Environnement pour la compilation dirigée par les données : supports d'exécution et expérimentations

Yves Mahéo

N^o d'ordre : 1404

THÈSE

Présentée devant

l'Université de Rennes 1

Institut de Formation Supérieure en Informatique et Communication

Pour obtenir

Le Titre de Docteur de l'Université de Rennes 1
Mention INFORMATIQUE

par

Yves Mahéo

Environnement pour la compilation dirigée par les données : supports d'exécution et expérimentations

Soutenue le 4 juillet 1995 devant la Commission d'examen

M.	Jean-Pierre	Banâtre	Président
M.	Jean-Luc	Dekeyser	Rapporteurs
M^{me}	Brigitte	Plateau	
M^{me}	Françoise	André	Examinateurs
M.	Jacques	Briat	
M.	Jean-Louis	Pazat	
M.	Thierry	Priol	

Remerciements

Je remercie M. Jean-Pierre Banâtre*, professeur et directeur de l'IRISA, qui m'a fait l'honneur de présider ce jury.*

Je remercie M^me^ Françoise André*, professeur à l'université de Rennes 1, d'avoir dirigé mon travail de thèse, et M. Jean-Louis* Pazat*, maître de conférences à l'INSA de Rennes, pour m'avoir soutenu, orienté et conseillé tout au long de ce travail.*

Je remercie M^me^ Brigitte Plateau, professeur à l'Institut National Polytechnique de Grenoble, et M. Jean-Luc Dekeyser*, professeur à l'Université des Sciences et Techniques de Lille, d'avoir bien voulu accepter la charge de rapporteurs.*

Je remercie M. Jacques Briat, maître de conférence à l'Université Joseph Fourier de Grenoble et M. Thierry Priol*, chargé de recherche INRIA, d'avoir accepté de juger ce travail.*

Je remercie enfin tous les membres du projet PAMPA (Programmation des architectures parallèles réparties : fondements et méthodologie) pour leur aide amicale permanente.

Table des matières

Introduction

Face aux besoins sans cesse croissants en puissance de calcul dans des domaines scientifiques de plus en plus variés tels que la météorologie, l'aérodynamique, ou la chimie, il s'est avéré que les progrès technologiques des ordinateurs mono-processeur ne suffiraient pas. Dans le même temps, des avancées significatives ont été faites dans le développement de réseaux de communication à haut débit. C'est donc sur la voie du parallélisme que se portent de nombreux efforts dans le domaine de l'architecture des ordinateurs.

Dans ce contexte sont apparues depuis quelques années les architectures parallèles à mémoire distribuées (APMD). Celles-ci sont constituées de multiples nœuds reliés par un réseau d'interconnexion de topologie fixe ou variable. Chaque nœud comporte un processeur accédant à une mémoire locale et échangeant des messages avec les autres nœuds via le réseau de communication.

Nous nous intéresserons plus particulièrement à la classe des APMD qualifiées de MIMD (*Multiple Instruction, Multiple Data* [46]) caractérisées par l'asynchronisme de leur comportement : chaque nœud possède sa propre unité de contrôle qui gère un flot d'instruction indépendamment des autres nœuds, contrairement aux machines SIMD (*Single Instruction, Multiple Data*) dans lesquelles tous les nœuds, généralement moins puissants, sont régis par un dispositif de contrôle unique.

Les APMD de type MIMD, que nous nommerons simplement APMD dans la suite de ce document, ont pour principal atout leur extensibilité. L'adjonction de nouveaux nœuds étant relativement aisée, les APMD peuvent comporter un grand nombre de processeurs (jusqu'à plusieurs centaines). De telles architectures massivement parallèles sont déjà construites ; elles sont dotées d'une puissance de calcul théorique considérable, l'objectif du Téraflop (billion d'instructions en virgule flottante par seconde) devant être atteint dans un avenir proche. Le coût de leur développement reste à l'heure actuelle élevé malgré l'utilisation de composants de base répandus. Une alternative de moindre coût est apparue par le biais de l'utilisation de stations de travail puissantes interconnectées par un réseau à haut débit spécialisé. Ces ensembles de stations constituent des APMD à part entière.

Le domaine d'applications principalement visé par les APMD est celui du calcul scientifique. Ces applications effectuent de grandes quantités de calculs sur des

structures de données importantes et souvent régulières, essentiellement des tableaux. Elles possèdent généralement une bonne localité des accès (un accès à un élément de tableau est fréquemment suivi d'un accès à un élément voisin) et un parallélisme potentiel à gros grain (on peut exécuter en parallèles des groupes entiers d'instructions), permettant d'aboutir à des programmes qui alternent des phases de calculs intensifs locaux et des phases de communication. Ce type de programmes est adapté aux caractéristiques matérielles des APMD sur lesquelles ils peuvent être exécutés efficacement.

Bien que plusieurs APMD soient proposées par différents constructeurs depuis quelques années, leur diffusion dans les milieux scientifiques est toujours restreinte. Bien qu'ayant la volonté de disposer d'une grande puissance de calcul, les scientifiques sont réticents devant l'effort de programmation à fournir au vu des environnements logiciels offerts avec les APMD. Les outils proposés ne permettent pas le développement rapide d'applications qui soient portables et qui tirent parti du potentiel de puissance de calcul des APMD. En particulier, ces outils répondent mal au besoin de réutilisation des nombreuses applications séquentielles existantes.

Le développement d'environnements de programmation des APMD fait l'objet de nombreuses recherches. C'est le thème choisi par l'équipe Pampa au sein de laquelle mon travail de thèse a été effectué. J'ai plus particulièrement participé au projet PANDORE, démarré en 1988, dont l'objectif est la construction d'un système de compilation pour APMD fondé sur la production de code parallèle à partir d'un programme séquentiel dans lequel une distribution des données a été spécifiée.

Après une première maquette développée par Henry Thomas [95], le système de compilation a été entièrement réécrit. Le développement du compilateur, appliquant un schéma de compilation de base, a fait l'objet de la thèse d'Olivier Chéron [37]. J'ai pour ma part défini et mis en œuvre le support d'exécution associé à ce compilateur, permettant l'exécution des programmes PANDORE sur l'hypercube Intel iPSC/2. Dans le cadre du développement de cet exécutif, l'intégration de techniques d'évaluation de performances adaptées au contexte de la compilation par distribution de données a été étudiée.

Par ailleurs, la conception d'un schéma de compilation optimisé a fait l'objet de la thèse de Marc Le Fur [79]. Conjointement à ce travail, et afin de tirer parti des optimisations du nouveau schéma, j'ai conçu et mis en œuvre un mode de gestion des données distribuées original et de nouvelles techniques d'optimisations des communications à l'exécution. Un exécutif plus complet et plus portable à été mis au point pour prendre en compte ces changements.

Afin d'évaluer le système de compilation obtenu, j'ai procédé à un certain nombre d'expérimentations sur des programmes de test ainsi que sur une application réelle.

Enfin, j'ai étudié la production de code par distribution de données pour une APMD disposant d'un mécanisme de mémoire virtuelle partagée (MVP). Une version

modifiée du système de compilation PANDORE a été développée et a pu faire l'objet d'expérimentations.

Plan du document

Le chapitre 1 situera l'approche de la compilation par distribution de données dans le contexte de la programmation des APMD. Nous insisterons sur la conception d'environnements de programmation fondés sur cette approche, lesquels constituent le cadre de ce travail de thèse.

Le chapitre 2 présentera un ensemble d'opérations adaptées à la compilation par distribution de données, ensemble formant ce que nous appellerons une machine abstraite. Ces opérations permettent de définir un cadre général pour la description des schémas de compilation. Les points clés de la mise en œuvre de ces opérations dans le compilateur et dans le support d'exécution seront abordés, en détaillant plus particulièrement les problèmes de gestion des données distribuées et les optimisations des communications à l'exécution. La description de travaux d'autres projets de recherche concernant ces opérations clôtureront le chapitre.

Le chapitre 3 décrira les travaux réalisés dans le système de compilation PANDORE. Deux mises en œuvre de la machine abstraite PANDORE seront présentées, la première reposant sur un support d'exécution utilisant la communication par messages et la deuxième exploitant les mécanismes d'une MVP.

Le chapitre 4 développera le travail d'expérimentation de l'environnement PANDORE. Nous y aborderons également les techniques d'évaluation de performances pour la compilation par distribution de données et décrirons l'incorporation et l'utilisation d'outils de mesure de performance dans le système PANDORE.

Le chapitre 5 fera le bilan des travaux effectués et abordera les perspectives offertes.

Chapitre 1

Contexte de l'étude

1.1 Programmation des APMD

1.1.1 Difficulté de programmation des APMD

La programmation directe des APMD est une tâche difficile, le programmeur doit en effet gérer à la fois la répartition des données dans les mémoires locales et la répartition des calculs sur l'ensemble des processeurs. Les applications doivent être conçues en termes de processus séquentiels coopérant par passage de messages.

En plus de la difficulté de conception, de nombreux problèmes de mise en œuvre restent à la charge du programmeur. Les applications dont les APMD sont la cible et en particulier les applications de calcul scientifique manipulent de grandes structures de données qu'il est nécessaire de découper. L'adressage des données ainsi distribuées est compliqué par le fait que l'on ne manipule plus une seule entité mais plusieurs. Le programmeur doit en outre gérer l'allocation de la mémoire nécessaire à la réception des messages. Il doit également, s'il veut obtenir de bonnes performances, tenir compte de paramètres inhérents à l'architecture cible et de son système d'exploitation tels que la latence des messages.

1.1.2 Supports de programmation

1.1.2.1 Bibliothèque de communication

Les environnements logiciels actuellement proposés avec les APMD consistent essentiellement en des bibliothèques de communication qui, associées à des langages séquentiels tels que Fortran ou C, permettent le développement d'applications distribuées sous la forme de processus s'échangeant des messages. Ces bibliothèques

mettent en œuvre une grande variété de types de communications (bloquantes, non bloquantes, tamponnées,...). Afin de résoudre les problèmes de portabilité engendrés par la profusion de ces types de communication et des interfaces associées, des travaux sont menés afin de proposer des bibliothèques de communication offrant des services généraux [18, 89], la disponibilité de telles bibliothèques sur de nombreuses plate-formes devant faciliter le portage des applications.

Le développement d'un code spécifique à chaque processus étant difficilement envisageable, on adopte en général le modèle de programmation SPMD (*Single Program Multiple Data* [39]) dans lequel un code semblable est exécuté par l'ensemble des processus qui agissent sur des données différentes, le comportement de chaque processus se distinguant des autres suivant son identité.

Malgré cela, les programmes obtenus sont difficiles à mettre au point, l'asynchronisme des processus et le non-déterminisme inhérent à leur exécution rendant complexe le contrôle de la correction et de la performance.

1.1.2.2 Mémoire virtuelle partagée

Afin de masquer l'aspect distribué des données, et donc de faciliter la tâche du programmeur d'APMD, un nouveau support de programmation a été proposé. Il consiste à offrir une mémoire virtuelle partagée (MVP) accessible à l'ensemble des processeurs. La MVP permet d'avoir une vision globale de la mémoire. Il s'agit d'une extension du concept de mémoire virtuelle mono-processeur au cas multi-processeur. La MVP s'appuie sur les services de mémoire virtuelle pour la transformation des adresses virtuelles en adresses physiques et sur le réseau de communication pour l'échange de données entre mémoires locales. Lorsqu'un processeur désire accéder à un mot mémoire qui n'est pas présent dans sa mémoire locale, la MVP se charge de le rapatrier. Afin d'exploiter la localité spatiale et temporelle des programmes, l'espace virtuel est découpé en pages, une page constituant l'unité d'allocation mémoire et de communication. L'existence de copies de pages dans différentes mémoires locales induit des problèmes de cohérence lorsque les données sont modifiées. Le maintien de la cohérence est alors assuré par différents protocoles internes à la MVP .

La réalisation d'une MVP peut être effectuée en modifiant le système d'exploitation associé à l'APMD [83, 77] ou être même incorporée au niveau matériel afin d'accroître l'efficacité comme sur la KSR par exemple [72].

Lorsqu'il dispose d'une MVP, le programmeur peut se contenter de gérer la coopération des processus par variables partagées ainsi que leur synchronisation, les accès aux données étant entièrement gérés par la MVP. Néanmoins, plusieurs recherches ont montré que l'obtention de performances est liée à une utilisation judicieuse de la MVP [76], qui tient compte notamment de la localité des accès. En effet, une mauvaise répartition des pages sur les processeurs est susceptible d'engendrer des

va-et-vient de pages ou des goulots d'étranglement qui peuvent entraîner un écroulement des performances globales du programme.

1.1.3 Distribution de programmes séquentiels

Étant donnée la difficulté de programmation parallèle explicite des APMD qui consiste en la description de processus coopérant par passage de messages ou par variables partagées via une MVP, l'idée selon laquelle un style de programmation séquentielle doit être conservé paraît séduisante. Ce type de programmation est en effet bien maîtrisé et un grand nombre d'applications ont déjà été développées.

Dans cet esprit, certaines recherches visent à fournir à l'utilisateur des bibliothèques de routines parallèles conçues manuellement pour s'exécuter efficacement sur une APMD [38, 53]. Ces routines peuvent être appelées depuis un programme séquentiel, l'obtention d'un programme parallèle se faisant ainsi de façon transparente. C'est le cas par exemple des bibliothèques de l'environnement EPEE [70] où l'utilisation d'un langage à objets renforce le potentiel de ré-utilisabilité et d'extensibilité des routines. Dans cette approche la performance des programmes repose sur l'efficacité de la mise en œuvre parallèle des bibliothèques. Étant donné l'effort de développement requis, ces dernières sont souvent très spécialisées, réduisant le spectre des applications traitées.

Des travaux sont également menés dans le domaine de la compilation. Il ne semble toutefois pas envisageable dans un avenir proche d'être capable de développer un compilateur produisant, à partir d'un code purement séquentiel, un code parallèle et distribué efficace, réalisant ainsi automatiquement la distribution des données et des calculs. On s'achemine actuellement vers des solutions partielles, dans lesquelles le programmeur guide le travail du compilateur.

Une première approche, mise en œuvre par exemple dans le compilateur Fortran-S [26], consiste à compiler un programme séquentiel dans lequel l'utilisateur a indiqué quelles sont les variables partagées et les boucles parallèles. Le compilateur utilise une MVP comme support d'exécution et génère un code SPMD dans lequel les itérations des boucles parallèles ont été distribuées, l'accès aux données partagées étant entièrement géré par la MVP.

La deuxième approche, qui ne requiert pas de MVP, peut être qualifiée de « compilation par distribution de données », elle a été retenue par de nombreux projets de recherche [101, 103, 64, 24, 31] et notamment par le projet PANDORE.

Ces deux approches ne sont pas forcément opposées, nous développerons dans cette thèse des techniques originales offrant la possibilité de rapprocher compilation par distribution du contrôle et compilation par distribution des données.

1.2 Approche de la compilation par distribution de données

Dans l'approche de la compilation par distribution de données, l'application est décrite dans un langage séquentiel impératif tel que Fortran ou *C* auquel on ajoute des annotations permettant de spécifier comment les données du programme sont réparties sur les processeurs. Le programme est soumis a un compilateur qui, guidé par la spécification de la distribution des données, produit un code distribué décrivant un ensemble de processus séquentiels communiquant par messages.

La génération de code distribué s'appuie sur le modèle SPMD et la règle des écritures locales :

- Le code généré par le compilateur est unique mais agit sur des données différentes, selon l'identité du processeur qui l'exécute.
- La règle des écritures locales permet de définir la répartition des calculs en fonction de la distribution des données : une instruction du programme source qui modifie une variable est exécutée par le processeur auquel cette variable a été attribuée par la distribution des données.

Considérons le programme suivant :

entier a, b, c, d
distribuer a et b sur p_1,
c sur p_2,
d sur p_3

$(I1)$ $a = a + b$
$(I2)$ $c = b + d$

L'application de la règle des écritures locales conduira à l'exécution des codes suivants sur les processeurs p_1, p_2 et p_3 :

Source	p_1	p_2	p_3
$(I1)$ $a = a + b$	$a = a + b$		
$(I2)$ $c = b + d$	envoyer b à p_2	recevoir d de p_3 recevoir b de p_1 $c = b + d$	envoyer d à p_2

L'instruction $(I1)$ est exécutée par le processeur p_1 ; puisque toutes les variables lues sont possédées par p_1, celui-ci est le seul à participer à l'exécution de $(I1)$, les autres processeurs passant directement à l'instruction suivante.

C'est le processeur p_2 qui effectue l'affectation ($I2$) car celle-ci modifie la variable c attribuée à p_2 par la distribution des données. Comme p_2 ne possède pas les deux variables lues, une coopération préalable à l'affectation proprement dite est nécessaire entre p_2 et les propriétaires respectifs de b et d.

Comme le possesseur d'une variable n'est pas toujours connu à la compilation (c'est par exemple le cas pour les éléments de tableau du type $A[i]$ où la valeur de i n'est connue qu'à l'exécution), le code généré par le compilateur comporte des tests sur la possession des données, qui seront appelés *gardes* dans la suite du document, permettant à chaque processeur de déterminer sa contribution à chaque instruction. On pourra générer par exemple le code SPMD suivant pour l'affectation $A[i] = B[i]$:

début cas
cas je possède $A[i]$ et $B[i]$ → **exécuter** $A[i] = B[i]$
cas je possède $A[i]$ mais pas $B[i]$ → **recevoir** $B[i]$ du possesseur de $B[i]$
exécuter $A[i] = B[i]$
cas je possède $B[i]$ mais pas $A[i]$ → **envoyer** $B[i]$ au possesseur de $A[i]$
fin cas

1.3 Environnements de compilation par distribution de données

La mise en œuvre de l'approche de la compilation par distribution de données fait intervenir de nombreuses techniques allant des aspects langages aux aspects systèmes. Elle nécessite le développement d'environnements complets (l'environnement PANDORE en est un exemple) comportant des compilateurs, des exécutifs et des outils d'évaluation de performances.

1.3.1 Langage

L'objectif de l'approche de la compilation par distribution de données est de conserver un style de programmation séquentiel classique. Toutefois, la nécessité actuelle de guider le compilateur par la spécification de la distribution des données a entraîné la définition d'extensions de langages impératifs. Plusieurs types d'extensions, essentiellement de Fortran et C ont été proposés afin d'une part de permettre la distribution des données et d'autre part d'aider le compilateur dans la détection du parallélisme potentiel (spécification de boucles parallèles). Ces extensions prennent la forme de directives ou de constructeurs ajoutés au langage. Le langage High Performance Fortran [47], dont la première version a été définie en 1993, constitue une tentative de normalisation en la matière.

La distribution des données concerne exclusivement les tableaux, les variables scalaires étant dupliquées sur l'ensemble des processeurs. On retrouve les notions suivantes dans la plupart des langages, nous prenons comme référence le langage HPF :

- *La distribution*
  Elle définit un découpage des tableaux en blocs rectangulaires de tailles égales qui sont attribués aux processeurs. On distingue les distributions régulières (BLOCK) qui associent un seul bloc à chaque processeur et les distributions cycliques (CYCLIC ou CYCLIC(k)) qui attribuent des blocs de taille 1 ou k circulairement aux processeurs.

- *L'alignement*
  Il permet de décrire la distribution d'un tableau relativement à un autre tableau ou relativement à un espace virtuel d'indices appelé TEMPLATE (Le template est alors distribué). L'alignement offre un moyen de s'affranchir du découpage rectangulaire imposé par les directives de distribution. Par exemple on pourra avoir une spécification de la forme ALIGN $A(i)$ WITH $B(2 * i + 1)$, indiquant que, quelque soit la distribution du tableau B, l'élément $A(i)$ se trouve sur le même processeur que l'élément $B(2 * i + 1)$.

Les distributions établissent une relation de possession entre les éléments de tableaux et les processeurs, qu'il s'agisse de processeurs physiques ou de processeurs virtuels organisés en tableaux comme en HPF.

D'autres recherches se situent en amont des langages de type HPF. Des outils mettant en œuvre des modèles de programmation de haut niveau ont été proposés. Avec HelpDraw [22, 21] par exemple, l'utilisateur décrit son algorithme sous la forme de manipulations géométriques à partir desquelles un programme HPF est automatiquement dérivé. Dans ObjectMath [49], le modèle de programmation est fondé sur la définition d'équations.

1.3.2 Compilateur

Le compilateur a en charge la production de code SPMD faisant coopérer des processus pour appliquer la règle des écritures locales. Pour des raisons de simplicité et de portabilité, les codes produits par les prototypes actuels sont généralement des codes source C ou Fortran qu'il faut soumettre au compilateur de l'APMD cible afin d'obtenir un programme parallèle exécutable.

L'application du schéma de compilation tel qu'il a été décrit plus haut ne permet pas d'obtenir des bonnes performances s'il est appliqué directement à chaque instruction du programme source, le nombre de gardes à évaluer et la granularité des échanges de données étant trop pénalisants. Le compilateur doit appliquer des optimisations afin de réduire le nombre de gardes et de minimiser le coût des com-

munications, en regroupant par exemple les messages ou en évitant de transférer plusieurs fois la même variable si elle n'a pas été modifiée après le premier transfert.

De telles optimisations font appel à des techniques diverses : techniques de détection du parallélisme potentiel du programme (notamment des boucles), de transformations de programme en vue de rendre exploitable ce parallélisme, d'analyse des domaines de données accédées [79] et de représentation de données distribuées.

1.3.3 Exécutif

Le code généré par le compilateur nécessite un support d'exécution effectuant un certain nombre de tâches :

- la gestion de la possession des données induite par la distribution,
- le transfert de données entre processus,
- l'accès aux données locales et reçues,
- l'allocation de la mémoire nécessaire notamment pour le stockage temporaire local des données distantes,
- la gestion des processus.

Le code réalisant ces tâches est contenu dans un composant logiciel du système de compilation : l'exécutif. Il peut être vu comme une bibliothèque de routines qu'il faut lier au code généré par le compilateur. L'efficacité de la mise en œuvre de cet exécutif est crucial pour l'obtention de performances.

Le niveau des opérations effectuées par l'exécutif peut être variable. Un exécutif de bas niveau, correspondant par exemple à la bibliothèque offerte par le système cible, imposera au compilateur de générer un code dépendant d'une plate-forme donnée, ce qui posera des problèmes de portabilité. En revanche, un exécutif de plus haut niveau renforcera la souplesse d'évolution de l'environnement de compilation tout en facilitant les optimisations à différents niveaux : dans le compilateur, dont la production de code sera simplifiée, et au sein même de l'exécutif qui pourra être plus facilement structuré. Un compromis doit cependant être réalisé dans la répartition des opérations entre le compilateur et l'exécutif afin de ne pas engendrer des calculs coûteux à l'exécution alors qu'ils pourraient être effectués par le compilateur.

