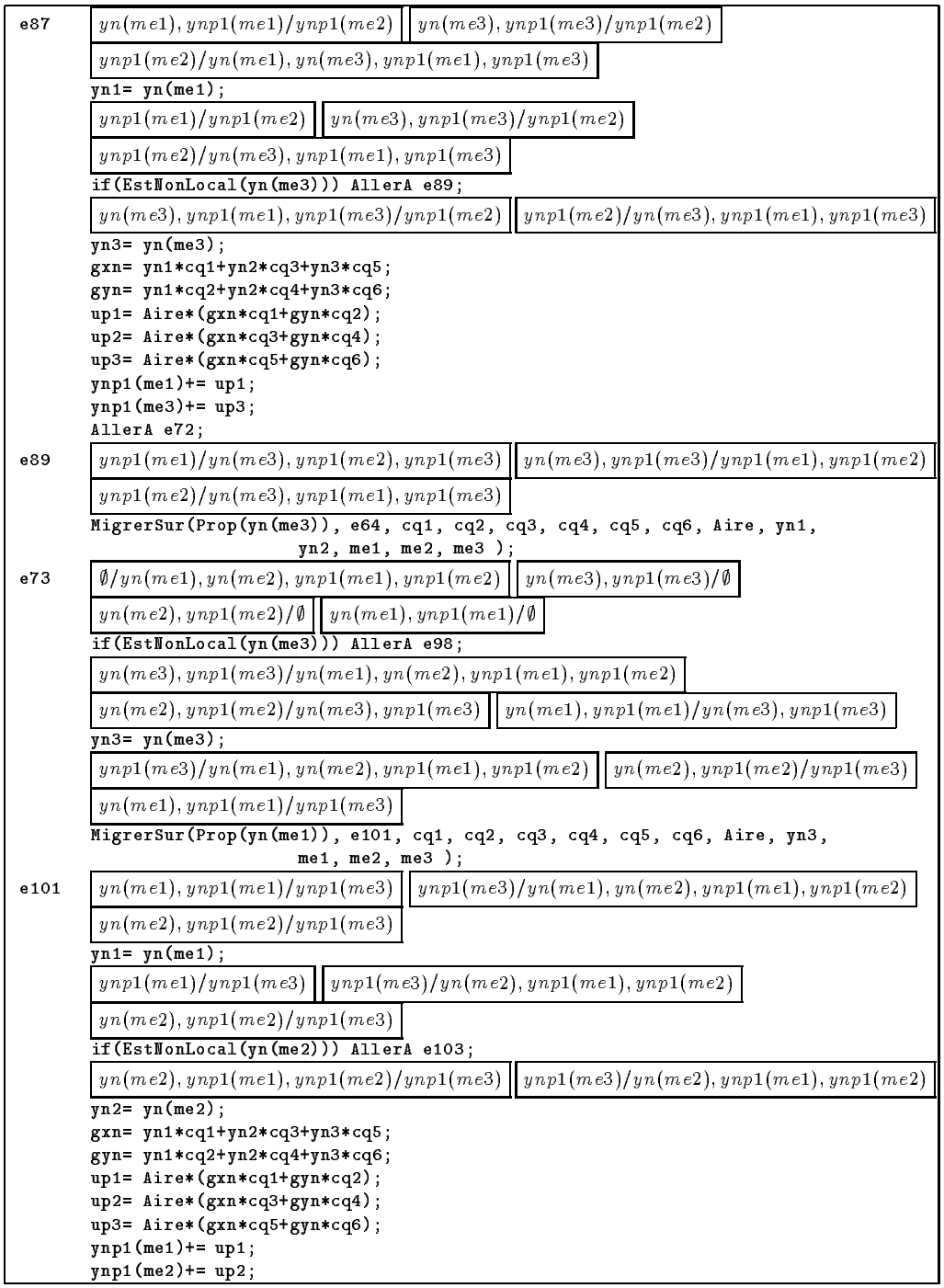

FIG. 32 - *code généré, partie 4*

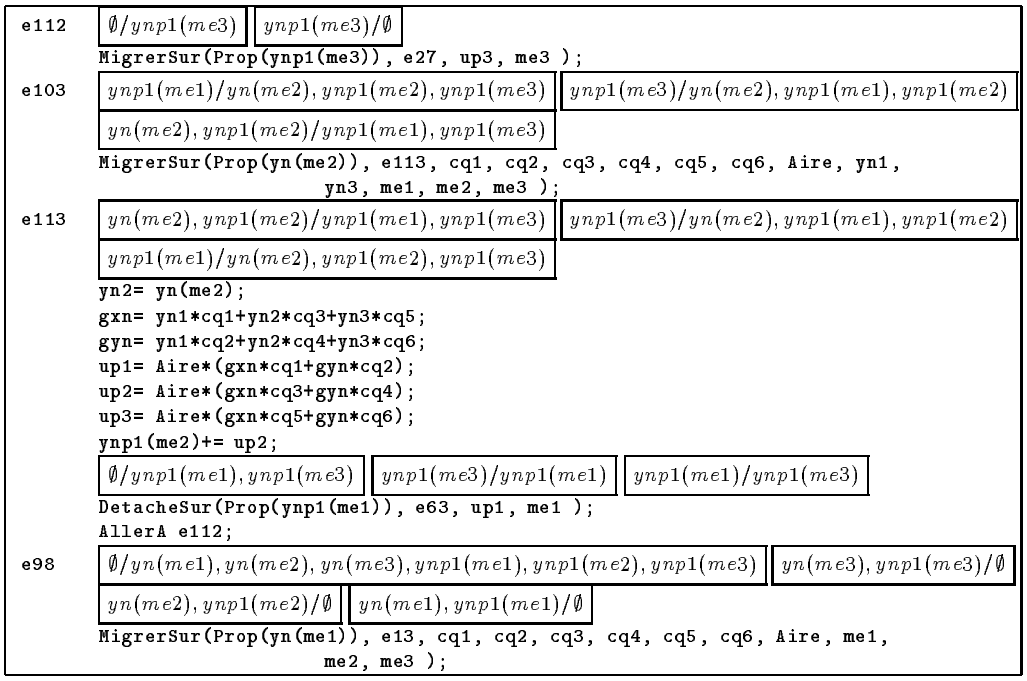

FIG. 33 - *code généré, dernière partie*

et `ynp1(me2)` sont mises à jour, les groupes d'alignement de ces deux variables ayant été fusionnés avant la migration sur le groupe de `yn(me3)`.

Ces deux mises à jour correspondent à deux sorties distinctes du graphe de flot/dépendance. Mais, les nœuds correspondant à ces sorties sont alignés et la séparation en sous-tâche n'est pas provoquée.

Dans le bloc d'instructions de l'étiquette `e64`, les nœuds `ynp1(me1)` et `ynp1(me2)` sont incompatibles. Il y a, dans ce cas, découpage en deux sous-tâches. La première sous-tâche exécute les instructions correspondant à la sortie de `ynp1(me1)`. Cette sous-tâche est crée par l'instruction `DetacheSur(Prop(ynp1(me1)), e63, up1, me1 );`.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399