La définition d'un exécutif a été souvent négligée dans les travaux sur les environnements de compilation par distribution de données. Un aspect original de notre travail dans l'environnement PANDORE a été d'isoler dans un exécutif structuré en plusieurs couches des opérations de haut niveau. Dans cet esprit, on peut citer l'environnement Adaptor [27] qui a pour cœur une bibliothèque pour la gestion des tableaux distribués et des communications [28].

1.3.4 Machines d'exécution cibles

La compilation par distribution de données visant les APMD, le modèle de programmation employé pour le code cible est tout naturellement celui des processus séquentiels coopérant par passage de messages. Nous nommerons dans la suite de ce document *machines à messages* les APMD programmées suivant ce modèle.

Dans le cadre de cette thèse, nous avons également considéré la production de code — par un compilateur dirigé par la distribution des données — pour les APMD disposant d'une mémoire virtuelle partagée, (nous les nommerons *machines à* MVP), considérant le fait que l'accès aux données distribuées constitue un des points cruciaux à résoudre efficacement et que les mécanismes de la MVP, qui s'appuient sur des composants matériels, peuvent être exploités à cet effet.

1.3.5 Performances

Si l'approche de la compilation par distribution de données facilite effectivement la programmation des APMD, les prototypes de compilateurs actuels ne produisent des codes performants que pour une classe réduite de programmes présentant des accès réguliers et une forte localité. Des expérimentations sont nécessaires pour démontrer la portée et l'efficacité des mises en œuvre et optimisations proposées. Certains projets [97, 27] fournissent des résultats obtenus sur des noyaux d'applications de calcul scientifique, souvent issus de l'algèbre linéaire. On doit considérer cette étape d'expérimentation comme une étape préalable à des expériences en vraie grandeur, c'est-à-dire sur des applications réelles.

L'analyse des performances des codes produits est une tâche difficile pour le programmeur mais aussi pour les développeurs des environnements de compilation, l'interprétation des seuls temps d'exécution ne suffisant pas toujours. Les environnements de compilation par distribution de données tentent donc d'intégrer des outils d'aide à l'évaluation de performances. Il s'agit d'outils de prédiction ou de mesure. Les techniques employées par ces outils — plus particulièrement par les outils de mesure — dépassent le cadre de la compilation par distribution de données (de nombreux aspects se retrouvent également dans les travaux sur la distribution du contrôle, comme par exemple dans l'environnement Alpes [75]), elles concernent notamment l'instrumentation de code, la collecte et l'analyse de données de performances.

Chapitre 2

Machine abstraite pour la compilation dirigée par les données

Le principe de la compilation dirigée par la distribution de données n'est pas étudié en fonction d'une machine spécifique, le schéma de compilation doit donc être conçu hors des références à une architecture cible donnée. La mise en œuvre doit également être portable. Il est donc nécessaire d'introduire un niveau d'abstraction adapté, au-dessus des machines d'exécution, afin de clarifier la description de l'approche et la structuration de sa mise en œuvre.

Dans ce chapitre, nous proposons de définir des opérations adaptées à la compilation dirigée par les données. Nous regroupons ces opérations sous le nom de « machine abstraite ». Ces opérations sont définies en fonction du schéma de compilation qui suit le modèle SPMD et applique la règle des écritures locales. Le but n'est pas de définir formellement une machine complète mais de se doter d'un cadre suffisamment général d'expression du schéma de compilation pour machine à messages ainsi que pour machine à MVP et de prendre en compte les optimisations des schémas. À notre connaissance, aucun travail de ce type n'a été effectué dans les autres projets portant sur les environnements de compilation par distribution données. On peut toutefois trouver une présentation d'opérations ayant des objectifs similaires dans [96].

Les éléments essentiels de la mise en œuvre des opérations sont également abordés, l'accent est mis sur la représentation des données distribuées et les optimisations liées aux mouvements de données.

La définition des opérations des machines abstraites et les points clés de mise en œuvre donnent d'une part un cadre général pour exprimer et comparer différentes approches suivies par plusieurs projets de recherche, ce que nous faisons au paragraphe 2.6 ; d'autre part, ces opérations nous serviront, dans le chapitre suivant, de

support pour la description du travail effectué dans le compilateur PANDORE.

2.1 Schéma de compilation de base

Le schéma de compilation définit la transformation du programme séquentiel source en un programme parallèle SPMD. Cette transformation est dirigée par la distribution des données et est effectuée instruction par instruction. La distribution des données, spécifiée par le programmeur, établit une relation de possession entre les données du programme et les processus[1] de la machine d'exécution. Elle se traduit par l'utilisation d'une fonction *owner*($\mathcal{R}$) qui indique l'ensemble des processus auxquels ont été attribuées les variables (scalaires ou éléments de tableaux) référencées dans l'ensemble $\mathcal{R}$. La transformation suit la règle des écritures locales : l'écriture d'une variable est effectuée uniquement par le processus qui la possède.

Si l'on fait abstraction de la mémoire, une instruction $\mathcal{S}$ qui écrit un ensemble de variables DEF($\mathcal{S}$) et qui lit un ensemble de variables USE($\mathcal{S}$) est traduite en la suite d'opérations :

1) Synchronisation de *owner*(DEF($\mathcal{S}$)) avec *owner*(USE($\mathcal{S}$)) :
les processus possédant les variables de DEF($\mathcal{S}$) attendent que les processus possédant les variables de USE($\mathcal{S}$) aient atteint ce point de synchronisation. Ceci garantit que les valeurs des variables de USE($\mathcal{S}$) qui seront lues ont une valeur « à jour » c'est-à-dire conforme à celle prise avant $\mathcal{S}$ lors de l'exécution séquentielle.

2) Masquage de $\mathcal{S}$ par *owner*(DEF($\mathcal{S}$)) :
conditionne l'exécution de $\mathcal{S}$ de sorte que seuls les possesseurs des variables de DEF($\mathcal{S}$) exécutent l'instruction $\mathcal{S}$. On dira que l'instruction $\mathcal{S}$ est alors « gardée ».

3) Synchronisation de *owner*(USE($\mathcal{S}$)) avec *owner*(DEF($\mathcal{S}$)) :
les processus possédant les variables de USE($\mathcal{S}$) attendent que les processus possédant les variables de DEF($\mathcal{S}$) aient atteint ce point. Ceci garantit que les variables de USE($\mathcal{S}$) ne sont par re-écrites avant qu'elles ne soient lues.

Considérons par exemple la suite de deux instructions S_1 et S_2 suivante :

$$\begin{array}{lcl} S_1 & : & a = b + c \\ S_2 & : & d = e \end{array}$$

1. Nous considérerons dans la suite du document qu'il n'y a qu'un seul processus par processeur et emploierons indifféremment les deux termes.

où chaque variable est possédée par un processeur différent. Ceux-ci sont notés P_a, P_b,... P_e. Le schéma de compilation produit le code :

S_1 : synchronisation de P_a avec P_b et P_c
si je suis P_a **alors** exécuter $a = b + c$
synchronisation de P_b et P_c avec P_a

S_2 : synchronisation de P_d avec P_e
si je suis P_d **alors** exécuter $d = e$
synchronisation de P_e avec P_d

Cette étape définit une parallélisation du programme séquentiel. Seuls sont effectivement synchronisés les processus qui participent à l'instruction (possesseurs des variables de DEF($\mathcal{S}$) et USE($\mathcal{S}$)). Les autres peuvent s'exécuter indépendamment. Deux instructions qui se suivent dans le programme séquentiel, si elles opèrent sur des ensembles de variables possédés par des processus différents, sont exécutées en parallèle. La figure 2.1 illustre une exécution du programme généré : S_1 et S_2 font intervenir des ensembles de variables disjoints, elles sont effectivement exécutées en parallèle car aucune synchronisation n'intervient entre les processeurs participant à S_1 et ceux participant à S_2.

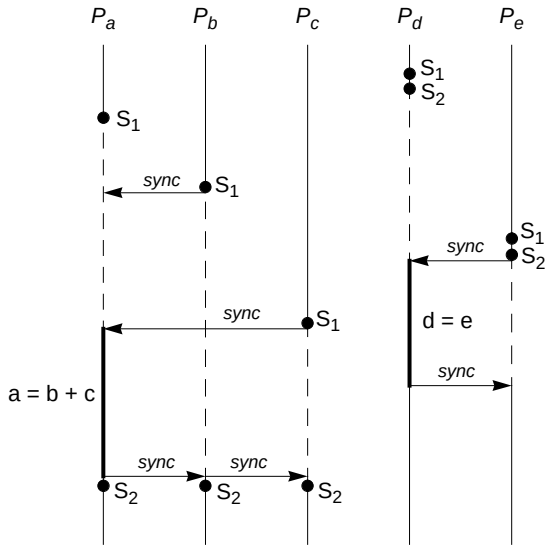


FIG. 2.1. – *Exécution parallèle de deux instructions*

Ce schéma de compilation peut être utilisé tel quel pour une machine à mémoire partagée où toutes les variables sont directement accessibles par tous les processus.

Sur les machines à mémoire distribuée que nous considérons ici, les mémoires sont privées, c'est-à-dire qu'un processeur ne peut accéder directement qu'à la mémoire qui lui est attachée. En plus de la relation de possession, la distribution des

données induit leur localisation physique dans les mémoires privées. On a alors, pour un processeur donné, la notion de variable *locale* c'est à dire présente dans sa mémoire privée et de variable *distante* c'est-à-dire stockée dans la mémoire d'un autre processeur. L'accès à une variable distante ne peut se faire que via une copie dans la mémoire locale. Une copie explicite est donc nécessaire avant l'opération de masquage de l'instruction. $\mathcal{S}$ est alors traduite comme suit :

(A)
1) Synchronisation de *owner*(DEF($\mathcal{S}$)) avec *owner*(USE($\mathcal{S}$))
2) Copie de USE($\mathcal{S}$) chez *owner*(DEF($\mathcal{S}$))
3) Masquage de $\mathcal{S}$ par *owner*(DEF($\mathcal{S}$))
4) Synchronisation de *owner*(USE($\mathcal{S}$)) avec *owner*(DEF($\mathcal{S}$))

L'utilisation d'une copie locale de USE($\mathcal{S}$) durant l'exécution de $\mathcal{S}$ permet de remonter la deuxième synchronisation avant le masquage :

(B)
1) Synchronisation de *owner*(DEF($\mathcal{S}$)) avec *owner*(USE($\mathcal{S}$))
2) Copie de USE($\mathcal{S}$) chez *owner*(DEF($\mathcal{S}$))
3) Synchronisation de *owner*(USE($\mathcal{S}$)) avec *owner*(DEF($\mathcal{S}$))
4) Masquage de $\mathcal{S}$ par *owner*(DEF($\mathcal{S}$))

2.2 Introduction des machines abstraites

2.2.1 Application du schéma sur machine à messages

Le schéma (B) est traditionnellement appliqué lorsque l'on vise une machine distribuée à messages. Les trois premiers points sont regroupés dans une opération que nous baptisons *Refresh*, le dernier point étant traité dans l'opération *Exec* [5]. Le schéma de compilation s'exprime alors par la production d'un code SPMD *parallèle* et *distribué* pour une machine abstraite comportant deux opérations (*Refresh* et *Exec*) :

$\mathcal{S} \Rightarrow$
- *Refresh*(USE($\mathcal{S}$), DEF($\mathcal{S}$))
- *Exec*(DEF($\mathcal{S}$), $\mathcal{S}$)

L'opération *Refresh* permet d'obtenir des copies à jour (rafraîchit) des variables distantes, l'*Exec* appliquant la règle des écritures locales proprement dite.

Dans cette traduction, les communications se font par files FIFO. Nous utilisons la réception bloquante (le récepteur est bloqué tant que le message n'est pas effec-

tivement reçu et disponible dans le tampon de réception) afin de mettre en œuvre la première synchronisation. Par contre, l'émission est non bloquante et tamponnée (l'émetteur ne continue en séquence que lorsque le tampon d'émission a été pris en compte par le système, le processus émetteur pouvant alors récrire dans celui-ci). Ceci met en œuvre la deuxième synchronisation.

Une mise en œuvre possible de l'opération *Refresh* est la suivante :

$$\textit{Refresh}(\text{USE}(\mathcal{S}),\ \text{DEF}(\mathcal{S})) \equiv \begin{array}{l}\textbf{si}\ \text{je possède DEF}(\mathcal{S})\ \text{et je ne possède pas USE}(\mathcal{S}) \\ \quad \textbf{alors}\ \text{je reçois USE}(\mathcal{S})\ \text{depuis}\ \textit{owner}(\text{USE}(\mathcal{S})) \\ \textbf{si}\ \text{je possède USE}(\mathcal{S})\ \text{et je ne possède pas DEF}(\mathcal{S}) \\ \quad \textbf{alors}\ \text{j'envoie USE}(\mathcal{S})\ \text{à}\ \textit{owner}(\text{DEF}(\mathcal{S}))\end{array}$$

Cette mise en œuvre est libre d'interblocage avec l'hypothèse de files FIFO infinies. En effet, durant l'exécution du *Refresh*, les ensembles de processus émetteurs, récepteurs et ne communiquant pas sont disjoints et à chaque réception correspond une émission. On ne peut donc pas avoir, par exemple, deux processus effectuant chacun, dans le même ordre, une réception depuis l'autre processus et une émission vers l'autre processus. Une preuve formelle de l'absence d'interblocage est donnée dans [16]. Le caractère infini des files est toutefois nécessaire à la correction, la détermination de la taille des files nécessaire à une exécution étant indécidable dans le cas général [32].

L'opération *Exec* est simplement mise en œuvre par un test sur la possession de la donnée :

$$\textit{Exec}(\text{DEF}(\mathcal{S}),\ \mathcal{S}) \equiv \quad \textbf{si}\ \text{je possède DEF}(\mathcal{S})\ \textbf{alors}\ \mathcal{S}$$

2.2.2 Application du schéma sur machine à MVP

Sur une machine à mémoire virtuellement partagée, la mémoire est également distribuée, il y a donc copie pour la lecture d'une donnée distante. Mais à la différence de la machine à messages, ces copies peuvent être maintenues cohérentes par le système. Deux utilisations de la mémoire virtuelle partagée sont possibles suivant que l'on considère les copies implicites ou explicites.

Copies implicites

On peut considérer que ces copies se font de manière totalement transparente et donc utiliser la machine à mémoire virtuelle partagée comme une machine à mémoire partagée. Le schéma de compilation (A) s'applique, à la seule différence que la copie est implicitement réalisée lors de l'*Exec*. Le schéma permet de produire du

code pour une machine abstraite comportant deux opérations (*Sync* et *Exec*) :

$$\mathcal{S} \Rightarrow \begin{array}{l} \bullet\ \mathit{Sync}(\mathrm{DEF}(\mathcal{S}) \cup \mathrm{USE}(\mathcal{S})) \\ \bullet\ \mathit{Exec}(\mathrm{DEF}(\mathcal{S}),\ \mathcal{S}) \\ \bullet\ \mathit{Sync}(\mathrm{DEF}(\mathcal{S}) \cup \mathrm{USE}(\mathcal{S})) \end{array}$$

L'opération *Exec* permettant de masquer l'exécution de $\mathcal{S}$ est identique à celle utilisée pour la machine à messages. L'opération *Sync* réalise la synchronisation des processus possédant les variables référencées en paramètre : un processus est bloqué sur cette opération tant que tous les processus possédant des variables passées en paramètre ne l'ont pas invoquée. On utilise ici une opération de synchronisation symétrique que l'on trouve de manière courante sur les MVP. Cette synchronisation est plus forte que les synchronisations asymétriques nécessaires mais en pratique, son emploi n'entraîne pas de surcoût notable.

L'absence d'interblocage est ici assurée par la symétrie de l'opération *Sync* : aucun cycle d'attente (p_0 se synchronise avec p_1, p_1 avec p_2 et p_1 avec p_0 par exemple) ne peut intervenir car tous les processus impliqués dans un *Sync* participent à la synchronisation avant de continuer en séquence. Les ensembles de processus participant au même moment à un *Sync* sont soit disjoints, soit égaux.

Copies explicites

Une alternative est de considérer que les copies peuvent se faire explicitement. On considère toutefois que la cohérence des copies est quand même maintenue automatiquement, les copies explicites se rajoutant aux copies implicites. Le schéma de compilation s'exprime alors exactement comme décrit en (A) :

$$\mathcal{S} \Rightarrow \begin{array}{l} \bullet\ \mathit{Sync}(\mathrm{DEF}(\mathcal{S}) \cup \mathrm{USE}(\mathcal{S})) \\ \bullet\ \mathit{Get}(\mathrm{USE}(\mathcal{S}),\ \mathrm{DEF}(\mathcal{S})) \\ \bullet\ \mathit{Exec}(\mathrm{DEF}(\mathcal{S}),\ \mathcal{S}) \\ \bullet\ \mathit{Sync}(\mathrm{DEF}(\mathcal{S}) \cup \mathrm{USE}(\mathcal{S})) \end{array}$$

L'opération $\mathit{Get}(\mathrm{USE}(\mathcal{S}),\ \mathrm{DEF}(\mathcal{S}))$ indique que les processus possédant des variables référencées dans $\mathrm{DEF}(\mathcal{S})$ doivent acquérir des copies des variables référencées dans $\mathrm{USE}(\mathcal{S})$. Cette opération peut être mise en œuvre par une lecture anticipée chez les processus voulus :

$\mathit{Get}(\mathrm{USE}(\mathcal{S}),\ \mathrm{DEF}(\mathcal{S})) \equiv$ **si** je possède $\mathrm{DEF}(\mathcal{S})$ **alors** lire $\mathrm{USE}(\mathcal{S})$

Remarque

On doit garder la deuxième synchronisation empêchant le possesseur de l'original de continuer tant que la copie n'est pas lue. En effet, si le possesseur de USE($\mathcal{S}$) n'effectue pas d'attente, il est susceptible de modifier les variables de USE($\mathcal{S}$) lors d'un *Exec* ultérieur. Considérons par exemple la compilation des deux instructions

$$\begin{array}{lcl} S_1 & : & a = b \\ S_2 & : & b = c \end{array}$$

avec a possédée par P_a, b par P_b et c par P_c. On suppose qu'avant ces instructions b vaut 2 et que c vaut 3. L'application du schéma de compilation donnera :

Sync($\{a, b\}$)
Get($\{b\}$, $\{a\}$)
Exec($\{a\}$, $a = b$)
Sync($\{a, b\}$)

Sync($\{b, c\}$)
Get($\{c\}$, $\{b\}$)
Exec($\{b\}$, $b = c$)
Sync($\{b, c\}$)

Si l'on omet la deuxième synchronisation, P_b n'attend pas que la valeur de b soit lue par P_a pour continuer en séquence. De ce fait, il modifie b trop tôt. La figure 2.2 montre deux possibilités d'exécution erronée (a vaut 3 et non pas 2 après exécution des deux instructions) :

i) P_b modifie b avant que P_a ait exécuté le *Get*, auquel cas P_b fournit la nouvelle valeur de b (*i.e.* 3) à P_a lors du *Get*.

ii) P_b modifie b après que P_a ait exécuté le *Get* mais avant qu'il n'ait terminé l'*Exec*, auquel cas P_b fournit la bonne valeur de b à P_a lors du *Get* mais cette copie est invalidée pendant l'*Exec* sur P_a et mise à jour automatiquement par le système lors de sa lecture par P_a.

2.3 Optimisation du schéma de compilation

Le schéma de compilation présenté en 2.1 peut être appliqué successivement à toutes les instructions élémentaires du programme séquentiel pour obtenir un programme SPMD parallèle distribué. Le cas de son application sur machine à messages a été l'objet d'une description détaillée [4] et d'une preuve [16].

Cependant, le nombre de synchronisations et d'évaluations de gardes ainsi que la granularité des communications dégradent fortement l'efficacité du code produit.

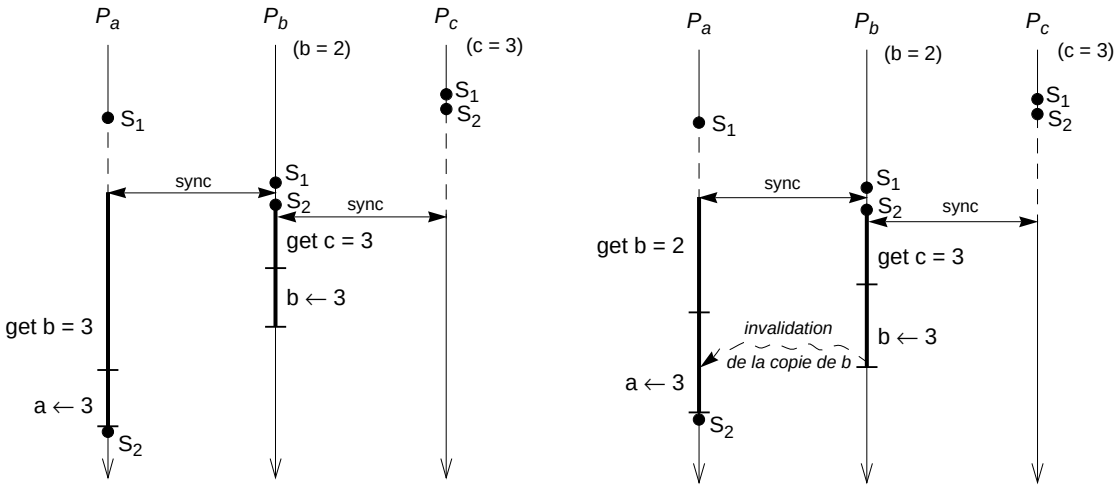


FIG. 2.2. – *Exécutions erronées en l'absence de la* $2^{ème}$ *synchronisation*

L'optimisation du schéma passe tout d'abord par l'augmentation du grain d'application du schéma, le code produit pouvant également être ultérieurement optimisé.

Il est souhaitable que le schéma de compilation puisse être appliqué non pas seulement aux instructions élémentaires mais à des groupes d'instructions. Ceci permet en effet de réduire le nombre de synchronisations produites, de factoriser les gardes et de regrouper les mouvements de données.

Les regroupements d'instructions qu'il est possible de construire techniquement et qui présentent un intérêt pour factoriser les tests sont représentés par les boucles et plus particulièrement les boucles parallèles. Le compilateur peut effectuer une analyse de dépendances sur le code source afin de les détecter. Il existe également des méthodes de transformation de programmes permettant de faire apparaître ce genre de boucles à partir de boucles comportant des dépendances. L'application de ces techniques pour application d'un schéma de compilation optimisé est étudiée dans [79].

La définition des opérations de la machine abstraite est élargie de sorte que le compilateur puisse traduire les instructions du programme séquentiel groupe par groupe. Pour passer des instructions élémentaires aux groupes d'instructions (*i.e.* les boucles parallèles), nous paramétrons les références aux variables par des variables libres (*i.e.* les variables d'itération). Un paramètre est également ajouté aux opérations : l'ensemble des valeurs prises par ces variables libres (*i.e.* ensemble des valeurs du vecteur d'itération). Dans la suite, on désignera cet ensemble sous le nom de *domaine*.

Par exemple, le schéma optimisé appliqué à la boucle parallèle :

pour i **de** 5 **à** 10
    $A[i] = B[i]$
fpour

produit le code suivant (pour une machine à messages) :

Refresh($\{B[i]\}$, $\{A[i]\}$, $\{i = 5..10\}$)
Exec($\{A[i]\}$, $A[i] = B[i]$, $\{i = 5..10\}$)

Ce code peut être naïvement mis en œuvre de la façon suivante :

— *Refresh* —
pour i **de** 5 **à** 10
    si je possède $A[i]$ et je ne possède pas $B[i]$
        alors je reçois $B[i]$ depuis *owner*($B[i]$)
    si je possède et $B[i]$ je ne possède pas $A[i]$
        alors j'envoie $B[i]$ à *owner*($A[i]$)
fpour

— *Exec* —
pour i **de** 5 **à** 10
    si je possède $A[i]$ **alors** $A[i] = B[i]$
fpour

Des améliorations peuvent également être mises en œuvre après application du schéma (optimisé ou non). Elles consistent essentiellement à déplacer et regrouper les opérations de la machine abstraite apparaissant dans le code intermédiaire, après une analyse de flots de données et d'éventuelles transformations de boucles. Ces dernières sont des manipulations de restructuration de codes « classiques » issues des vectoriseurs et paralléliseurs pour machine à mémoire partagée telles que la distribution, la fusion ou le partitionnement de boucles [2, 99, 102].

2.4 Définition des opérations des machines abstraites

Les définitions données ici permettent de prendre en compte un gros grain d'application du schéma de compilation (elles manipulent des domaines) et d'éventuels regroupements (elles possèdent des ensembles de paramètres) : par exemple regrouper des *Refresh* consistera à produire un seul *Refresh* auquel on passe en paramètre l'union des paramètres des *Refresh* considérés.

Machine à messages

L'opération Refresh

L'opération *Refresh* prend en paramètre un ensemble (non ordonné) de triplets :

$Refresh(T_1,\ T_2, \dots))$

Chaque T_k est de la forme $(\mathcal{R}(\mathcal{I}),\ \mathcal{R}_P(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$ où

- $\mathcal{I}$ est un ensemble de variables libres ;
- $\mathcal{R}(\mathcal{I})$ et $\mathcal{R}_P(\mathcal{I})$ sont des ensembles de références à des variables distribuées paramétrés par $\mathcal{I}$;
- $\mathcal{D}(\mathcal{I})$ est un ensemble de valeurs prises par $\mathcal{I}$.

Pour chaque triplet T_k, l'opération *Refresh* rend disponible sur les processus possédant une variable référencée dans $\mathcal{R}_P(\mathcal{I})$, une copie des variables référencées dans $\mathcal{R}(\mathcal{I})$ pour toutes les valeurs de $\mathcal{I}$ décrites dans $\mathcal{D}(\mathcal{I})$. Les processus fournissant les copies attendent que les copies soit constituées et les processus bénéficiant des copies attendent que les copies soient effectuées. Par exemple,

$Refresh((\{A[i], B[i]\},\ \{C[i], D[i]\},\ \{i = 0, 2, 4\}),\ (\{A[j]\},\ \{C[j]\},\ \{j = 1, 3, 5\}))$

rafraîchit
$A[0]$ et $B[0]$ sur les processus qui possèdent $C[0]$ ou $D[0]$;
$A[2]$ et $B[2]$ sur les processus qui possèdent $C[2]$ ou $D[2]$;
$A[4]$ et $B[4]$ sur les processus qui possèdent $C[4]$ ou $D[4]$;
$A[1]$ sur les processus qui possèdent $C[1]$;
$A[3]$ sur les processus qui possèdent $C[3]$;
$A[5]$ sur les processus qui possèdent $C[5]$.

L'opération Exec

L'opération *Exec* prend également en paramètre un ensemble de triplets où figurent un ensemble de références paramétré $\mathcal{R}_P(\mathcal{I})$ et un domaine $\mathcal{D}(\mathcal{I})$:

$Exec((\mathcal{R}_P(\mathcal{I}),\ \mathcal{S}(\mathcal{I}),\ \mathcal{D}(\mathcal{I})), \dots)$

Pour chaque triplet, on précise en plus un paramètre $\mathcal{S}(\mathcal{I})$: un ensemble d'instructions élémentaires paramétrées par $\mathcal{I}$. L'opération *Exec* fait en sorte que seuls les processus possédant une variable référencée dans $\mathcal{R}_P(\mathcal{I})$ exécutent les instructions de $\mathcal{S}(\mathcal{I})$, ceci pour toutes les valeurs de $\mathcal{I}$ décrites dans $\mathcal{D}(\mathcal{I})$.

Par exemple,

$Exec((\{A[i]\},\ \{A[i] = B[i],\ C[i] = D[i]\},\ \{i = 0, 2, 4\}),$
$\quad(\{B[i]\},\ \{B[i] = B[i] + 1\}; \{i = 1, 3, 5\}))$

effectuera les affectations suivantes[2] :
$A[0] = B[0]$ et $C[0] = D[0]$ sur les processus qui possèdent $A[0]$;
$A[2] = B[2]$ et $C[2] = D[2]$ sur les processus qui possèdent $A[2]$;
$A[4] = B[4]$ et $C[4] = D[4]$ sur les processus qui possèdent $A[4]$;
$B[1] = B[1] + 1$ sur les processus qui possèdent $B[1]$;
$B[3] = B[3] + 1$ sur les processus qui possèdent $B[3]$;
$B[5] = B[5] + 1$ sur les processus qui possèdent $B[5]$;

Aucun ordre n'est imposé dans l'exécution des différentes instructions de $\mathcal{S}(\mathcal{I})$ sur un processus donné. Ceci est justifié par le fait que l'opération *Exec* est utilisée dans un contexte particulier où les instructions de $\mathcal{S}(\mathcal{I})$ sur un processus donné peuvent commuter. La définition d'une opération *Exec* plus générale où il pourrait exister des dépendances entre les instructions de $\mathcal{S}(\mathcal{I})$ pour un processus donné nécessiterait l'ajout d'un paramètre supplémentaire à l'opération. Ce paramètre décrirait les dépendances entre les instructions des ensembles $\mathcal{S}(\mathcal{I})$ sous la forme d'un graphe par exemple.

Machine à MVP

L'opération Sync

L'opération *Sync* prend en paramètre un ensemble de couples :

$$Sync((\mathcal{R}_P(\mathcal{I}),\ \mathcal{D}(\mathcal{I})), \dots)$$

La définition des composantes $\mathcal{R}_P(\mathcal{I})$ et $\mathcal{D}(\mathcal{I})$ est la même que pour les opérations *Refresh* ou *Exec*. L'opération *Sync* effectue une seule synchronisation d'un ensemble de processus. Un processus participe à la synchronisation si, pour un ou plusieurs des couples passés en paramètre, il possède une des variables référencées dans $\mathcal{R}_P(\mathcal{I})$ pour une des valeurs appartenant à $\mathcal{D}(\mathcal{I})$.

L'opération Get

L'opération *Get* prend les mêmes paramètres que l'opération *Refresh*. Sa fonction est similaire mis à part les synchronisations : aucune attente n'est effectuée dans le *Get*.

L'opération Exec

Il s'agit de la même opération que l'opération *Exec* pour la machine à messages.

2. On peut avoir une affectation de $C[i]$ dans le paramètre $\mathcal{S}(\mathcal{I})$ sans que $C[i]$ n'apparaisse dans l'ensemble $\mathcal{R}_P(\mathcal{I})$ si le compilateur peut déterminer que $C[i]$ et $A[i]$ se trouvent sur le même processeur pour les valeurs de i considérées.

2.5 Points clés de mise en œuvre

On pourrait envisager de considérer les opérations de la machine abstraite comme des primitives de l'exécutif mais cela obligerait à effectuer la totalité du travail supporté par ces opérations à l'exécution et donc empêcherait bon nombre d'optimisations, allant de ce fait à l'encontre des objectifs visé lors de l'élargissement du grain d'application du schéma de compilation.

C'est pourquoi on considère que les opérations de la machine abstraite constituent un langage intermédiaire dans le compilateur et que l'interprétation de la machine abstraite est faite en partie par un travail statique (*i.e.* par le compilateur) et en partie par un travail dynamique (*i.e.* par l'exécutif).

On distingue deux parties (haute et basse) dans le compilateur :

- La partie haute agit en deux temps, elle effectue :
  - l'analyse du source et l'application du schéma de compilation pour produire un code intermédiaire SPMD faisant appel aux opérations *Refresh*, *Exec*, *Sync* et *Get* ;
  - l'analyse et la transformation du code intermédiaire qui déplace et regroupe les opérations *Refresh*, *Exec*, *Sync* et *Get*.
- La partie basse traduit ces opérations en code de plus bas niveau, comprenant notamment des appels aux primitives de l'exécutif.

Nous développons dans la suite les points clés de la mise en œuvre de la machine abstraite, en précisant ce qui peut être effectué à la compilation. L'efficacité de cette mise en œuvre dépend en effet de facteurs qui n'ont pas été abordés lors de sa définition : l'accès aux données doit être déterminé, des domaines d'itération et de données sont pris en compte, et des communications doivent être mises en place pour les copies de données distantes.

2.5.1 Gestion des données distribuées

2.5.1.1 Représentation des données et accès

Une des tâches principales de la machine abstraite est la représentation et l'accès aux données distribuées. Le problème se pose principalement pour les tableaux distribués. On passe d'une entité unique à une collection d'entité : les blocs possédés par les processus.

Les tableaux manipulés dans le programme source et dont des références aux éléments apparaissent en paramètre des opérations de la machine abstraite sont de vrais tableaux multi-dimensionnels. Aucune hypothèse n'est faite à ce niveau sur leur représentation. Une référence à un élément de tableau est représentée par un

nom de variable et un vecteur d'indices. Or les machines d'exécution ne permettent en fait que la manipulation d'une mémoire linéaire.

La machine abstraite doit donc :

- stocker les éléments des tableaux distribués dans la mémoire de la machine d'exécution (mémoires privées ou mémoire virtuellement partagée).
- permettre l'accès aux données, c'est-à-dire fournir l'adresse mémoire correspondant à un élément de tableau repéré par un nom de variable et un vecteur d'indices.

Occupation mémoire et rapidité d'accès

Un compromis doit être réalisé entre le coût mémoire induit par la représentation des tableaux et la rapidité des accès aux éléments. La solution extrême consistant à allouer le tableau entier sur tous les processus n'est évidemment pas applicable en général : si aucune transformation n'est à opérer en ce qui concerne les accès, le surcoût mémoire est prohibitif. Cette méthode ne peut donc être utilisée que pour les tableaux de petite taille. Inversement, une représentation mémoire peu coûteuse (typiquement réalisée en transformant la déclaration d'un tableau $A(N)$ en une déclaration locale $A(N/P)$ où P est le nombre de processus) associée à des transformations d'indices utilisant plusieurs opérations coûteuses telles que la division entière ou le modulo est à éviter.

Uniformité

Deux types de données sont à considérer sur un processus donné : les éléments *locaux* (attribués à ce processus par la distribution) et les éléments *reçus*, copies temporaires d'éléments distants. On qualifie d'uniforme une gestion des tableaux distribués dans laquelle la représentation et l'accès sont identiques qu'il s'agisse des éléments de la partition locale ou des éléments reçus.

En dehors de la simplicité évidente, l'avantage d'une gestion de tableaux uniforme réside dans le fait qu'il n'y a pas besoin de séparer les calculs purement locaux (n'utilisant que des données locales) des calculs requérant des données distantes.

En effet, la plupart des optimisations appliquées à des boucles séparent le code produit en une phase de communication et une phase de calcul durant laquelle on accède aux éléments locaux et aux éléments précédemment reçus (voir par exemple [67, 81]). Si le compilateur n'est pas capable de découper la phase de calcul en une partie n'accédant en lecture qu'aux éléments locaux et une partie n'accédant qu'aux éléments reçus, le calcul du possesseur de chaque élément est à effectuer à l'exécution à chaque référence afin de sélectionner le mécanisme d'accès approprié.

La séparation des calculs purement locaux des calculs nécessitant des données distantes est possible dans certains cas simples mais peut s'avérer très complexe (à

la fois en ce qui concerne la compilation et le code généré) dans le cas général. Il est donc préférable de ne pas faire l'hypothèse de cette séparation lors de la définition de la gestion des tableaux distribués.

Indépendance vis à vis des instructions

Dans le programme source soumis au compilateur, on distingue les spécifications de distribution des données des instructions manipulant ces données. On dira qu'une gestion des tableaux distribués est indépendante vis à vis des instructions si elle est entièrement définie à partir de la déclaration du tableau original et de la spécification de sa distribution.

Ce n'est pas le cas lorsque la représentation distribuée d'un tableau et son mode d'accès est sujette également à l'analyse des boucles où les références à ce tableau apparaissent ou du type des accès à ce tableau dans le code séquentiel. Considérons par exemple deux nids de boucles successifs impliquant un même tableau distribué. Pour chaque nid de boucle, à partir des bornes et des références aux tableaux, on détermine une gestion de tableau optimale différente. Pour pallier cette différence, deux solutions sont envisageables :

- Remettre en cause les définitions idéales de chaque gestion et trouver une gestion commune. Ce compromis est difficile à déterminer et risque d'être fortement préjudiciable.
- Effectuer un changement de représentation et/ou d'accès entre les deux nids de boucles. Cela peut occasionner à l'exécution un réarrangement des données ou un recalcul de mécanisme d'accès.

Dans les deux cas, on risque d'aboutir à un surcoût en temps ou en mémoire, surcoût qui n'existe pas avec une gestion de tableaux indépendante.

Conservation de la contiguïté

Une propriété intéressante d'une représentation des tableaux distribués est la conservation de la contiguïté des éléments. En effet, le fait que des éléments contigus dans la représentation séquentielle du tableau soit aussi contigus dans la représentation distribuée présente plusieurs avantages :

- L'exploitation de processeurs vectoriels est possible.
- Il est plus facile d'utiliser des communications directes — c'est-à-dire des communications dans lesquelles on n'effectue pas d'empaquetage/dépaquetage entre les mémoires locales et les tampons de communication —, les ensembles d'éléments à communiquer étant généralement contigus dans la représentation séquentielle (*e.g.* lignes ou colonnes d'un tableau bi-dimensionnel).
- L'optimisation du code généré (par le compilateur de la machine cible) est facilitée.

- Les défauts de cache lors des accès successifs aux éléments sont moins probables (il est fréquent que les éléments soient accédés suivant leur représentation séquentielle).

2.5.1.2 Possession des données

Pour pouvoir appliquer la règle des écritures locales via l'utilisation de la fonction *owner*, la machine abstraite doit être capable de mettre en œuvre la relation de possession entre les données du programme et les processus de la machine d'exécution, conformément aux indications du programmeur.

La détermination de possesseurs de données à l'exécution est une fonctionnalité centrale de la machine abstraite qui apparaît dans toutes ses opérations, elle doit être mise en œuvre efficacement. Il s'agit d'associer à un élément de tableau donné un ou plusieurs identificateurs de processus de la machine d'exécution.

Par ailleurs, on peut mettre en œuvre des fonctionnalités annexes :

- La distribution des tableaux se faisant en deux temps (partitionnement en blocs du tableau puis attribution des blocs aux processus), on doit permettre de déterminer le ou les possesseurs d'un bloc de données.
- La relation de possession inverse peut aussi être nécessaire, la machine abstraite doit permettre de décrire l'ensemble des données possédées par un processus donné.

2.5.2 Restriction des domaines

Les opérations de la machine abstraite que nous avons définies prennent en paramètre des domaines. Représenter et manipuler ces domaines efficacement est indispensable à la fois dans le compilateur et dans l'exécutif. Nous illustrons ce problème pour les domaines de calcul mentionnés dans l'opération *Exec*.

Les domaines peuvent être considérés comme des ensembles de points élémentaires à énumérer. Ainsi, la mise en œuvre directe de l'opération *Exec* consiste à effectuer un test de possession pour l'ensemble des valeurs du domaine :

$Exec(\{A[i+x]\},\ \{A[i+x] = B[i]\},\ \{i = 0, 2, 4..98\}) \equiv$

pour i **de** 0 **à** 98 **pas** 2
  si je possède $A[i+x]$ **alors** $A[i+x] = B[i]$
fpour

N'importe quels domaines et n'importe quelles références peuvent être pris en compte de cette manière. Le domaine est représenté dans le code généré par le compilateur par une boucle qui est entièrement parcourue par chaque processus.

Cette méthode ne requiert aucune analyse particulière, en revanche, son efficacité est réduite.

L'optimisation principale concernant les domaines fait en sorte que chaque processus prenne en charge seulement une partie du domaine spécifié, réduisant ainsi le nombre d'opérations élémentaires. On la référence sous le nom de restriction de domaines. Les techniques de restriction de domaines constituent une part importante de la recherche consacrée aux systèmes de compilation par distribution de données. Ces techniques ne font pas l'objet de la présente étude, on en donne ici un bref aperçu.

On considère deux types d'optimisations suivant que la restriction est faite statiquement ou dynamiquement.

2.5.2.1 Restriction statique

On parle de restriction statique des domaines quand on peut déterminer dès la compilation la part du domaine que chaque processus aura à sa charge. Cette optimisation a été envisagée dès les premiers travaux sur les environnements de compilation dirigée par les données [101, 33, 8] ; nous verrons l'intégration de ce type de techniques dans les environnements actuels au paragraphe 2.6. La part de calcul attribuée à chaque processus est calculée à partir d'une analyse symbolique du domaine, des références et de la distribution des variables. Le code généré décrit un parcours restreint sous forme de boucles dont les bornes sont des expressions calculées à la compilation. Par exemple, si A est un tableau de 1000 éléments divisé en blocs de 100 et réparti sur 10 processus, on pourra avoir :

$Exec(\{A[i]\},\ \{A[i] = B[i]\},\ \{i = 0..99\}) \equiv$

pour i **de** $moi*100$ **à** $moi*100+99$
  $A[i] = B[i]$
fpour

où moi est le numéro du processus.

L'utilisation de techniques d'analyse symbolique limite le champ d'application de ce genre de restriction à des domaines et références particuliers liés aux choix de représentation des domaines. Parmi ces représentations, on peut citer :

- *les pavés* codés par un couple (borne inférieure, borne supérieure) dans chaque dimension ;
- *les sections de tableaux* où un pas est rajouté dans chaque dimension au couple (borne inférieure, borne supérieure) afin de tenir compte de trous dans le domaine ;

- *les polyèdres* codés par un ensemble de contraintes affines sur les indices correspondant aux dimensions, permettant une description plus fine (mais sans trou) que le pavé.

2.5.2.2 Restriction dynamique

Lorsque les domaines ou les références ne permettent pas d'appliquer une restriction statique, on peut toutefois produire un code particulier qui calculera à l'exécution un domaine restreint [93]. Ces domaines calculés doivent donc être manipulables à l'exécution, ils sont représentés par exemple par une liste d'indices, le compilateur générant des boucles pour manipuler ces listes. Par exemple, en considérant que le tableau C est dupliqué, on pourra avoir la mise en œuvre suivante (pour 4 processus) :

$Exec(\{A[C[i]]\},\ \{A[C[i]] = B[i]\},\ \{i = 0..999\}) \equiv$

soit $L[]$ une table de listes de numéros d'itération
indicée par les numéros de processus
pour i **de** $moi*250$ **à** $moi*250+249$
ajouter i à $L[owner(A[C[i]])]$
fpour
Regrouper les $L[]$ des différents processus
pourtout $i \in L[moi]$
$A[C[i]] = B[i]$
fpourtout

Chaque processus calcule une partie de la répartition du domaine d'itération entre les processus. On regroupe ensuite ces contributions de sorte que chaque processus connaisse la portion de domaine qui est à sa charge. Le parcours du domaine restreint peut alors être effectué. On évite ainsi de parcourir l'ensemble du domaine global $\{i = 0..999\}$ moyennant une coopération entre les processus.

2.5.3 Communications

Les opérations *Refresh* et *Get* effectuent des copies de données distantes. Les définitions du paragraphe 2.4 mettent en jeu des transferts d'ensembles de données qui doivent être mis en œuvre en utilisant les mécanismes de communication des machines d'exécution sous-jacentes.

La latitude de choix de mise en œuvre est plus grande dans le cas des machines à messages que dans celui des machines à MVP. La suite de ce paragraphe concernera donc essentiellement la mise en œuvre de l'opération *Refresh* sur une machine à messages.

Objectif

Le travail de la machine abstraite consiste d'abord à déterminer les différents émetteurs et récepteurs. Les communications se font alors en trois phases.

Pour les émetteurs, on effectue :

- *Le calcul des ensembles de données à émettre* qui peut être partiellement ou totalement fait à la compilation. Dans le cas où une partie du calcul est fait à l'exécution, on doit pouvoir représenter les ensembles en mémoire.
- *La constitution des tampons d'émission* d'après la description des ensembles (empaquetage).
- *L'émission effective des contenus des tampons.*

Pour les récepteurs, on a les opérations duales, c'est-à-dire :

- *Le calcul des ensembles de données à recevoir.*
- *La réception dans les tampons de réception.*
- *La mise à jour de la représentation locale des données distantes* (dépaquetage).

Optimiser les communications consiste donc à diminuer le coût de préparation des communications (en mémoire et en temps) ainsi qu'à réduire les temps de transfert [3] proprement dits. Il faut donc minimiser :

- le temps de calcul des ensembles à l'exécution ;
- le temps de constitution des tampons de communication à l'exécution ;
- la taille des tampons d'émission/réception ;
- la taille de la représentation des ensembles en mémoire ;
- le nombre de messages (afin de ne pas être pénalisé par leur latence) ;
- le nombre d'éléments superflus (éléments échangés du fait des approximations utilisées dans la description des ensembles) ;
- le nombre d'éléments redondants (éléments échangés en plusieurs exemplaires).

La minimisation de tous ces paramètres est contradictoire : par exemple, approcher les ensembles à communiquer par des pavés augmente la compacité de la représentation des ensembles en mémoire mais peut augmenter le nombre d'éléments superflus et donc le volume de données échangées. Les optimisations doivent donc réaliser un compromis, en prenant éventuellement en ligne de compte des caractéristiques de la machine d'exécution (latence et débit des communications, coût des copies mémoire, coût d'allocation...). La prise en compte de telles caractéristiques ne rentre pas en contradiction avec les critères d'indépendance vis à vis des architectures cibles et de portabilité dans la mesure ou elles constituent un ensemble de

3. On considère que le temps de transfert d'un message a une composante fixe (la latence) et une composante proportionnelle à sa taille.

paramètres communs à un grand nombre de machines dont seules les valeurs varient d'une machine à l'autre.

Toutes les mises en œuvre des optimisations des communications doivent évidemment garantir l'absence d'interblocage sous l'hypothèse de files FIFO infinies. De plus, il est souhaitable qu'une hypothèse plus faible (taille de files finie) soit faite. Même s'il n'est pas possible de décider de la taille des files nécessaire pour l'ensemble du programme [32], le rafraîchissement d'un ensemble de données autorise la mise en place d'un contrôle de flux — localisé dans la mise en œuvre d'*un Refresh* — permettant de réduire le risque de saturation des files. Ce contrôle de flux peut éventuellement prendre en compte des paramètres de la machine cible.

Optimisations

L'optimisation principale, couramment appelée *vectorisation*, a pour objectif de diminuer le nombre de messages. Elle consiste à communiquer un ensemble d'éléments « contigus » dans le tableau séquentiel (portion de ligne ou de colonne d'une matrice par exemple) sans hypothèse sur la contiguïté de la représentation locale des éléments. Deux optimisations différentes sont couvertes par le terme de vectorisation :

- *La communication directe* dans laquelle l'ensemble communiqué dans un message est un ensemble d'éléments contigus dans la représentation locale à la fois chez l'émetteur et le récepteur, rendant inutile l'empaquetage et le dépaquetage des données.
- *L'agrégation de messages* qui consiste à regrouper plusieurs ensembles de données dans un seul message. Le nombre de messages est diminué mais il est nécessaire d'utiliser des tampons intermédiaires, le gain apporté par l'agrégation de messages est donc moins grand, à taille de données égale, que l'utilisation de communications directes.

Par ailleurs, on peut exploiter certaines opérations de communication de haut niveau comme les *communications collectives* qui sont souvent implantées efficacement, bénéficiant parfois de dispositifs matériels spécifiques. C'est particulièrement vrai pour la diffusion qui doit être préférée à la répétition de messages point-à-point.

Enfin, il est possible d'utiliser le *recouvrement calcul/communication* qui permet de diminuer non pas le temps réel de communication mais le temps apparent, en tirant parti du fait que, sur de nombreuses plates-formes, les calculs peuvent se faire en parallèle avec une partie du transfert de données.

2.6 État de l'art

Les travaux sur la compilation dirigée par les données visent essentiellement des machines d'exécution à messages. Nous présentons d'abord des travaux effectués dans

le cadre du développement d'environnement complets de compilation pour machine à messages. Il s'agit des projets Vienna Fortran et Fortran D dont les objectifs et le déroulement se rapprochent de ceux du projet PANDORE. Ces descriptions seront faites à l'aide des opérations de la machine abstraite que nous venons de définir, confirmant ainsi leur généralité (une présentation suivant l'optique des auteurs peut être trouvée dans [3]). Ceci nous permettra de dégager les points de ressemblance et les différences entre les deux projets.

La dernière partie de ce paragraphe complétera l'étude d'un des points clés de la mise en œuvre des machines abstraites qu'est la gestion des données distribuées en décrivant des travaux plus récents qui étendent les techniques intégrées dans les compilateurs cités plus haut.

2.6.1 Vienna Fortran

Le *Vienna Fortran Compilation System* [103] est un système de parallélisation automatique de programmes exprimés en Vienna Fortran [35] en vue de leur exécution sur une machine parallèle à mémoire distribuée. Le code source est traduit en un code SPMD exprimé en Fortran et appelant des primitives de communications de la bibliothèque Parmacs [34] ou du système NX/2 pour l'intel iPSC/860. Ce projet s'appuie sur le restructureur interactif SUPERB [101] dont il reprend les techniques de parallélisation de codes réguliers.

Le langage Vienna Fortran ajoute un certain nombre d'annotations à Fortran. Elles permettent la définition de grilles de processeurs virtuels, la distribution directe des tableaux sur les processeurs (BLOCK, CYCLIC ou définies entièrement par l'utilisateur) ainsi que l'alignement de tableaux. La redistribution et le réalignement dynamique sont autorisés. Un constructeur FORALL permet de spécifier des boucles parallèles.

Le compilateur emploie deux schémas de compilation que nous allons décrire ci-dessous.

2.6.1.1 Analyse de recouvrement

La compilation des nids de boucles réguliers[4] suit la règle des écritures locales en masquant les instructions élémentaires et en insérant des communications pour produire des opérations similaires aux *Refresh* et *Exec* élémentaires décrits au paragraphe 2.2.1.

L'*analyse de recouvrement*, décrite dans la thèse de Michael Gerndt [51], est utilisée pour définir la représentation des tableaux distribués ; elle sert également

4. Les bornes de boucles et les expressions d'indices d'accès aux tableaux doivent être des fonctions affines d'un seul indice de boucle englobant.

de base à l'augmentation du grain de l'opération *Refresh* ainsi qu'aux optimisations ultérieures appliquées aux *Refresh*.

Elle a pour but de déterminer une approximation, pour chaque processus et chaque tableau distribué, de la partie distante du tableau à laquelle le processus accède en lecture dans un nid de boucle. Cette analyse ne s'applique qu'aux distributions qui associent un seul bloc à chaque processeur ; elle considère une instruction élémentaire située dans un nid de boucle. Pour chaque référence en lecture, on détermine une zone de recouvrement spécifique à chaque processeur, c'est la zone rectangulaire englobant la partition locale et les données distantes susceptibles d'être lues durant l'exécution du nid de boucle. Cette zone est décrite par un descripteur de recouvrement (DDR_p), spécifique à un processeur p, donnant dans chaque dimension les deux extensions nécessaires à la partition locale. Afin de respecter le modèle SPMD, on prend le maximum des extensions des DDR_p pour en obtenir qu'un seul DDR.

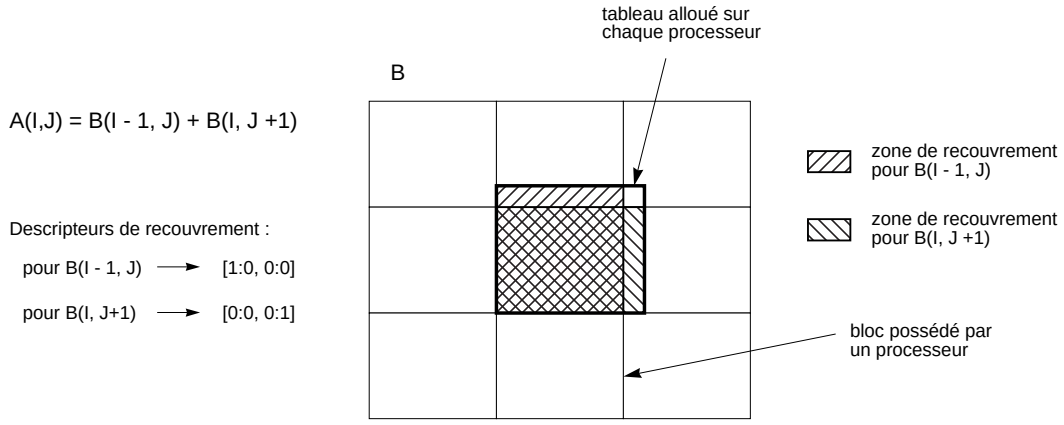


FIG. 2.3. – *Analyse de recouvrement*

Gestions des tableaux distribués

Les descripteurs de recouvrement servent à l'allocation des tableaux distribués. Sur chaque processeur, l'espace pour les données locales et distantes nécessaires est alloué dans un tableau dont la taille dans chaque dimension est égale à la taille de la partition locale plus celle donnée par le descripteur de recouvrement. Si plusieurs références à la même variable sont présentes, on fusionne d'abord les différents descripteurs en un seul en prenant les plus grandes valeurs des extensions (voir figure 2.3). Les accès aux données se font donc de façon uniforme via une conversion de chaque indice du vecteur d'accès de la forme $i \rightarrow (i - moi \times B)$ où moi est l'identité du processeur et B est la taille du bloc dans la dimension concernée. On peut noter que cette gestion des tableaux distribués conserve un maximum de contiguïté des éléments puisque, pour tous les éléments alloués localement, le fait

qu'ils soient contigus dans le tableau séquentiel implique le fait qu'ils le soient aussi dans le tableau local.

Optimisation de la génération des Refresh

Les *DDR* sont d'abord utilisées pour désigner les processeurs chez lesquels les données sont à rafraîchir, l'opération *Refresh*(*Ref*, *DDR*) indiquant que les processeurs pour lesquels *Ref* appartient à *DDR* doivent recevoir la donnée émise par le propriétaire de *Ref*.

Un algorithme fondé sur une analyse de dépendances permet de déterminer à quel niveau de boucle il est possible de placer les *Refresh* qui prennent alors en compte un domaine. Les domaines sont représentés par des sections de tableaux décrites dans chaque dimension par un triplet (borne inférieure, borne supérieure, pas). À cette étape, les communications sont vectorisées.

Deux autres optimisations sont ensuite appliquées :

- Les *Refresh* ne générant aucune communication et ceux générant des communications redondantes (entièrement réalisées par un *Refresh* précédent) sont éliminés.

- Afin de regrouper des messages, on fusionne dans certains cas plusieurs *Refresh* en un seul prenant en paramètre un ensemble de triplets (référence, *DDR*, domaine).

Mise en œuvre des Refresh/Exec

C'est le compilateur qui effectue la restriction des domaines de calcul de l'opération *Exec*. Une analyse symbolique des domaines et des distributions permet le partitionnement (*strip mining* [99]) des boucles décrivant le domaine, aboutissant à la génération des bornes restreintes paramétrées par l'identité du processeur. Les bornes des boucles ne tiennent pas compte de la déclaration du tableau local, elles restent donc globales.

L'opération *Refresh* est quant à elle optimisée à l'exécution. L'exécutif réalise notamment l'union des sections de tableau et des descripteurs de recouvrement qui permet d'éviter des envois redondants. Cette union est effectuée à l'exécution et non pas à la compilation afin de ne pas rajouter d'éléments superflus résultant de la fusion conservative des *DDR* (comme cela est fait pour l'allocation des tableaux locaux). Ces calculs d'union ne sont pas pénalisants compte tenu de la compacité des représentations des sections de tableau et des *DDR* ainsi que de la faible complexité des algorithmes mis en jeu. Ces unions sont toutefois liées à l'utilisation de tampons de communications supplémentaires ainsi qu'à des opérations d'empaquetage/dépaquetage des données.

Il faut noter que les approximations engendrées par l'analyse de recouvrement peuvent provoquer des surcoûts importants tant en ce qui concerne l'allocation mémoire que les communications et par ce fait diminuent son applicabilité réelle.

2.6.1.2 Inspecteur / Exécuteur

Le constructeur FORALL permet en Vienna Fortran la spécification de boucle parallèle. On utilise le FORALL notamment pour permettre l'optimisation de boucles où apparaissent des indirections dans les accès aux tableaux distribués. On y applique une technique d'optimisation dynamique connue sous le nom d'inspecteur/exécuteur [93]. La mise en œuvre de cette technique repose sur l'utilisation de la bibliothèque Parti, développée en vue de l'exécution parallèle de programmes agissant sur des structures de données irrégulières [41]. Cette bibliothèque a été également utilisée par d'autres projets dans des cadres similaires [40, 59, 29]. L'exécution de la boucle se déroule en deux phases : l'inspecteur et l'exécuteur. L'inspecteur analyse dynamiquement les communications dans la boucle, en calcule une description et convertit les adresses globales en adresses locales. L'exécuteur effectue les communications proprement dites et exécute les calculs. Nous décrivons ici les différentes étapes de l'inspecteur et de l'exécuteur réparties entre les opérations *Refresh* et *Exec* dans le cadre du FORALL bien que ces opérations ne soient pas explicitement générées.

$$\begin{array}{l}\textsf{FORALL } i = 1, N \\ \quad A(C(i)) = B(C(i)) \\ \textsf{ENDDO}\end{array} \quad \Rightarrow \quad \begin{array}{l}\mathit{Refresh}(\mathcal{R}(i),\ \mathcal{R}_P(i),\ \mathcal{D}(i)) \\ \mathit{Exec}(\mathcal{R}_P(i),\ \mathcal{S}(i),\ \mathcal{D}(i))\end{array} \quad \left\| \begin{array}{lcl} \mathcal{R}_{\mathcal{P}}(i) & \equiv & \{A(C(i))\} \\ \mathcal{R}(i) & \equiv & \{B(C(i))\} \\ \mathcal{D}(i) & \equiv & 1..N \\ \mathcal{S}(i) & \equiv & A(C(i)) = B(C(i)) \end{array} \right.$$

L'opération *Refresh* est transformée par le compilateur en un code effectuant les actions suivantes :

- Le domaine $\mathcal{D}(i)$ est parcouru en examinant $\mathcal{R}_P(i)$ et $\mathcal{R}(i)$ afin de déterminer, pour un processeur p :
  - la liste des itérations locales ($exec_p$) en fonction du possesseur de $\mathcal{R}_P(i)$,
  - pour chaque variable v de $\mathcal{R}(i)$, la liste des indices des références distantes pour les itérations locales ($global\_ref_p^v$).
- Pour chaque variable v de $\mathcal{R}(i)$, à partir de $exec_p$, $global\_ref_p^v$ et de la distribution de v, et après une phase de communication globale, sont déterminés :
  - la taille du tampon de réception,
  - un ordonnancement décrivant les ensembles de données à émettre et à recevoir ainsi que l'emplacement des données reçues.

  On prépare également durant cette étape le travail de l'*Exec* en constituant, pour chaque variable lue v, une table d'indirection ($local\_ref_p^v$). Pour cela, on

utilise les fonctions de conversion d'indices dérivées de la distribution de v. Ces fonctions de conversion peuvent faire intervenir des opérations de division ou de modulo. Pour les ordonnancements et les tables d'indirection, les redondances présentes dans $global\_ref_p^v$ sont éliminées grâce à l'utilisation de tables de hachage.

- Les tampons de réception sont alloués pour étendre la partition locale.
- On procède à l'échange des données conformément à l'ordonnancement précalculé.

L'opération *Exec* est transformée en un code réutilisant des listes $exec_p$ et $local\_ref_p^v$ calculées par le *Refresh*. Ce code effectue le parcours des itérations de $exec_p$ pour exécuter $S(i)$. On accède aux données locales et reçues de façon uniforme via la table d'indirection $local\_ref_p^v$.

Il faut noter que la complexité de la phase d'inspecteur est la même que celle de la boucle séquentielle et qu'elle fait intervenir des calculs et des communications coûteuses. Ceci fait que la stratégie de l'inspecteur/exécuteur n'est vraiment applicable que lorsque la phase d'inspecteur peut être réutilisée pour plusieurs itérations, notamment en la sortant hors d'une boucle.

La représentation des tableaux distribués construite par l'inspecteur est spécifique à une boucle donnée, et en particulier différente de la représentation définie dans les cas réguliers par l'analyse de recouvrement. Ceci empêche donc toute réutilisation des représentations entre les parties de code régulières et irrégulières.

2.6.2 Fortran D

Le projet Fortran D [63, 64] a mis en œuvre un compilateur réalisant la transformation de programmes séquentiels écrits en Fortran et comportant des spécifications de distribution de tableaux en programmes parallèles, exprimés en Fortran 77, faisant appel à des routines de communications du système NX/2 tournant sur la machine Intel iPSC/860.

Le langage Fortran D [48] offre de multiples instructions de distribution de données, il permet notamment de définir des tableaux virtuels (DECOMPOSITION) qui peuvent être distribués de façon régulière ou entrelacée (BLOCK, CYCLIC). Les tableaux de données sont alignés sur ces décompositions afin d'être distribués. Fortran D fournit par ailleurs un constructeur de boucles parallèles FORALL (de type data-parallèle, différent de celui de Vienna Fortran) et autorise la redistribution.

Le schéma de compilation, dont les initiateurs du projet sont à l'origine [33], suit la règle des écritures locales et le modèle SPMD. Le compilateur a été bâti dans le cadre de l'environnement d'analyse et de transformation de programme Parascope [13]. La compilation est constituée d'un certain nombre d'analyses et de

transformations de code qui visent à produire directement du code comportant des envois et des réceptions de messages. Nous les présentons ici en termes du schéma de compilation présenté en début de chapitre qui produit les opérations intermédiaires *Refresh* et *Exec*.

Analyse préalable

L'analyse du programme source porte sur les instructions apparaissant dans les boucles non nécessairement parallèles. Pour une instruction donnée, elle vise principalement à définir pour chaque processeur et pour chaque niveau de boucle :

- L'ensemble des itérations locales *IL*.
  C'est le résultat de l'intersection du domaine d'itération avec l'image de la partition locale du tableau écrit par l'inverse de la fonction d'accès à ce tableau.

- L'ensemble des données distantes pour chaque référence lue.
  Il est défini par l'image de *IL* par la fonction d'accès en lecture, auquel on soustrait la partition locale.

Le compilateur utilise une structure de données appelée Descripteur de Section Régulière (*DSR*) pour calculer une approximation de ces ensembles [10]. Les *DSR* offrent une représentation compacte de domaines triangulaires-rectangles[5].

Application du schéma de compilation

Les instructions d'une boucle sont tout d'abord divisées en groupes caractérisés par des instructions possédant le même ensemble *IL*. Les boucles contenant un seul groupe d'instructions sont dites uniformes. Le schéma de compilation est appliqué aux boucles parallèles uniformes ou à défaut aux instructions élémentaires. Plusieurs transformations de programmes préalables sont appliquées pour augmenter le nombre de nids uniformes (distribution et fusion de boucle notamment).

Le compilateur réalise la restriction des domaines de calculs décrits dans les *Exec* par partitionnement des boucles parcourant le domaine. Dans certains cas [97], le compilateur parvient à prendre en compte la déclaration de la partition locale dans la définition des bornes réduites qui sont alors locales.

Pour compléter le calcul des ensembles de données distantes, une analyse de dépendance détermine le niveau de boucle auquel on peut effectuer les communications [11]. Cette analyse permet de sortir certains *Refresh* des boucles. De plus, les *Refresh* décrivant des *DSR* qui se chevauchent ou se jouxtent sont fusionnés (*message coalescing*). Ceci élimine les transferts redondants.

5. Dans la version actuelle du compilateur, les approximations calculées sont rectangulaires, les analyses effectuées sont donc proches de celles de l'analyse de recouvrement de Vienna Fortran.

Lors de la mise en œuvre du *Refresh*, les ensembles à communiquer sont décrits par des *DSR*, réalisant ainsi la vectorisation des messages. De plus, tous les messages ayant un même destinataire sont regroupés en un seul, moyennant des copies et des empaquetages/dépaquetages (*message aggregation*). Par ailleurs, un certain nombre de cas ont été identifiés, pour lesquels des optimisations spécifiques peuvent être appliquées :

- Utilisation de communications collectives (diffusions, opérations globales, etc.) [82].
- Déplacement des envois au plus tôt et des réceptions au plus tard (*message pipelining*).
- Utilisation de communications non tamponnées (primitives isend et irecv du système NX/2) grâce auxquelles les copies mémoire/tampons systèmes peuvent être faites parallèlement aux calculs.

Représentation et accès aux données

Le compilateur Fortran D représente la partition locale d'un tableau par un tableau de même dimension mais dont la taille dans chaque dimension distribuée est réduite (divisée par le nombre de processeurs). L'accès aux éléments de la partition locale se fait via une conversion d'indice *global vers local* dans chaque dimension distribuée de la forme $i \rightarrow (i - moi \times B)$ (moi est l'identité du processeur et B est la taille du bloc dans la dimension concernée) pour les distributions attribuant un seul bloc par processeur et de la forme $i \rightarrow (i-1)\, div\, B$ pour les distributions cycliques. Lorsque le compilateur parvient à rendre locales les bornes de boucles, ces conversions sont inutiles, mais dans ce cas, les références aux variables d'itération ne faisant pas partie d'une expression d'indice nécessitent des conversions inverses (*global vers local*).

Les *DSR* qui décrivent les ensembles de données distantes calculés lors de la phase d'analyse initiale servent à sélectionner le type de gestion des éléments reçus depuis d'autres processeurs.

- Lorsque les éléments distants forment une frontière de la partition locale, la déclaration de la partition locale est étendue de façon à contenir les éléments distants (technique du recouvrement ou *overlap*). Les accès aux éléments distants se font alors de la même manière que les accès aux éléments de la partition locale (uniformité des accès).
- Dans les autres cas, des tampons mémoire sont alloués statiquement ou dynamiquement. Par conséquent, les accès ne peuvent plus être uniformes. L'utilisation de tampons nécessitent la séparation des itérations purement locales des itérations requérant des données distantes dans la mise en œuvre de l'*Exec* ; pour ces dernières, les références aux éléments de tableaux sont transformées

par le compilateur en références aux tampons, avec une éventuelle linéarisation.

2.6.3 Gestion des données distribuées

Les techniques de gestion des données distribuées telles que celles que nous avons décrites dans l'étude des projets Vienna Fortran et Fortran D sont des techniques de base qui, à l'heure actuelle, sont les seules à avoir été complètement intégrées à des environnements de compilation par distribution de données.

Plusieurs projets de recherche [36, 62, 23, 25] visent à étendre ces techniques pour prendre en compte des tableaux ayant des distributions générales (au sens d'HPF). Ils considèrent en général l'alignement et la distribution en CYCLIC(k).

Dans la plupart des travaux, le choix de la représentation des tableaux distribués est dicté par la volonté d'occuper l'espace mémoire minimal pour la partition locale. La représentation des éléments distants reçus n'est généralement pas traitée de la même manière, empêchant l'uniformité des accès.

La définition du mécanisme d'accès aux données locales dépend fortement des techniques de compilation dont la description dépasse le cadre de cette thèse (voir [79]). L'objectif généralement visé est la description des ensembles accédés (ensemble des éléments à émettre, à recevoir ou à calculer dans un nid de boucles) directement dans la représentation locale. Nous illustrons ici un exemple de technique de représentation et d'accès présenté dans [36] et qui a inspiré plusieurs travaux (*e.g.* [62]).

2.6.3.1 Représentation des tableaux

Pour chaque dimension d'un tableau A, la distribution d'un tableau est définie par

- la taille du tableau : n ;
- la fonction d'alignement sur le *template* T :
  $A(\ldots, i, \ldots)$ est aligné avec $T(\ldots, ai + b, \ldots)$;
- la fonction de distribution du *template* sur P processeurs : des blocs de k éléments sont attribués cycliquement aux processeurs (distribution CYCLIC(k)).

Les éléments locaux sont stockés dans un tableau de même dimension que le tableau original, en fonction des paramètres de distribution et du nombre de processeurs. La figure 2.4 illustre la représentation d'un tableau mono-dimensionnel, on y a fait apparaître la structure à la fois globale et locale du *template* T bien qu'elle ne donne lieu à aucune allocation mémoire. L'extension aux cas multi-dimensionnels se fait en appliquant la technique à chaque dimension qui possède sa propre liste de paramètres (P, n, a, b, k).

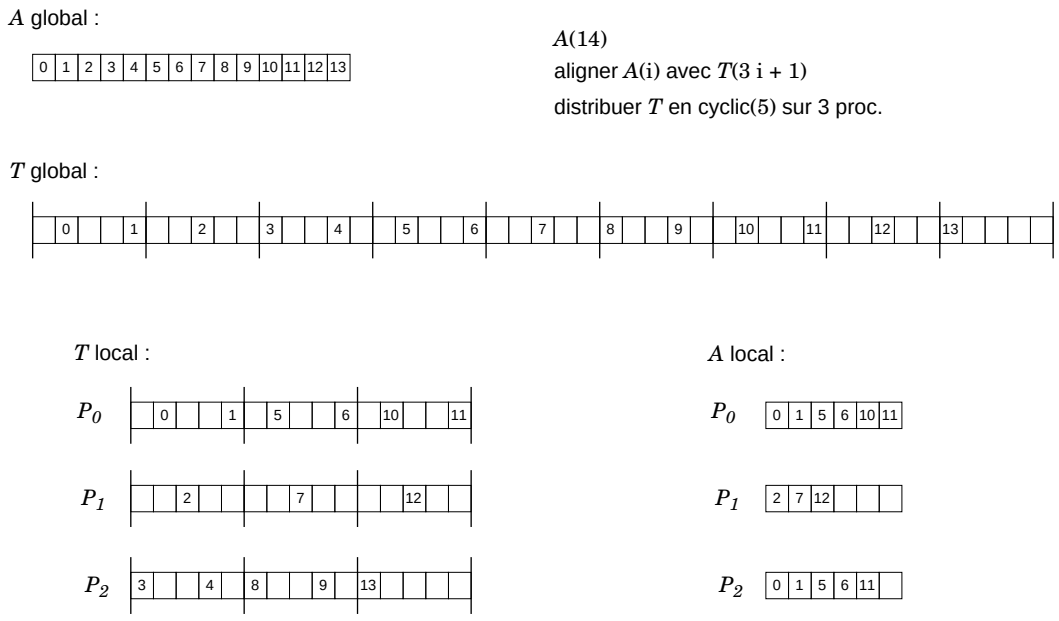


FIG. 2.4. – *Représentation locale compactée d'un tableau mono-dimensionnel*

2.6.3.2 Accès

Les auteurs proposent une méthode d'accès aux données locales fondée sur l'utilisation d'automates d'états finis. Ils considèrent un tableau distribué et la description d'un ensemble d'éléments globaux accédés (dans une boucle) et se proposent de calculer la suite des emplacements locaux atteints sur chacun des processeurs. L'ensemble des éléments globaux accédés est supposé être décrit par une section de tableau $A(l : h : s)$ où l est la borne inférieure, h la borne supérieure et s le pas. La technique est exposée dans le cas mono-dimensionnel puis étendue aux cas multi-dimensionnels.

La liste des éléments accédés localement par un processeur donné est définie par un emplacement de départ dans le tableau local et par la succession des sauts à effectuer pour passer d'un élément à l'élément suivant. Cette séquence est modélisée par un automate d'états fini comportant au plus k états. Cet automate est identique pour tous les processeurs, il est calculé en fonction de P, k et du pas d'alignement s. L'état de départ et le nombre de transitions à effectuer pour couvrir l'ensemble de la part locale de la section $A(l : h : s)$ sont spécifiques à chaque processeur, ils sont calculés en fonction de P, k, s mais aussi de l et h. Ces calculs sont fondés sur la résolution de k équations diophantiennes dont on considère, pour chacune d'entre elles, les solutions minimales et maximales positives[6]. En représentant l'automate

6. L'algorithme utilisé a une complexité en $O(\min(k \log k + \log s,\ k \log k + \log P))$, certains cas particuliers sont reconnus et font l'objet de simplifications.

par une table Δ de longueur L_Δ la suite des accès *ind* à la part locale de la section de A est donné par :

$ind = ind\_début\,;\ i = 0\,;$
tantque $(ind \leq ind\_fin)$
accéder à $A[ind]$
$ind = ind + \Delta[i]$
i $= (i + 1)\ mod\ L_\Delta$
fintantque

L'extension aux cas multi-dimensionnels se fait en appliquant la méthode de calcul de l'automate à chaque dimension. La boucle d'accès devient une boucle à plusieurs niveaux.

Cette technique permet d'obtenir un accès rapide aux éléments locaux. Toutefois, la gestion des tableaux n'est pas uniforme, la représentation choisie excluant la prise en compte des éléments reçus. Par ailleurs elle n'est pas indépendante des instructions du programme puisque les automates sont calculés en fonction de la section de tableau décrivant les accès globaux spécifiés dans le source. Si l'on considère le cas général où les bornes et pas de sections de tableaux ne sont connus qu'à l'exécution, le calcul de l'automate est à réaliser à l'exécution pour chaque référence de chaque boucle, ce qui peut ralentir l'exécution de façon non négligeable.

Chapitre 3

La machine abstraite Pandore

Nous présentons dans ce chapitre la machine abstraite associée à l'environnement de programmation PANDORE. La structure et les différents composants de l'environnement PANDORE sont d'abord décrits puis nous détaillons la mise en œuvre de la machine abstraite sur deux machines d'exécution : une machine à messages (la POM) et une machine à mémoire virtuelle partagée (Koan).

3.1 L'environnement Pandore II

3.1.1 Structure

L'environnement PANDORE développé dans l'équipe Pampa constitue un environnement de programmation des APMD. Il comprend un compilateur, un exécutif et plusieurs outils d'analyse d'exécution.

Le langage source, avec une syntaxe proche de celle du langage *C*, permet d'exprimer un algorithme et de spécifier une distribution des données. Pour pouvoir profiter des nombreuses applications qui vont être écrites en HPF [47], un traducteur HPF vers C-PANDORE a été conçu et ajouté à l'environnement [69].

Le compilateur C-PANDORE a été écrit par Olivier Chéron en Caml [37]. Il met en œuvre une chaîne complète de compilation. Il se compose d'une partie frontale générant une forme interne relativement indépendante du langage source, d'un transformateur de cette forme interne séquentielle en une forme décrivant un programme parallèle SPMD et d'une partie terminale comprenant un générateur de code qui produit du source *C* contenant des appels à l'exécutif.

L'exécutif est constitué d'un ensemble de primitives permettant de s'interfacer avec une machine d'exécution donnée, que ce soit une machine à messages ou une machine à mémoire virtuelle partagée. Il autorise également l'instrumentation du code généré afin de pouvoir analyser l'exécution des programmes distribués. Les mé-

canismes d'instrumentation ainsi que l'exploitation de ses résultats pour l'analyse des exécutions distribuées sont décrits dans le chapitre 4.

La figure 3.1 décrit la structure du cœur de l'environnement PANDORE. La partie antérieure du compilateur produit à partir du source *C*-PANDORE un code SPMD en *langage intermédiaire* (langage d'entrée des machines abstraites). Cette forme intermédiaire, qui contient les opérations *Refresh*, *Exec*, *Sync* et *Get*, est transformée en un code *C* constituant la sortie du compilateur. Ce code fait appel aux primitives de l'exécutif qui permet l'interface avec la machine d'exécution (machine à messages POM ou machine à MVP Koan). La compilation du code *C* généré permet d'obtenir au final un code machine pouvant s'exécuter sur l'architecture parallèle cible.

Certaines des actions de la machine abstraite peuvent être effectuées à la compilation. C'est pourquoi la mise en œuvre de cette machine abstraite est répartie entre le compilateur et l'exécutif.

3.1.2 Le langage

3.1.2.1 Généralités

Le langage PANDORE possède les caractéristiques communes à la plupart des langages séquentiels impératifs comme Pascal ou Fortran. On y retrouve les constructions habituelles telles que l'affectation, la conditionnelle, l'itération, la structure de bloc, la déclaration de procédures ou de fonctions. Sa syntaxe est basée sur un sous-ensemble du langage *C*. Ce choix n'est pas ici fondamental, l'utilisation de la syntaxe d'un autre langage ne remettant pas en cause l'ensemble de l'environnement.

Tous les types de bases de *C* sont utilisables mais le seul constructeur de type disponible est le tableau. Les pointeurs, structures et unions, qui posent des problèmes de distribution spécifiques au langage *C*, n'ont pas été inclus au langage. En conséquence, les tableaux de PANDORE sont, contrairement à ceux de *C*, de vrais constructeurs multi-dimensionnels.

La structuration d'un programme PANDORE se fait essentiellement à l'aide de définitions de phases distribuées. Deux autres constructeurs ont été également ajoutés au langage : les fonctions closes et les macros.

- La phase distribuée se présente sous la forme d'une procédure, introduite par le mot-clé `dist`, dans laquelle on précise une distribution pour chacun des paramètres formels. Ces spécifications de distribution guideront le compilateur pour distribuer sur les processeurs le corps de la phase distribuée. Une phase distribuée ne peut être appelée que du programme principal.
- Les fonctions closes sont des fonctions (rendant un résultat) n'acceptant que des paramètres scalaires dont le passage s'effectue par valeur. Dans le corps

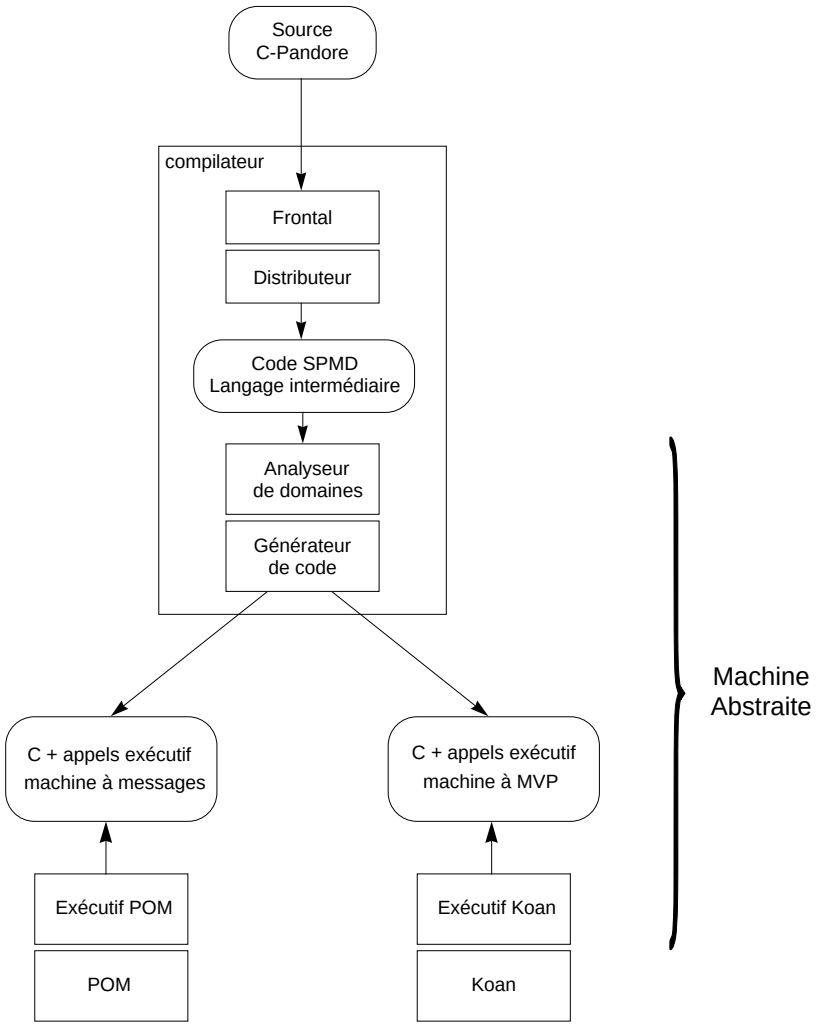


FIG. 3.1. – *Cœur de l'environnement* PANDORE

de ces fonctions, on ne peut faire référence à des scalaires ou des tableaux déclarés à l'extérieur. Les fonctions closes n'ont donc pas d'effet de bord, elles peuvent être vues comme des opérateurs du langage.

- Les macros sont des sortes de procédures dont les paramètres scalaires ou tableaux sont passés par nom avec une syntaxe identique au passage par adresse de *C*. Les corps des macros sont expansés dans le code de l'appelant au début de la compilation.

Si les macros et les fonctions closes sont des constructions dont le seul but est de faciliter l'écriture de programmes, la phase distribuée est quant à elle un élément clé du langage car c'est elle seule qui entraîne la génération de code parallèle. En effet, l'exécution d'un programme PANDORE est une succession de phases séquentielles — correspondant aux instructions, appels aux macros et fonctions closes dans le

programme principal — entrecoupées de phases parallèles représentées par les appels aux phases distribuées.

La figure 3.2 donne un exemple de programme PANDORE. Il comprend un programme principal (fonction `main`) dans lequel on appelle une macro (`init`) et une phase distribuée (`calc`). Dans cette phase distribuée, il est fait appel à la fonction close `norm` et à la macro `init`.

À noter que la notion d'adresse n'existe pas dans les programmes *C*-PANDORE puisque, contrairement au langage *C*, aucune définition de la représentation des données n'est fournie au niveau du langage.

3.1.2.2 La phase distribuée

Passage de paramètre

Le passage de paramètre pour une phase distribuée se fait par nom, la notation pour les paramètres effectifs est celle du passage par adresse du langage *C*. Dans la déclaration du paramètre formel, on précise le mode d'utilisation de la variable : `IN`, `OUT` ou `INOUT` suivant que la valeur du paramètre est significative respectivement à l'entrée de la phase uniquement (la valeur du paramètre effectif reste inchangée après l'exécution de la phase), à la sortie de la phase uniquement ou à l'entrée et à la sortie de la phase. Ce mode est identique à celui utilisé dans Ada ou Fortran 90.

Répartition de données

Dans une phase distribuée, les variables sont réparties sur les processus, cette répartition guidant le processus de compilation.

- Le premier type de répartition d'une variable est la duplication : la variable est intégralement présente sur tous les processus.
- Le second type de répartition est la distribution, il n'est applicable qu'aux tableaux. Ceux-ci sont découpés en blocs et les blocs sont distribués sur l'ensemble des processus.

Classes de variables

On distingue trois classes de variables au sein d'une phase distribuée. Suivant leur classe, on associe aux variables une répartition et un mode d'utilisation.

- Les paramètres formels, scalaires ou tableaux (`c`, `A` et `B` dans l'exemple de la figure 3.2). On doit spécifier une répartition seulement pour les tableaux, les scalaires étant systématiquement dupliqués. On indique également pour chaque paramètre un mode d'utilisation `IN`, `OUT`, `INOUT`.

```
#define N 1024
#define P 16

/* ————————— Fonction close ————————— */
double norm(double a, double b)
{ return(a / sqrt(b/N)) ; }

/* ————————— Macro ————————— */
macro init(double T[N][N])
{
 int i,j ;
 for (i=0 ; i<N ; i++)
    for (j=0 ; j<N ; j++)
       T[i][j] = rand() ;
}

/* ————————— Phase distribuée ————————— */
dist calc(int c mode IN,
          double A[N][N] by block(N, N/P) map regular(0,1) mode INOUT,
          double B[N][N] replicate mode IN
         )
    int V[N] by block(2) map wrapped(0) ;
{
 int i,j ;
 init(A) ;
 for (i=0 ; i<N ; i++) {
    V[i] = B[i][0] ;
    for (i=0 ; i<N ; i++)
       A[i][j] = A[i][j] + c * norm(V[i], B[i][j]) ;
 }
}

/* ————————— Programme principal ————————— */
main()
{
 double X[N][N], Y[N][N] ;
 int v=3 ;
 init(Y) ;
 calc(&v, X, Y) ;
}
```

FIG. 3.2. – *Exemple de programme Pandore II*

- Les variables locales réparties, déclarées dans l'entête de phase, sont uniquement des tableaux (`V` dans l'exemple). On leur associe une répartition. Le mode d'utilisation est ici inutile car la portée de ces variables est limitée à la phase distribuée.
- Les variables locales, scalaires ou tableaux dont la déclaration apparaît dans le corps (`i`, `j` et `X` dans l'exemple). Leur déclaration est identique aux variables automatiques des fonctions C. Elles sont systématiquement dupliquées.

On peut noter que la répartition des variables d'une phase distribuée est propre à cette phase distribuée. Comme la répartition est attachée aux paramètres formels et aux variables locales, elle est identique pour tous les appels à la phase : ce n'est notamment pas un paramètre de la phase que l'on pourrait instancier différemment à chaque appel.

Distribution de tableaux

La distribution des tableaux de PANDORE se fait en deux étapes : la décomposition en blocs (fonction *block*) et le placement des blocs sur les processus (fonctions *map*). Soit la spécification de distribution générique suivante :

$$int\ V[h_0]\cdots[h_{n-1}]\ by\ block\ (s_0,\ldots,s_{n-1})\ map \left|\begin{array}{l} regular \\ wrapped \end{array}\right| (d_0,\ldots,d_{n-1})$$

- Décomposition en blocs (partitionnement).
  La fonction *block* permet de spécifier la taille des blocs dans chaque dimension (entiers s_i). On découpe donc chaque tableau en blocs rectangulaires de tailles égales[1].
- Placement des blocs.
  Les fonctions *map regular* et *map wrapped* indiquent sur quel processeur se trouve chaque bloc. Dans le cas *regular*, les blocs sont placés de façon contiguë par groupes de $PdivB$, P étant le nombre de processus indiqué par l'utilisateur sur la ligne de commande du compilateur et B le nombre total de blocs. Dans le cas *wrapped* les blocs sont attribués cycliquement aux processus. L'ordre suivant lequel les blocs ou les groupes de blocs sont associés aux processus est indiqué par les paramètres d_k qui forment une permutation de $(0,\ldots,n-1)$. Les blocs sont attribués d'abord dans la dimension d_0 puis la dimension d_1, etc. La figure 3.3 illustre plusieurs distributions possibles d'un tableau bidimensionnel.

1. Les blocs aux extrémités du tableau sont plus petits dans la dimension k si s_k ne divise pas h_k.

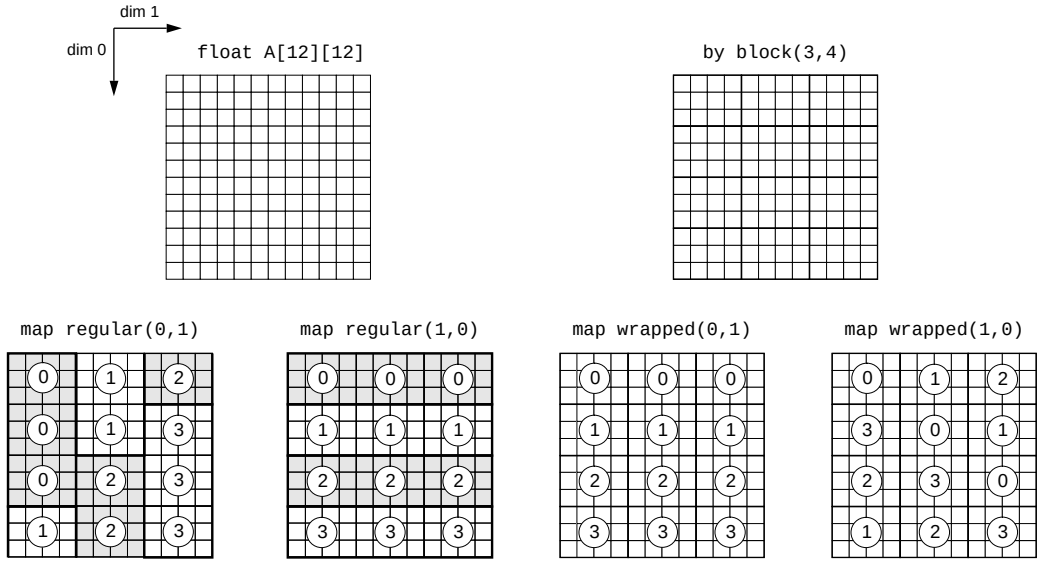


FIG. 3.3. – *Décomposition en blocs et placement des blocs sur 4 processus*

3.1.3 Le compilateur

3.1.3.1 Modèle de programme parallèle

Le modèle de programme du code actuellement produit par le compilateur est le modèle *maître-esclaves*, appelé ici *hôte-nœuds*; c'est-à-dire que la compilation d'un programme PANDORE produit deux codes C, un code pour l'hôte et un code pour les nœuds. Cette séparation est issue du modèle de programmation des architectures parallèles initialement visées par PANDORE (Intel iPSC). L'hôte exécute les instructions du programme principal (notamment les entrées/sorties) et déclenche les phases distribuées sur les processus nœuds. Le corps des phases distribuées est exécuté en parallèle sur les nœuds. Suivant le mode d'utilisation des paramètres des phases distribuées, les données correspondant aux paramètres sont envoyées de l'hôte aux nœuds au début des phases et rapatriées à la fin des phases.

Ce modèle doit être abandonné (pour obtenir un modèle purement SPMD) dans une version future du compilateur dans laquelle les imbrications d'appels de phases distribuées seront autorisées, le programme principal devenant lui-même une phase distribuée.

3.1.3.2 Le distributeur

La compilation des phases distribuées, réalisée par le *distributeur*, suit la règle des écritures locales et produit un code SPMD pour les machines abstraites PANDORE visant les machines à messages et les machines à MVP. Les opérations de ces machines

abstraites sont dérivées des opérations générales présentées au chapitre précédent. On distingue deux schémas de compilation qui produisent des spécialisations des opérations *Refresh*, *Exec*, *Sync* et *Get*. Le premier schéma de compilation, le *schéma de base*, peut être utilisé pour traduire n'importe quel programme source ; il opère au niveau d'une instruction. Le deuxième, le *schéma optimisé*, est appliqué à certains nids de boucles ; il permet d'obtenir une efficacité accrue du code généré.

Par souci de clarté nous décrivons dans la suite les deux schémas de compilation en terme de production non pas des *Refresh* et *Exec*, *Sync* et *Get* généraux mais de leurs versions spécialisées.

Le schéma de base

L'application du schéma de compilation de base permet de compiler l'intégralité du langage source. Ce schéma ne vise que les opérations adaptées à la machine à messages (*Refresh* et *Exec*). Nous verrons dans le paragraphe 3.3 pourquoi une version adaptée à la MVP n'est pas utile.

Nous illustrons ici son utilisation lors de la compilation d'une instruction d'affectation [4]. Soit $\mathcal{A}$ une affectation. On définit les ensembles ordonnés suivants :

- USE($\mathcal{A}$) : l'ensemble des références en lecture à des variables distribuées apparaissant dans $\mathcal{A}$.
- DEF($\mathcal{A}$) : l'ensemble des références en écriture à des variables apparaissant dans $\mathcal{A}$.

La compilation de l'affectation $\mathcal{A}$ produira la séquence d'opérations suivante :

Refresh_base(USE($\mathcal{A}$), DEF($\mathcal{A}$))
Exec_base(DEF($\mathcal{A}$), $\mathcal{A}$)

Comme une seule instruction est ici prise en compte et une seule instance de cette instruction est considérée, les opérations *Refresh_base* et *Exec_base* sont des versions simplifiées des opérations générales *Refresh* et *Exec* présentées au chapitre précédent. La spécification d'un domaine est notamment inutile :

- *Refresh_base*($\mathcal{R}$, $\mathcal{R}_P$) rend accessible aux processus qui possèdent les variables référencées dans $\mathcal{R}_P$ les valeurs des variables référencées dans $\mathcal{R}$.
- *Exec_base*($\mathcal{R}_P$, $\mathcal{A}$) fait en sorte que seuls les processus possédant une variable référencée dans $\mathcal{R}_P$ exécutent l'affectation $\mathcal{A}$.

Même si l'ensemble $\mathcal{R}_P$ est toujours un singleton, nous gardons une notation d'ensemble pour des raisons d'homogénéité.

Par exemple, l'affectation

$$X[i] \;=\; X[i+1] \;+\; Y[i]$$

sera traduite en

Refresh_base($\{X[i+1], Y[i]\}$, $\{X[i]\}$)
Exec_base($\{X[i]\}$, $X[i] = X[i+1] + Y[i]$)

Le schéma optimisé

La granularité du schéma de base, qui ne traite qu'une seule affectation, est source d'inefficacité. Un schéma de compilation optimisé a donc été défini, qui sépare les lectures des écritures pour un *nid commutatif*, c'est-à-dire un nid de boucles parfaitement imbriquées dont les itérations peuvent commuter [80]. C'est le cas notamment pour les boucles parallèles et les réductions.

Pour un nid de boucles $\mathcal{L}$ comprenant une unique affectation $\mathcal{A}$, on adopte les définitions suivantes :

- $\mathcal{I}$: un ensemble de variables libres correspondant aux variables d'itération de $\mathcal{L}$.
- $\mathcal{A}(\mathcal{I})$ l'affectation paramétrée par $\mathcal{I}$ constituant le corps du nid de boucle.
- DOM($\mathcal{L}$, $\mathcal{I}$) : le domaine de variation de $\mathcal{I}$ correspondant à l'espace d'itération de $\mathcal{L}$.
- USE($\mathcal{A}$, $\mathcal{I}$) : l'ensemble paramétré par $\mathcal{I}$ des références en lecture à des variables distribuées apparaissant dans $\mathcal{A}$.
- DEF($\mathcal{A}$, $\mathcal{I}$) : l'ensemble paramétré par $\mathcal{I}$ des références en écriture à des variables apparaissant dans $\mathcal{A}$.

On distingue les schémas de compilation produisant des opérations adaptées aux machines à messages et aux machines à MVP.

Machine à messages

Pour une machine à messages, la compilation d'un nid commutatif $\mathcal{L}$ produira la séquence d'opérations :

Refresh_opt(USE($\mathcal{A}$, $\mathcal{I}$), DEF($\mathcal{A}$, $\mathcal{I}$), DOM($\mathcal{L}$, $\mathcal{I}$))
Exec_opt(DEF($\mathcal{A}$, $\mathcal{I}$), $\mathcal{A}$, DOM($\mathcal{L}$, $\mathcal{I}$))

Les opérations *Refresh_opt* et *Exec_opt* sont également des versions simplifiées des opérations *Refresh* et *Exec* décrites au chapitre précédent. En effet, une seule affectation est considérée, même si plusieurs instances sont prises en compte via la donnée d'un domaine d'itération. Les opérations *Refresh_opt* et *Exec_opt* sont définies par :

- *Refresh_opt*($\mathcal{R}(\mathcal{I})$, $\mathcal{R}_P(\mathcal{I})$, $\mathcal{D}(\mathcal{I})$) rend accessible aux processus possédant une variable référencée dans $\mathcal{R}_P(\mathcal{I})$ une copie des valeurs des variables référencées dans $\mathcal{R}(\mathcal{I})$ pour toutes les valeurs des variables de $\mathcal{I}$ décrites par $\mathcal{D}(\mathcal{I})$.

- *Exec_opt*($\mathcal{R}_P(\mathcal{I})$, $\mathcal{A}(\mathcal{I})$, $\mathcal{D}(\mathcal{I})$) fait en sorte que seuls les processus possédant une variable référencée dans $\mathcal{R}_P(\mathcal{I})$ exécutent l'affectation $\mathcal{A}(\mathcal{I})$ pour toutes les valeurs des variables de $\mathcal{I}$ décrites par $\mathcal{D}(\mathcal{I})$.

Par exemple le nid commutatif

for $i = 0, N$
  for $j = i, N$
    $X[i][j] \ = \ X[i+1][j] \ + \ Y[i]$

sera traduit en

Refresh_opt($\{X[i+1][j], Y[i]\}, \{X[i]\}, \{0 \leq i \leq N, \ i \leq j \leq N\}$)
Exec_opt($\{X[i]\}, \{0 \leq i \leq N, \ i \leq j \leq N\}, X[i][j] \ = \ X[i+1][j] \ + \ Y[i]$)

Machine à MVP

Lorsque l'on vise une machine à MVP, la compilation d'un nid commutatif $\mathcal{L}$ produira la séquence d'opérations :

Sync_opt(USE($\mathcal{A}$, $\mathcal{I}$) $\cup$ DEF($\mathcal{A}$, $\mathcal{I}$), DOM($\mathcal{L}$, $\mathcal{I}$))
Exec_opt(DEF($\mathcal{A}$, $\mathcal{I}$), DOM($\mathcal{L}$, $\mathcal{I}$), $\mathcal{A}$)
Sync_opt(USE($\mathcal{A}$, $\mathcal{I}$) $\cup$ DEF($\mathcal{A}$, $\mathcal{I}$), DOM($\mathcal{L}$, $\mathcal{I}$))

L'opération *Sync_opt*($\mathcal{R}(\mathcal{I})$, $\mathcal{D}(\mathcal{I})$) effectuant la synchronisation de tous les processus possédant une des variables de $\mathcal{R}(\mathcal{I})$ pour une des valeurs de $\mathcal{D}(\mathcal{I})$. L'opération *Exec* est identique à celle utilisée pour la machine à messages.

Le distributeur identifie le cas où les variables écrites sont dupliquées sur l'ensemble des processus, cas où des copies explicites permettent une plus grande efficacité[2]. Il insère donc un appel à l'opération *Get* effectuant ces copies, la compilation de $\mathcal{L}$ produisant alors la séquence :

Sync_opt(USE($\mathcal{A}$, $\mathcal{I}$) $\cup$ DEF($\mathcal{A}$, $\mathcal{I}$), DOM($\mathcal{L}$, $\mathcal{I}$))
Get_opt(USE($\mathcal{A}$, $\mathcal{I}$), DEF($\mathcal{A}$, $\mathcal{I}$), DOM($\mathcal{L}$, $\mathcal{I}$))
Exec_opt(DEF($\mathcal{A}$, $\mathcal{I}$), DOM($\mathcal{L}$, $\mathcal{I}$), $\mathcal{A}$)
Sync_opt(USE($\mathcal{A}$, $\mathcal{I}$) $\cup$ DEF($\mathcal{A}$, $\mathcal{I}$), DOM($\mathcal{L}$, $\mathcal{I}$))

L'opération *Get_opt*($\mathcal{R}(\mathcal{I})$, $\mathcal{R}_P(\mathcal{I})$, $\mathcal{D}(\mathcal{I})$) rend accessible aux processus possédant une variable référencée dans $\mathcal{R}_P(\mathcal{I})$ une copie des valeurs des variables référencées dans $\mathcal{R}(\mathcal{I})$ pour toutes les valeurs des variables de $\mathcal{I}$ décrites par $\mathcal{D}(\mathcal{I})$. Contrairement au *Refresh*, aucune attente n'intervient dans cette opération.

2. Les raisons de ce choix seront explicitées au paragraphe 3.3.

3.1.3.3 Partie terminale du compilateur

La partie terminale du compilateur compile partiellement les opérations des machines abstraites. Ses deux principales fonctions, qui seront détaillées par la suite, sont :

- la transformation des accès aux données ;
- l'analyse des domaines et la génération de parcours restreints lors du traitement des opérations *Refresh_opt*, *Exec_opt*, et *Get_opt*. Cette partie du compilateur a été définie et mise en œuvre par Marc Le Fur [79]. Elle repose sur des techniques de manipulation de polyèdres.

Dans le code *C* généré durant cette dernière phase de la compilation, il est fait appel aux primitives de l'exécutif.

3.1.4 L'exécutif

L'exécutif permet l'exécution des opérations des machines abstraites. Il existe en deux versions distinctes : la première version cible une machine à messages et la deuxième, une machine à MVP. L'exécutif se compose d'un ensemble de primitives *C* structuré en deux niveaux, le premier niveau assure le lien avec le compilateur et le second constitue l'interface avec la machine d'exécution cible. Pour un type de machine d'exécution donné (machine à messages ou machine à MVP) on a à modifier uniquement cette deuxième couche de l'exécutif pour passer d'une plateforme parallèle à une autre.

La tâche de l'exécutif regroupe les points suivants :

Résolution des accès Dans le code généré par le compilateur, les éléments des tableaux distribués sont référencés sous la forme d'une adresse paginée[3]. L'exécutif doit donc réaliser la génération des adresses mémoire correspondantes.

Gestion des possesseurs L'application de la règle des écritures locales nécessite de calculer l'identité du processus possédant un élément ou un bloc d'éléments de tableau distribué. L'exécutif effectue ce calcul ainsi que le masquage d'instruction nécessaire dans les opérations *Exec*.

Transferts de données Les mouvements de données entre nœuds imposés durant l'opération *Refresh* sont effectués par l'exécutif. Outre les communications proprement dites, cela nécessite le codage et décodage d'ensembles d'éléments ainsi que l'application d'optimisations telles que la vectorisation ou l'agrégation.

3. La pagination des tableaux sera décrite au paragraphe 3.2.2.

Allocation mémoire Sur chaque processus, l'exécutif gère l'allocation et la libération de la mémoire où sont stockées les données locales, les données reçues et les différentes tables nécessaires à la résolution des accès.

Coopération hôte–nœuds L'exécutif est en charge du déclenchement des phases distribuées depuis l'hôte, des mouvements de données lors du passage de paramètres dans les phases distribuées.

3.2 Mise en œuvre sur machine à messages

Nous décrivons dans cette section comment a été mis en œuvre la machine abstraite PANDORE sur un ensemble de machines à messages. Nous décrivons d'abord une machine à messages générale, *la P.O.M.* qui recouvre plusieurs plates-formes parallèles puis nous développons la mise en œuvre des fonctionnalités de la machine abstraite (représentation et accès aux données, opérations *Refresh* et *Exec*) dans le compilateur et dans l'exécutif.

Une des solutions pour mettre en œuvre la machine abstraite PANDORE est de faire en sorte que les primitives de l'exécutif appellent directement les fonctions du système d'exploitation de l'architecture cible. Cette manière de procéder souffre évidemment d'un manque de portabilité. C'est pourquoi nous avions défini, dans une version antérieure de l'environnement PANDORE, un ensemble de primitives qui constituaient un dénominateur commun aux sous-ensembles utiles des routines de quelques systèmes cibles envisagés (NX/2 [91], Picl [50] sur iPSC/2).

Plusieurs projets de l'équipe Pampa utilisaient ou voulaient utiliser un ensemble voisin de routines dont certaines avaient déjà fait l'objet de développements sur diverses plate-formes parallèles, notamment à des fins d'observation d'exécution distribuées. Ceci a conduit à la définition d'une machine à messages plus générale, la POM, que nous avons adoptée comme machine d'exécution cible pour la machine abstraite PANDORE.

3.2.1 La POM

La POM (Parallel Observable Machine) [54, 55, 56] définit un modèle de machine à messages et se présente sous la forme d'une bibliothèque portable fournissant des services de communication et d'observation. Elle a été implantée sur plusieurs architectures et systèmes (NX/2 sur Intel iPSC, NX/OSF1 et Sunmos sur Intel Paragon XP/S [66], TCP/IP sur réseaux de stations Unix, simulateur sur station Unix) ainsi qu'au-dessus de PVM.

La POM n'est pas destinée principalement à un utilisateur final, le but est de fournir un ensemble minimal de primitives efficaces destinées à être utilisées dans un

code généré automatiquement ou à l'intérieur d'autres bibliothèques parallèles. Cet objectif différencie la POM d'autres bibliothèques de communication plus répandues comme PVM [18] ou MPI [89] dont le spectre d'utilisation visé est plus large et dont le manque de spécialisation peut nuire à leur efficacité ou à la facilité de leur portage.

3.2.1.1 Modèle de machine

Le modèle de machine défini par la POM comprend un nombre quelconque de *nœuds d'application* numérotés de 0 à $N-1$. Elle peut inclure en outre un noeud supplémentaire, baptisé *nœud observateur*. Ces nœuds possèdent une mémoire privée et communiquent via deux média distincts :

- Le premier médium est un réseau complètement maillé dédié aux communications point à point. Les canaux de ce réseau sont FIFO et fiables (il n'y a ni perte ni déséquencement de messages). Un nœud d'application peut émettre sur un canal sortant, recevoir sur un canal entrant et tester les files de réception.
- Le deuxième médium permet la diffusion de messages. Il s'agit également d'un réseau complètement maillé de canaux fiables et FIFO. Avec ce médium, un nœud peut effectuer une diffusion (émission sur tous ses canaux sortant), recevoir sur un canal entrant et tester les files de réception.

La distinction entre les deux réseaux est nécessaire car, sur bon nombre de plates-formes parallèles, il est difficile de garantir à faible coût le caractère FIFO de canaux virtuels de communication où circulent à la fois des messages diffusés et des messages émis en point-à-point. En effet, les communications en mode point-à-point et en diffusion font appel à des protocoles différents, voire à des dispositifs matériels distincts et généralement, le système d'exploitation associé n'assure pas d'ordre entre les messages de différentes natures.

S'il existe un nœud observateur, on doit alors considérer un troisième médium : un réseau comportant un ensemble de canaux reliant chaque nœud d'application à l'observateur. Avec ce médium d'observation, un nœud d'application peut émettre sur le canal sortant. Le nœud observateur peut quant à lui recevoir sur un canal entrant et tester les files de réception.

3.2.1.2 Interface

Les primitives utilisables par les nœuds d'application de la POM (préfixés par **APS_**) sont essentiellement des primitives de communication :

Émission : **APS_send** et **APS_bcast** permettent d'envoyer et de diffuser un message. L'émission est non bloquante, c'est-à-dire que la primitive retourne dès

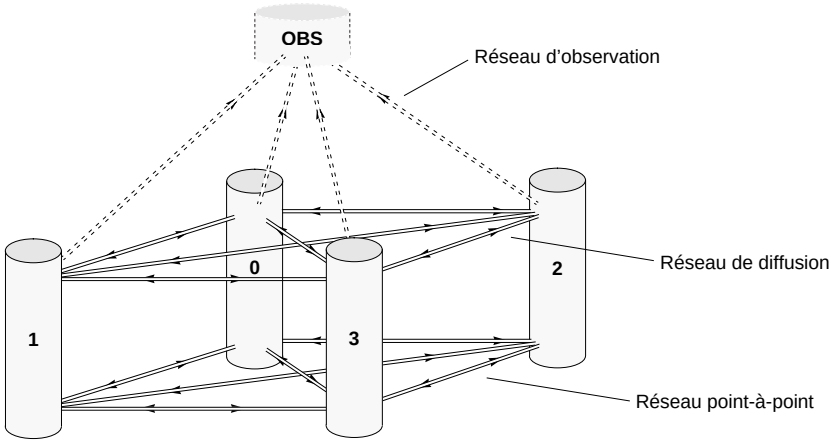


FIG. 3.4. – *Modèle de la machine virtuelle POM*

que le message est pris en compte par le système. Le tampon d'émission est alors réutilisable.

Réception : **APS_recv_from** et **APS_recv_bcast_from** permettent la réception sur un canal donné suivant que le médium utilisé est le médium point à point ou le médium de diffusion. La réception est bloquante, les primitives retournent lorsque le message a été effectivement reçu.

Test de files : **APS_probe_from** et **APS_probe_bcast_from** permettent de savoir sans bloquer si un message est disponible dans une file de réception. Ces primitives sont utilisées conjointement avec les primitives **APS_info_length** et **APS_info_pid** qui informe sur la longueur et la provenance du message ainsi détecté. Ces primitives ne sont utilisées dans la machine abstraite PANDORE qu'à des fins d'instrumentation (voir chapitre 4).

Bien qu'elle ne soit pas nécessaire dans l'environnement PANDORE, la réception indéterministe (sans précision de canal) est possible dans la POM. Par ailleurs, la POM offre un certain nombre de primitives permettant la génération, la collecte et l'exploitation de traces d'exécutions distribuées. Ces primitives et leur utilisation seront abordées au chapitre 4.

3.2.2 Gestion des tableaux distribués

Une des abstractions fournie par la machine abstraite PANDORE est le nommage homogène des tableaux multidimensionnels. Ceci nécessite une transformation lors

de la mise en œuvre sur une machine à messages telle que la POM car celle-ci définit un espace mémoire privé sur chaque processus. Deux aspects sont à étudier :

- La représentation des tableaux, c'est-à-dire la façon dont les portions de tableaux distribués vont être rangées dans les mémoires locales des processus.
- Les accès aux éléments de tableaux, c'est-à-dire comment, à partir d'un accès multidimensionnel noté par un nom de tableau et un vecteur d'indices, on accède à l'élément mémoire correspondant.

Pour un processus p donné, on distingue deux catégories d'éléments dont on doit définir la représentation et le mode d'accès : les éléments locaux, attribués à p par la distribution et les éléments reçus depuis d'autres processeurs, copies temporaires d'éléments distants.

Nous présentons en détail dans cette section la gestion des tableaux distribués employée dans la mise en œuvre de la machine abstraite PANDORE sur la POM. Nous discutons ensuite cette mise en œuvre en fonction des critères exposés au chapitre précédent.

3.2.2.1 Principe

La gestion des tableaux distribués de la machine abstraite PANDORE suit le schéma d'adressage des systèmes classiques de pagination de la mémoire. Dans de tels systèmes, l'espace mémoire logique est découpé en groupes d'éléments contigus, les pages. Ces pages ont une taille fixe et prédéterminée. L'accès aux éléments repose sur l'utilisation de composants matériels qui séparent une adresse logique en deux parties : un numéro de page et un *offset* dans cette page. Le numéro de page est utilisé comme index dans la table des pages qui contient l'adresse de chaque page dans la mémoire physique. Cette adresse de base est combinée avec l'offset pour obtenir l'adresse physique de l'élément. Avec une taille de page S, on obtient le numéro de page PG et l'offset OF à partir d'une adresse logique α par $PG = \alpha \; div \; S$ et $OF = \alpha \; mod \; S$. Si l'espace d'adresses logique est plus grand que l'espace physique, des mécanismes de gestion de mémoire virtuelle sont ajoutés. Dans ce cas, les pages peuvent ne pas être toujours présentes en mémoire physique mais temporairement chargées depuis la mémoire secondaire.

En ce qui concerne la machine abstraite PANDORE, la gestion porte sur les variables (les tableaux distribués) et non pas la mémoire. L'objectif n'est donc pas de construire une mémoire virtuelle partagée. De plus, on reste ici au niveau logiciel plutôt que de s'appuyer sur un composant matériel. Par rapport à un système de mémoire virtuelle partagée, nous avons donc des différences fondamentales :

- La notion de défaut de page n'a pas de sens car toutes les données distantes ont été rapatriées explicitement avant d'être lues. Par ailleurs, il n'y a pas d'obligation d'effectuer les communications page par page.

- Un mécanisme d'accès différent peut être défini pour chaque tableau, en particulier la taille des pages peut être spécifique à un tableau donné.
- L'espace d'adresses original est multi-dimensionnel ; par conséquent on applique une linéarisation de cet espace avant le découpage en pages.

3.2.2.2 Pagination des tableaux distribués

On définit une représentation et un mécanisme d'accès pour chaque tableau. L'espace d'indice d'un tableau donné est linéarisé par une fonction $\mathcal{L}$. L'espace d'adresse linéaire obtenu est découpé en pages de taille fixe S. Un processus stocke uniquement les pages qui contiennent au moins un élément qu'il lui a été attribué par la distribution ou un élément reçu depuis un autre processus. Suivant la distribution du tableau, $\mathcal{L}$ et S, une page peut être possédée par un ou plusieurs processus.

L'accès aux éléments locaux et reçus se fait de la même façon. En effet, en ce qui concerne les accès, un processus se comporte comme si la totalité du tableau était directement visible. La différence entre les pages contenant des éléments locaux et les pages contenant uniquement des éléments reçus réside dans la manière dont celles-ci sont allouées et remplies et non pas dans la manière dont on y accède.

Un couple (PG,OF) est calculé à partir du vecteur d'indices initial $(i_0, \ldots, i_{n-1})$ grâce à la fonction de linéarisation $\mathcal{L}$ et la taille de page S :

$$PG = \mathcal{L}(i_0, \ldots, i_{n-1}) \; div \; S$$
$$OF = \mathcal{L}(i_0, \ldots, i_{n-1}) \; mod \; S$$

La table des pages TP est stockée sur chaque processus. Elle indique l'adresse de base de chaque page présente dans la mémoire locale. L'offset est ajouté à cette adresse de base pour obtenir l'emplacement exact de l'élément.

Le découpage en page est également utilisé pour le calcul des possesseurs d'éléments. Une table similaire à TP stocke, pour chaque page, les numéros des processus qui possèdent la page. Cette table est présente dans la mémoire locale de chaque processus.

3.2.2.3 Réglage des paramètres

Pour un tableau donné, les paramètres de la pagination qui peuvent être réglés sont la taille de page S et la fonction de linéarisation $\mathcal{L}$. La valeur de ces paramètres doit être définie de façon à obtenir des performances d'accès maximales et limiter l'encombrement mémoire de la représentation.

Comme la rapidité d'accès est le critère principal, les opérations coûteuses en temps (division, modulo, multiplication) sont à éviter dans le calcul du couple

(PG,OF) mais aussi dans l'évaluation de la fonction $\mathcal{L}$. Pour ce faire, on introduit des puissances de deux, ce qui transforme les divisions entières, modulos et multiplications en de simples opérations logiques. De plus, la spécification de la décomposition du tableau est prise en compte. Intuitivement, on choisit S et $\mathcal{L}$ de telle sorte que les pages s'adaptent à la fois à la forme des blocs (les pages sont généralement dans le sens de la plus grande dimension des blocs) et à la taille des blocs (la taille des pages est proche de la taille des blocs dans cette dimension). On veille également à ce que les pages soient possédées par aussi peu de processus que possible.

Pour une définition plus formelle, considérons la distribution de tableau PANDORE suivante sur P processus.

$$int\ V[h_0]\cdots[h_{n-1}]\ by\ block\ (s_0,\ldots,s_{n-1})\ map \left|\begin{array}{l} regular \\ wrapped \end{array}\right| (d_0,\ldots,d_{n-1})$$

n	nombre de dimensions du tableau distribué	$n > 0$
h_k	taille du tableau dans la $k^{ème}$ dimension	$h_k > 0$
s_k	$k^{ème}$ paramètre de la fonction de décomposition	$1 \leq s_k \leq h_k$
d_k	$k^{ème}$ paramètre de la fonction de placement	$(d_k)_0^{n-1} = permut(0,..,n-1)$

On considère l'accès à un élément de V noté $V[i_0]\cdots[i_{n-1}]$. Pour introduire des puissances de deux, on définit la fonction $\theta_{sup}(n)$ (resp. $\theta_{inf}(n)$) pour étendre un entier à la puissance supérieure (resp. inférieure) :

$$\theta_{sup}(n) = 2^\rho \text{ avec } 2^\rho \leq n < 2^{\rho+1}$$
$$\theta_{inf}(n) = 2^\rho \text{ avec } 2^{\rho-1} < n \leq 2^\rho$$

Avant de définir S et $\mathcal{L}$, on choisit une dimension particulière, δ, la dimension suivant laquelle la taille des blocs est la plus grande. Si plusieurs dimensions vérifient cette propriété, on choisit une dimension non distribuée ou une dimension avec une taille de bloc égale à une puissance de deux s'il en existe. S est alors donnée par :

si $s_\delta = h_\delta$ ou $s_\delta = 2^\rho$
alors $S = \theta_{sup}(s_\delta)$
sinon $S = \theta_{inf}(s_\delta)$

$\mathcal{L}$ est la fonction de linéarisation de C pour les tableaux multi-dimensionnels appliquée à une permutation du vecteur d'indices. Cette permutation place l'index

correspondant à la dimension δ en dernière position. De plus, les dimensions du tableau (coefficients de $\mathcal{L}$) sont étendues aux puissances de deux supérieures :

$$\mathcal{L}(i_0, \ldots, i_{n-1}) \;=\; \sum_{k=0}^{n-1} \left(i'_k \prod_{l=k+1}^{n-1} h'_l \right)$$

où i'_k est le $k^{ème}$ indice d'accès après permutation, *i.e.* :

$$\begin{array}{l} i'_{n-1} = i_\delta \\ \forall k \in 0, \ldots, \delta{-}1 \;\; i'_k = i_k \\ \forall k \in \delta, \ldots, n{-}2 \;\; i'_k = i_{k+1} \end{array}$$

et h'_k est la taille étendue du tableau dans la $k^{ème}$ dimension, *i.e.* :

$$\begin{array}{l} h'_{n-1} = \theta \left(\left\lceil \dfrac{h_\delta}{S} \right\rceil \right) \times S \\ \text{si} \;\; n > 1 \\ \quad h'_0 = \; h_0 \;\; \text{si} \;\; \delta > 0, \;\; \text{sinon} \;\; h_1 \\ \quad \forall k \in 1, \ldots, \delta{-}1 \;\; h'_k = \theta(h_k) \\ \quad \forall k \in \delta, \ldots, n{-}2 \;\; h'_k = \theta(h_{k+1}) \end{array}$$

La figure 3.5 illustre le découpage en pages pour deux exemples 2D : dans le cas où une dimension n'est pas distribuée (tableau A) et dans celui où toutes les dimensions sont distribuées (tableau B).

3.2.2.4 Optimisations

Contrairement aux systèmes de pagination classique, le calcul effectif de l'adresse linéaire $\mathcal{L}(i_0, \ldots, i_{n-1})$ avant son découpage en (PG, OF) n'est pas obligatoire puisque ce découpage n'est pas opéré par un dispositif matériel qui nécessite la donnée d'une adresse mémoire. Par ailleurs, ce calcul intermédiaire peut entraîner des opérations inutiles comme dans l'exemple suivant.

$$\begin{array}{l} A[100][200] \;\; \text{by block}(10, 200) \\ \quad S \;=\; 256 \\ \quad \mathcal{L}(i, j) \;=\; 256\, i + j \end{array}$$

Le numéro de page et l'offset sont obtenus par

$$\begin{array}{l} PG = (256\, i + j) \; div \; 256 \\ OF = (256\, i + j) \; mod \; 256 \end{array}$$

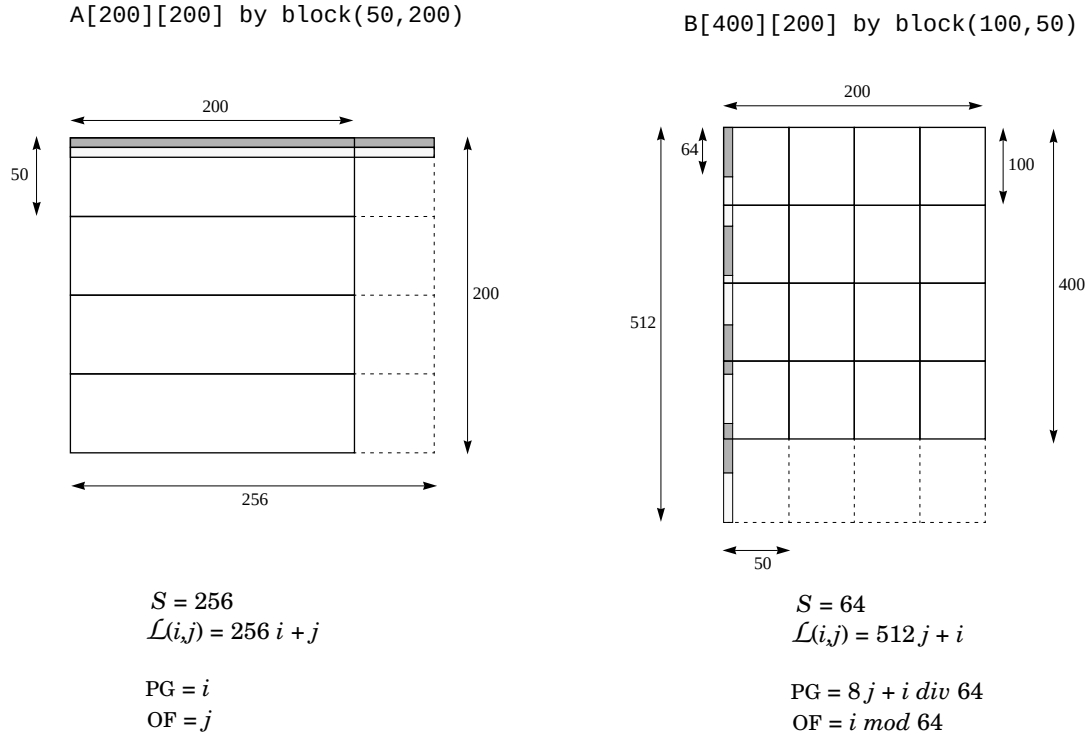


FIG. 3.5. – *Découpage en pages de tableaux 2D*

Ces expressions pourraient être de façon évidente simplifiée en $PG = i$ et $OF = j$. Pour rendre ces simplifications clairement visibles, on exprime directement PG et OF) comme une fonction du vecteur d'indices :

$$page(i_0, \ldots, i_{n-1}) \;=\; (PG, OF)$$

avec

$$PG \;=\; \sum_{k=0}^{n-2} \left(i'_k \prod_{l=k+1}^{n-1} np'_l \right) \;+\; i'_{n-1} \; div \; S$$
$$OF \;= i'_{n-1} \; mod \; S$$

où np'_k est le nombre de pages dans la $k^{ème}$ dimension après permutation :

$$np'_{n-1} = \frac{h'_{n-1}}{S}$$
$$\forall k \in 0, \ldots, n-2 \;\; np'_k = h'_k$$

Quand la dimension δ n'est pas distribuée, c'est-à-dire quand $h_\delta = s_\delta$, l'indice i'_{n-1} (*i.e.* i_δ) est toujours inférieur ou égal à S, on peut alors supprimer le *div* et le *mod*.

$$\begin{aligned} PG &= \sum_{k=0}^{n-2} \left(i'_k \prod_{l=k+1}^{n-1} np'_l \right) \\ OF &= i'_{n-1} \end{aligned}$$

Les deux exemples suivants montrent le résultat des optimisations (voir aussi la figure 3.5) :

$A[200][100][50]$ by block$(5, 100, 10)$

$$\begin{aligned} PG &= (8192i + 128k + j)\ div\ 128 = 64i + k \\ OF &= (8192i + 128k + j)\ mod\ 128 = j \end{aligned}$$

$B[500][200]$ by block$(100, 10)$

$$\begin{aligned} PG &= (512j + i)\ div\ 64 = 8j + (i\ div\ 64) \\ OF &= (512j + i)\ mod\ 64 = i\ mod\ 64 \end{aligned}$$

3.2.2.5 Calcul des possesseurs

Chaque processus stocke la table des possesseurs TO qui indique, pour chaque page, le numéro du processus qui possède la page. Cette table peut être remplie en utilisant la fonction *owner*(PG,OF) qui retourne le possesseur d'un élément.

$$owner(PG, OF) = map \circ page^{-1}(PG, OF)$$

La fonction $page^{-1}$, inverse de la fonction *page*, retourne le vecteur d'indices correspondant à un numéro de page et à un offset :

$$page^{-1}(PG, OF) = (i_0, \ldots, i_{n-1})$$

avec

$$\begin{aligned} & i_\delta = S \times \left(PG\ mod\ np'_{n-1} \right) + OF \\ & \forall k \in 0, \ldots, \delta-1 \quad i_k = i'_k \\ & \forall k \in \delta+1, \ldots, n-1 \quad i_k = i'_{k-1} \\ & \forall k \in 0, \ldots, n-2 \quad i'_k = \left(PG\ mod\ \left(\prod_{l=k}^{n-1} np'_l \right)\ div\ \left(\prod_{l=k+1}^{n-1} np'_l \right) \right) \end{aligned}$$

La fonction *map* associe un numéro de processus à un vecteur d'indices ; son contenu dépend de la fonction de placement utilisée dans la spécification de distribution :

- placement *regular* :
  $map(i_0, ..., i_{n-1}) \;=\; \left(\sum_{k=0}^{n-1} \left((i_k \; div \; s_k) \prod_{d_l<d_k} nb_l\right)\right) \; div \; \left\lceil \frac{\prod_{j=0}^{n-1} nb_j}{P} \right\rceil$
- placement *wrapped* :
  $map(i_0, ..., i_{n-1}) \;=\; \left(\sum_{k=0}^{n-1} \left((i_k \; div \; s_k) \prod_{d_l<d_k} nb_l\right)\right) \; mod \; P$

où nb_k est le nombre de blocs dans la $k^{ème}$ dimension : $nb_k = \left\lceil \frac{h_k}{s_k} \right\rceil$

La définition de S et $\mathcal{L}$ assure que le nombre de possesseurs d'une page est inférieur ou égal à deux. Si le possesseur d'une page est toujours unique, n'importe quelle valeur valide de OF peut être utilisée pour déterminer le possesseur d'une page. Dans le cas où le possesseur d'une page n'est pas toujours unique, on peut calculer OF_{lm}, l'offset à partir duquel le possesseur change. La table des possesseurs stocke alors, pour chaque page, les deux possesseurs et la limite OF_{lm}.

$$\forall OF \in 0, \ldots, OF_{lm} - 1 \quad owner(PG, OF) = owner(PG, 0)$$
$$\forall OF \in OF_{lm}, \ldots, S - 1 \quad owner(PG, OF) = owner(PG, OF_{lm})$$

avec

$$OF_{lm} \;=\; \text{si } \varphi < S \text{ alors } \varphi \text{ sinon } 0$$
$$\varphi \;=\; ((PG \; mod \; np'_{n-1}) \times (s_\delta - S)) \; mod \; s_\delta$$

3.2.2.6 Mise en œuvre

Tables et pages

Toutes les informations nécessaires au remplissage de la table des possesseurs de pages et la table des offsets-limites sont connues à la compilation. Ces tables pourraient par conséquent être définies statiquement. Cependant, afin de ne pas trop rallonger la taille du code généré, le compilateur produit des fonctions qui allouent et remplissent ces tables à l'exécution, au début des phases distribuées.

Pour chaque tableau distribué V, une table des possesseurs `TO_V` est définie. Si certaines pages du tableau peuvent être possédées par deux processus, trois tables sont alors nécessaires : la table des possesseurs de la première partie des pages `TO1_V`, la table des possesseurs de la deuxième partie des pages `TO2_V` et la table contenant les offset-limites `TL_V`.

L'exécutif est en charge d'allouer et de remplir les tables des pages et les pages elles-mêmes. Les tables des pages et les pages qui forment la partition locale sont

allouées au début de la phase distribuée. Afin d'augmenter le nombre d'éléments locaux contigus, la partition locale est allouée en un seul bloc mémoire. Les éléments de la partition locale sont éventuellement reçus du processus hôte au début de la phase distribuée et rapatriés à la fin de la phase. La gestion des pages contenant des éléments reçus dépend du schéma de compilation, elle est détaillée dans la section 3.2.3.

Accès

On fait en sorte que la majeure partie du calcul de l'adresse mémoire correspondant à un élément de tableau distribué soit fait à la compilation. Le compilateur transforme une référence à un élément de tableau $V[I]$, où I est un vecteur d'indices en un appel à une primitive de l'exécutif `access(desc_V, PG, OF)` où `PG` et `OF` sont des expression sur I. Toutes les sous-expressions constantes ont été calculées et l'on a effectué les optimisations décrites dans la section précédente de telle sorte que les expressions obtenues ne contiennent que des additions, des décalages et des masques de constantes. Hormis l'évaluation de I, le travail restant à faire à l'exécution est donc l'évaluation de ces additions et opérations logiques et le passage par la table des pages associée à V.

Calcul des possesseurs

La détermination du possesseur d'un élément $V[I]$ est effectuée de la même façon que l'accès. Le compilateur génère un appel à une macro de l'exécutif `owner(desc_V, PG, OF)`. Un accès à la table `TO_V[PG]` est suffisant à l'exécution pour trouver le numéro de processus dans le cas où le possesseur d'une page est toujours unique. Si une page peut être possédée par deux processus, un test est nécessaire pour savoir de quel coté de l'offset-limite se trouve l'élément.

3.2.2.7 Performances

On a vu que le code généré pour l'accès aux éléments de tableaux distribués ne fait intervenir que quelques opérations peu coûteuses qui occasionnent un surcoût faible par rapport à un accès séquentiel. Le fait de faire apparaître des puissances de deux dans la fonction de linéarisation peut même conduire à un temps d'accès distribué inférieur au temps d'accès séquentiel.

Afin d'effectuer une évaluation plus précise, nous avons comparé, dans le cadre d'une affectation d'un scalaire, différents temps d'accès en lecture, dans le cas ou la partie droite de l'affectation est :

- t_s : une référence à un élément de tableau telle qu'elle peut apparaître dans un programme séquentiel ;
- t_p : un appel à une macro qui utilise le mécanisme d'accès par pages ;

	Sparcstation		iPSC/2		Paragon	
	favor.	*defav.*	*favor.*	*defav.*	*favor.*	*defav.*
t_s	0.30	0.42	0.94	2.05	0.16	0.26
t_p	0.34	0.38	2.14	2.26	0.22	0.25
t_b	0.48	1.58	3.52	9.86	0.21	2.68

TAB. 3.1. – *Comparaison des temps d'accès (en µs) sur trois plates-formes*

- t_b : un appel à une macro mettant en œuvre un mécanisme d'accès fondé sur le calcul de blocs[4].

Le tableau 3.1 montre les résultats de l'expérience, les temps sont indiqués en microsecondes. Le tableau est un tableau bi-dimensionnel de réels en simple précision. Les cas favorables et défavorables ont été considérés selon que les tailles du tableau étaient des puissances de deux ou non. L'expérience a été effectuée sur une SparcStation 2, sur un nœud de l'Intel iPSC/2 et sur un nœud de l'Intel Paragon XP/S. Les optimisations des compilateurs ont été inhibées pour éviter toute perturbation due au fait que la mesure se fait sur une boucle d'accès.

De même, la détermination du possesseur d'un élément de tableau requiert seulement quelques opérations simples. Son coût en temps demeure donc très faible. Comme pour l'accès aux éléments de tableaux, il est préférable d'exploiter la décomposition en pages bien qu'il semble plus naturel de baser le calcul de possesseur sur le calcul du bloc.

La rapidité d'accès aux éléments et de calcul de possesseur a un impact important sur les performances globales des programmes générés, comme en témoignent les résultats présentés dans le tableau 3.2. Les temps d'exécution (en secondes) de différentes versions d'un programme de relaxation itérative[5] opérant sur un tableau de 1024×1024 réels en simple précision et exécuté sur un iPSC/2 sont donnés. La comparaison est faite, pour différents nombres de processeurs, entre l'utilisation de la gestion des tableaux par pages et celle de la gestion fondée sur les calculs de blocs mentionnée plus haut, ceci pour deux schémas de compilation : un schéma de base et un schéma optimisé (ces deux schémas seront détaillés dans la suite de ce chapitre). On voit que le gain apporté par la gestion de tableau par page est important dans les deux cas et particulièrement pour le schéma optimisé.

Le prix à payer pour la rapidité d'accès aux éléments et la rapidité de calcul des possesseurs est un besoin accru en mémoire. Le surcoût mémoire est seulement dû aux tables. En effet, lorqu'une page contient des éléments qui n'ont pas d'équivalent

4. Ce mécanisme d'accès était utilisé dans une version antérieure de PANDORE [37], il consiste à calculer le numéro du bloc et l'index dans le bloc linéarisé, le calcul faisant intervenir un modulo et une division entière.

5. Il s'agit du programme Red-Black SOR décrit page 105.

P	*Schéma de base*		*Schéma optimisé*	
	Bloc	*Page*	*Bloc*	*Page*
4	92.9	39.6	31.4	7.0
8	84.2	35.0	19.4	3.7
16	74.9	32.9	12.6	2.1
32	72.4	31.7	7.3	1.14

TAB. 3.2. – *Comparaison des temps d'exécution (en secondes) entre la gestion par page et la gestion fondée sur le calcul de numéro de bloc pour deux schémas de compilation*

Spécification de distribution	*Nombre de pages*	*Partition minimale (en octets)*	*Tables des pages (en octets)*	*Surcoût local*
double A[100000] by block(1000)	196	25000	1960	×1.08
double A[100000] by block(1024)	98	25000	588	×1.02
double A[1000][1000] by block(1,1000)	1000	250000	6000	×1.02
double A[1000][2000] by block(50,500)	8000	500000	80000	×1.16
double A[1000][2000] by block(50,512)	4000	500000	24000	×1.05
double A[100][100][100] by block(100,1,50)	10000	250000	60000	×1.24

TAB. 3.3. – *Coût mémoire pour quelques distributions communes*

dans le tableau original ou quand seulement une partie d'une page distante est accédée dans une boucle optimisée, on alloue seulement une portion de la page.

Le tableau 3.3 donne les volumes en mémoire nécessaires pour quelques distributions courantes sur 32 processus. Pour chaque distribution, on indique le nombre total de pages, l'espace mémoire minimum théorique requis sur chaque processeur, l'espace effectif alloué sur chaque processus pour les tables et finalement le surcoût par rapport à la partition minimale. Les volumes mémoire sont exprimés en octets. On peut noter que le fait de remplacer certaines tailles de bloc (ou de tableau) par des puissances de deux fait décroître sensiblement le surcoût.

3.2.2.8 Discussion

La gestion des tableaux par pages satisfait les critères que nous avions développés au chapitre 2.

- Elle réalise un compromis acceptable entre le temps d'accès aux éléments et l'occupation mémoire induite par la représentation. On peut noter que contrairement à plusieurs projets [97, 103, 36, 25], l'effort a porté sur l'accélération des accès, qui nous semble le critère primordial, l'utilisation de la mémoire n'étant pas toujours optimale. On peut remarquer à cet égard que l'efficacité en mémoire de la gestion des tableaux par pages est liée à l'hypothèse que les blocs résultant de la distribution de tableaux sont relativement gros. Des blocs de la taille d'un seul élément entraîneraient une consommation mémoire plus importante que celle induite par la duplication de l'espace d'adressage complet. On peut cependant s'interroger sur le caractère indispensable de telles décompositions que l'on peut considérer comme une absence de décomposition.

- Elle constitue une gestion uniforme des éléments locaux et reçus, facilitant le travail du compilateur qui n'a pas à séparer les calculs purement locaux de ceux nécessitant des données distantes. La tâche du compilateur est en outre facilitée par le fait que le passage de l'espace d'adresses globales à l'espace d'adresses locales ne se fait qu'à l'exécution.

- Elle est définie uniquement à partir de la déclaration du tableau et de sa décomposition en bloc, elle est de ce fait indépendante de toute analyse des instructions du programme. Comme elle est de surcroît indépendante de l'attribution des blocs aux processus, son utilisation dans des compilateurs d'autres langages « data-parallèles » est envisageable dans la mesure où la distribution des tableaux définit toujours une décomposition en blocs.

- La contiguïté est préservée dans une certaine mesure. En effet, les éléments d'une page, qui sont par définition contigus, le sont aussi — au sens de la contiguïté dans un espace multi-dimensionnel — dans le tableau d'origine. La contiguïté mémoire est conservée dans le cas où la direction des pages correspond à la dernière dimension du tableau. De plus, toutes les pages d'une partition locale sont allouées au sein d'un même bloc de mémoire, ce qui tend à renforcer la contiguïté.

3.2.3 Opérations Refresh / Exec

La mise en œuvre des opérations *Refresh* et *Exec* de la machine abstraite PANDORE sont effectuées en deux temps : le compilateur expanse ces opérations en opérations élémentaires qui sont représentées par des appels à des primitives de l'exécutif. C'est

dans la mise en œuvre de ces primitives que l'on utilise les routines de communication de la POM.

Nous décrivons ici le travail du compilateur et les détails de mise en œuvre des principales primitives de l'exécutif pour les deux schémas de compilation présentés au paragraphe 3.1.3.

3.2.3.1 Le schéma de base

Travail du compilateur

Le travail du compilateur consiste essentiellement à transformer le rafraîchissement de plusieurs variables en une série de rafraîchissements élémentaires. En outre, il se charge de l'allocation des temporaires scalaires où seront stockées les copies d'éléments distants. Par exemple, si les variables X et Y sont de type `float`, la séquence d'opérations suivante, issue de l'application du schéma de compilation de base sur l'affectation $X[i] = X[i+1] + Y[i]$:

Refresh_base$(\{X[i+1], Y[i]\}, \{X[i]\})$
Exec_base$(\{X[i]\}, X[i] = X[i+1] + Y[i])$

donnera lieu à la production du code

```
{
  float tmp1, tmp2;
  procid p;
  int pg, of;

  pg = PG_X(i);
  of = OF_X(i);
  p = owner(desc_X, pg, of);
  refresh_dist(tmp1, desc_X, PG_X(i+1), OF_X(i+1), p);
  refresh_dist(tmp2, desc_Y, PG_Y(i), OF_Y(i), p);
  exec_dist(p, desc_X, pg, of, tmp1 + tmp2);
}
```

On utilise le fait que l'opération *Exec_base* suit immédiatement le *Refresh_base*. On peut donc limiter la portée des temporaires *tmp1* et *tmp2*. L'allocation de ces temporaires utilise en outre le fait que le langage cible dispose de la structure de bloc, ils sont déclarés comme des variables locales au bloc créé.

Les éléments de tableaux sont identifiés par un descripteur ($desc_X$ ou $desc_Y$) et par un couple d'expressions ($PG()$ et $OF()$) calculées par le compilateur à partir du vecteur d'indice de la référence au tableau.

La distribution des tableaux n'autorise qu'un seul possesseur pour un élément de tableau donné. La référence à l'ensemble OWN($\{X[i]\}$) est donc traduite par un appel à la primitive `owner`() qui rend l'identificateur du possesseur d'un élément.

Le numéro du processus qui doit effectuer l'affectation est utilisé à la fois dans les primitives `refresh_dist`() et `exec_dist`(), il est donc précalculé. Il en est de même pour le numéro de page et l'offset identifiant l'élément à affecter ; ils sont donc stockés dans deux variables intermédiaires.

Travail de l'exécutif

Les tâches principales de l'exécutif sont de faire communiquer les processus lors du rafraîchissement élémentaire et de masquer l'affectation. La mise en œuvre des primitives `refresh_dist` et `exec_dist` est la suivante :

```
refresh_dist(tmp, desc, PG, OF, rhs_p) ≡
{
    procid lhs_p = owner(desc, PG, OF)
    si myself = rhs_p alors
        si lhs_p ≠ rhs_p
            alors send(access(desc, PG, OF), lhs_p)
            sinon tmp = access(desc, PG, OF)
        fsi
    fsi
    si myself = lhs_p alors
        si lhs_p ≠ rhs_p
            alors recv(tmp, rhs_p)
        fsi
    fsi
}
```

```
exec_dist(rhs_p, desc, PG, OF, expression) ≡
{
    si myself = rhs_p alors
        access(desc, PG, OF) = expression
    fsi
}
```

Cas des variables dupliquées

Lorsque la variable à affecter est une variable dupliquée, scalaire ou élément de tableau, une variante du code présenté plus haut est générée. Il est simplifié du fait

que l'ensemble des processus sur lesquels on doit rafraîchir les données et qui doivent effectuer l'affectation est toujours égal à la totalité des processus. Par exemple, la séquence d'opérations issue de l'affectation de scalaire flottant $a = X[i+1] + Y[i]$:

Refresh_base$(\{X[i+1], Y[i]\}, \{a\})$
Exec_base$(\{a\}, X[i] = X[i+1] + Y[i])$

donnera lieu à la production du code

```
{
  float tmp1, tmp2;

  refresh_rep(tmp1, desc_X, PG_X(i+1), OF_X(i+1));
  refresh_rep(tmp2, desc_Y, PG_Y(i), OF_Y(i));
  exec_rep(a, tmp1 + tmp2);
}
```

Les références à des variables dupliquées sont représentées par le texte original (par exemple a ou $A[i]$). La référence à l'ensemble des possesseurs de $\{a\}$ est rendue implicite.

Le travail de l'exécutif, dans la primitive `refresh_rep`, consiste à faire en sorte que le possesseur de la variable à rafraîchir diffuse sa valeur aux autres processus :

```
refresh_rep(tmp, desc, PG, OF) ≡
{
    procid lhs_p = owner(desc, PG, OF)
    si myself = lhs_p
        alors
            tmp = access(desc, PG, OF)
            bcast(tmp)
        sinon
            recv_bcast(tmp, lhs_p)
    fsi
}
```

La primitive `exec_rep` réalise l'affectation, sans masquage puisque tous les processus doivent l'effectuer. Elle n'est en fait présente que pour des raisons d'homogénéité.

```
exec_rep(tmp, expression) ≡
{
    tmp = expression
}
```

3.2.3.2 Le schéma optimisé

La tâche du compilateur et celle de l'exécutif sont beaucoup plus complexes que dans le cas du schéma de base. Lors de l'opération *Refresh*, les communications concernant une référence ne portent plus sur un élément de tableau mais sur un ensemble d'éléments. De plus, alors que dans le schéma de base, au plus deux processus pouvaient être impliqués dans les communications sous-jacentes au rafraîchissement d'une référence, tous les processus participent potentiellement au rafraîchissement de l'ensemble d'éléments lié à chaque référence.

Pour illustrer le travail du compilateur, nous considérerons dans les paragraphes qui suivent, l'exemple de la compilation du nid de boucles

pour i **de** 1 **à** $N-2$
  pour j **de** i **à** $N-2$
    $X[i][j] \ = \ Z[i][j] + Y[i-1][j] \ + \ Y[i+1][j]$

qui est traduit en

$Refresh\_opt(\mathcal{R}(\mathcal{I}),\ \mathcal{R}_P(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$
$Exec\_opt(\mathcal{R}_P(\mathcal{I}),\ \mathcal{A}(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$

où

$\mathcal{I} \ = \ \{i, j\}$
$\mathcal{D}(\mathcal{I}) \ = \ \{1 \leq i \leq N-2,\ i \leq j \leq N-2\}$
$\mathcal{R}(\mathcal{I}) \ = \ \{Y[i-1][j], Y[i+1][j], Z[i][j]\}$
$\mathcal{R}_P(\mathcal{I}) \ = \ \{X[i][j]\}$
$\mathcal{A}(\mathcal{I}) \ = \ X[i][j] \ = \ Z[i][j] + Y[i-1][j] \ + \ Y[i+1][j]$

Les tableaux X et Y sont partitionnés en P groupes de lignes et le tableau Z est partitionné en P groupes de colonnes comme illustré sur la figure 3.6 ($N = 512$).

L'opération Refresh

Principe

La mise en œuvre de l'opération *Refresh_opt* est répartie entre le compilateur et l'exécutif. Pour chaque référence de $\mathcal{R}(\mathcal{I})$, le compilateur constitue un système de contraintes affines en utilisant également la référence apparaissant dans $\mathcal{R}_P(\mathcal{I})$ et le domaine $\mathcal{D}(\mathcal{I})$. Ce système caractérise les éléments à communiquer, il définit en fait un polyèdre dont les points peuvent être énumérés par un nid de boucles.

```
int X[N][N] by block(N/P,N) map wrapped(0,1) mode OUT
int Y[N][N] by block(N/P,N) map wrapped(0,1) mode IN
int Z[N][N] by block(N,N/P) map wrapped(0,1) mode IN
```

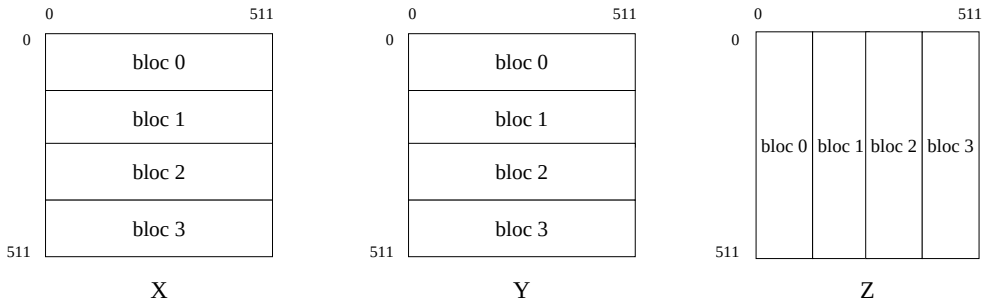


FIG. 3.6. – *Partitionnement des tableaux X, Y et Z*

Sur l'exemple, pour la référence $Z[i][j]$, on constitue le système de contraintes suivant

$$\begin{cases} 0 \leq kX \leq 3 \\ 0 \leq kZ \leq 3 \\ 1 \leq i \leq 510 \\ i \leq j \leq 510 \\ 128\,kX \leq i \leq 128\,kX\ + 127 \\ 128\,kZ \leq j \leq 128\,kZ\ + 127 \end{cases}$$

qui définit l'ensemble des points (kX, kZ, i, j) dans lesquels le vecteur d'itération (i, j) est tel que la référence de la partie gauche de l'affectation $X[i][j]$ écrit dans le bloc kX de X et la référence de la partie droite $Z[i][j]$ lit dans le bloc kZ de Z. Ce polyèdre est décrit par le nid de boucles suivant :

pour kX **de** 0 **à** 3
  pour kZ **de** kX **à** 3
    pour i **de max**$(128\,kX,\ 1)$ **à min**$(128\,kX + 127,\ 510)$
      pour j **de max**$(i,\ 128\,kZ)$ **à min**$(128\,kZ + 127,\ 510)$

En ajoutant des gardes sur la possession des blocs kX et kZ dans ce parcours, on peut produire un code SPMD d'envoi et de réception :

- code d'envoi

  pour kX **de** 0 **à** 3
    si `myself` $\neq$ possesseur du bloc kX de X **alors**

pour kZ **de** kX **à** 3
si `myself` = possesseur du bloc kZ de Z **alors**
pour i **de max**(128 kX, 1) **à min**(128 kX + 127, 510)
pour j **de max**(i, 128 kZ) **à min**(128 kZ + 127, 510)
envoyer $Z[i][j]$ au possesseur du bloc kX de X

- code de réception

pour kX **de** 0 **à** 3
si `myself` = possesseur du bloc kX de X **alors**
pour kZ **de** kX **à** 3
si `myself` $\neq$ possesseur du bloc kZ de Z **alors**
pour i **de max**(128 kX, 1) **à min**(128 kX + 127, 510)
pour j **de max**(i, 128 kZ) **à min**(128 kZ + 127, 510)
recevoir $Z[i][j]$ du possesseur du bloc kX de X

Le code de communication ainsi obtenu ne permet pas d'obtenir des performances satisfaisantes. Les communications se font élément par élément et certains envois peuvent être redondants. En effet, lorsque par exemple deux références en partie droite de l'affectation sont des références à une seule variable (cas de $Y[i-1][j]$ et $Y[i+1][j]$ dans l'exemple), certains éléments vont être communiqués plusieurs fois au même processeur sur l'ensemble du domaine. Des envois peuvent être également redondants lorsque les fonctions d'accès des références de la partie droite sont non injectives.

L'obtention d'une efficacité correcte repose sur la mise en œuvre de plusieurs optimisations dans l'exécutif :

- Les messages sont agrégés : les petits messages sont regroupés dans des messages plus gros afin de masquer l'effet de la latence.
- Des communications directes sont utilisées si nécessaire. Dans ce cas, ce sont des éléments stockés de façon contiguë à la fois chez l'émetteur et le récepteur qui sont transférés. Il n'y a donc pas ici d'empaquetage/dépaquetage ni de copie de mémoire entre les tampons d'émission/réception et les représentations locales.
- Les communications redondantes sont évitées : un élément est transféré une seule fois.

Structure du code généré

Afin de mettre en œuvre les optimisations citées plus haut l'opération *Refresh_opt* est décomposée en une séquence de codes de description de communication et de codes d'échange. Les codes de description serviront à remplir un certain nombre de structures de données décrivant ce qui doit être communiqué, les codes d'échanges effectueront les communications proprement dites. On a un code de description par référence de $\mathcal{R}(\mathcal{I})$ et un code d'échange par variable de $\mathcal{R}(\mathcal{I})$. Afin de diminuer

les synchronisations, chaque séquence de codes de description pour une variable donnée est immédiatement suivie du code d'échange de cette variable. Les différentes variables de $\mathcal{R}(\mathcal{I})$ sont traitées successivement. Dans l'exemple, on aura donc la séquence suivante :

Refresh_opt$(\mathcal{R}(\mathcal{I}),\ \mathcal{R}_P(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$ $\equiv$
{
code de description pour $Y[i-1][j]$
code de description pour $Y[i+1][j]$
code d'échange pour Y

code de description pour $Z[i][j]$
code d'échange pour Z
}

Code de description

Pour un processus p le code de description associé à une référence est destiné à :

- déterminer les processus auxquels p doit envoyer des éléments ;
- déterminer, pour chacun des ces destinataires, l'ensemble des éléments à envoyer ;
- déterminer les processus desquels p doit recevoir des éléments.

La description des ensembles d'éléments à recevoir sont transmis dans les messages, elle est donc calculée une seule fois par l'émetteur. Le code de description est directement dérivé du code d'envoi et de réception présenté plus haut. Pour la référence $Z[i][j]$ de l'exemple, on génère le code suivant :

pour kX **de** 0 **à** 3
  $pX =$ `owner_bloc`$(desc_X,\ kX)$
  si `myself` $\neq\ pX$ **alors**
    pour kZ **de** kX **à** 3
      si `myself` $=$ `owner_bloc`$(desc_Z,\ kZ)$ **alors**
        pour i **de max**$(128\,kX,\ 1)$ **à min**$(128\,kX + 127,\ 510)$
          pour j **de max**$(i,\ 128\,kZ)$ **à min**$(128\,kZ + 127,\ 510)$
            `elt_pack`$(desc_Z,\ j,\ i,\ pX)$
        `add_recver`$(desc_Z,\ pX)$

pour kX **de** 0 **à** 3
  si `myself` $=$ `owner_bloc`$(desc_X,\ kX)$ **alors**
    pour kZ **de** kX **à** 3
      $pZ =$ `owner_bloc`$(desc_Z,\ kZ)$
      si `myself` $\neq\ pZ$ **alors**
        `add_sender`$(desc_Z,\ pZ)$

Ce code fait appel à la primitive de l'exécutif `owner_bloc`() qui rend le numéro du processus possédant le bloc k de la variable V, et aux primitives `add_recver`(), `add_sender`() et `elt_pack`() qui permettent de construire respectivement, pour une variable V :

- la liste des processus destinataires D_V ;
- la liste des processus expéditeurs E_V ;
- la table (indexée par les numéros de processus) des listes de segments à expédier : S_V.

L'utilisation de la primitive `add_recver`() dans le code d'émission est indispensable bien qu'a priori, la liste D_V puisse être constituée uniquement grâce à la primitive `elt_pack`(). En effet, il se peut que, pour certains couples de blocs ((kX, kZ) dans l'exemple) atteints par les deux premières boucles et tests, il n'y ait pas d'élément à émettre (*i.e.* le nid de boucles (i,j) soit vide). Afin d'éviter notamment un interblocage, il faut quand même qu'un message (vide) soit émis vers le processeur du bloc kX car celui-ci attendra un message, conformément à la liste des émetteurs qu'il aura constituée dans le code de réception dual.

La notion de segment est ici utilisée pour réduire le volume de données nécessaire à la description des ensembles d'éléments à communiquer. Un segment est un ensemble contigus d'éléments appartenant à une page. Il est représenté par un triplet (*pg*, *ofs*, *ofe*) où *pg* est le numéro de la page, *ofs* est l'offset correspondant au début du segment et *ofe* l'offset de la fin du segment.

Afin de limiter la complexité de la primitive `elt_pack`, on ne mémorise qu'un seul segment par page. Les différentes listes ne sont pas réinitialisées à vide entre chaque code de description, on parvient ainsi à éviter d'enregistrer deux fois le même élément et donc d'éviter les envois redondants de données. Toutefois, le fait de coder les ensembles d'éléments par des segments et le fait de n'avoir qu'un seul segment par page peut entraîner l'enregistrement d'éléments superflus dans le cas où les éléments à envoyer ne sont pas contigus.

Code d'échange

Le code d'échange est entièrement inclus dans l'exécutif. Le code généré se limite à l'appel de la primitive `exchange`($desc_V$) qui se charge de l'ensemble des envois et réceptions de segments pour l'ensemble des destinataires et expéditeurs. Le code de la primitive `exchange` est donné dans la figure 3.7.

Les segments sont communiqués différemment suivant que leur taille est inférieure ou supérieure à un seuil. Les petits segments sont regroupés en un seul message, on doit donc effectuer une allocation mémoire supplémentaire et des copies à la fois chez l'émetteur et chez le récepteur. En revanche, on utilise un message par gros segment, ce qui permet d'utiliser des communications directes, le segment passant directement de la page chez l'émetteur à la page chez le récepteur. Le seuil

```
exchange(desc_V) ≡
{
    —— Envois des segments ——
    pourtout p ∈ D_V
        séparer S_V[p] en
          - la liste des gros segments GS_V
          - la liste des petits segments PS_V
        len_agg = longueur des contenus de PS_V
        envoyer (GS_V + len_agg) à p
        pourtout s ∈ GS_V
            envoyer contenu de s à p
        finpourtout
        si len_agg ≠ 0 alors
            allouer un buffer buf_agg de longueur len_agg
            pourtout s ∈ PS_V
                aggréger le contenu de s dans buf_agg
            finpourtout
            envoyer (PS_V + buf_agg) à p
            libérer buf_agg
        fsi
    finpourtout
    —— Réception des segments ——
    pourtout p ∈ E_V
        recevoir (GS_V + len_agg) de p
        pourtout s ∈ GS_V
            recevoir contenu de s de p
        finpourtout
        si len_agg ≠ 0 alors
            allouer un buffer buf_agg de longueur len_agg
            recevoir (PS_V + buf_agg) de p
            pourtout s ∈ PS_V
                copier le contenu de s de buf_agg dans la page correspondante
            finpourtout
            libérer buf_agg
        fsi
    finpourtout
}
```

FIG. 3.7. – *Code de la primitive* `exchange`

S séparant les petits des gros segments est déterminé à partir de paramètres spécifiques à l'architecture cible : la latence des messages l_c, la latence de la recopie mémoire l_m, le débit des messages d_c et le débit de la recopie mémoire d_m. On cal-

cule une approximation du seuil idéal en négligeant la latence du message contenant les petits segments agrégés. Si, pour un segment de longueur t, le coût de sa recopie mémoire est de $l_m + t\,d_m$ et le coût de sa transmission est de $l_c + t\,d_c$ alors on a :

$$S \;=\; \frac{l_c \;-\; 2\,l_m}{2\,d_m}$$

Une fois les deux listes constituées, on corrige le fait de négliger la latence du message contenant les petits segments agrégés en transférant les n segments de PS_V dans GS_V si nécessaire, c'est-à-dire si

$$n(2\,l_m - l_c) + t(2\,d_m + d_c) + l_c \;>\; 0$$

ce qui, en particulier, est toujours vrai lorsque $n = 1$.

On peut remarquer dans la figure 3.7 que le choix stipulant que les récepteurs reçoivent la description des ensembles d'éléments (listes de segments) qu'ils doivent recevoir plutôt que de les calculer comme les émetteurs entraîne éventuellement des communications supplémentaires, *i.e.* des transferts de messages ne contenant pas d'éléments de tableaux : dans le cas où seuls des petits segments sont à transmettre, un message préalable indiquant au récepteur la taille de l'agrégat est nécessaire pour allouer le tampon de réception.

Absence d'interblocage et contrôle de flux

La mise en œuvre proposée pour l'opération *Refresh_opt* est exempte d'interblocage (sous l'hypothèse de files FIFO infinies) :

- Par la dualité des parcours des polyèdres générés, le code de description assure la correspondance entre les ensembles de processus émetteurs et récepteurs. Le fait que la description des ensembles d'éléments soit placés dans les messages complète la garantie de correspondance entre les émissions et les réceptions.
- Dans la primitive `exchange`, toutes les émissions sont effectuées avant les réceptions, empêchant ainsi tout interblocage.

L'hypothèse FIFO infinie n'est pas garantie par la POM et il est donc possible qu'un interblocage intervienne dans le cas où toutes les émissions sont bloquées par la saturation des tampons. Le risque de saturation des files a été toutefois limité dans la mesure où la primitive `exchange` est invoquée pour chaque variable lue dans la boucle. Il s'agit d'une solution intermédiaire entre un échange pour chaque référence et un seul échange pour tout le *Refresh_opt*. Les expérimentations conduites jusqu'à présent ne justifiaient pas d'ajouter un contrôle de flux supplémentaire dans la primitive `exchange`. C'est pourquoi toutes les émissions y sont faites au plus tôt.

On pourrait cependant envisager de définir un ordonnancement plus asymétrique des émissions/réceptions (réception d'une partie des données avant d'avoir tout émis). Cet ordonnancement pourrait éventuellement tenir compte de la taille des tampons particulière à une plate-forme cible (cette taille pourrait être un paramètre fourni par la POM, au même titre que le débit des transferts par exemple).

L'intérêt d'une telle mise en œuvre serait toutefois à étayer par des expérimentations, étant donné les éventuelles attentes induites par la possibilité de réceptions antérieures aux émissions correspondantes.

L'opération Exec

Le travail du compilateur dans la mise en œuvre de l'opération *Exec_opt* suit les mêmes principes que pour l'opération *Refresh_opt*. A partir de la référence apparaissant dans $\mathcal{R}_P(\mathcal{I})$ et de $\mathcal{D}(\mathcal{I})$, le compilateur constitue un système de contraintes caractérisant les éléments à calculer. Sur l'exemple, l'opération à effectuer est

$$\mathit{Exec\_opt}(\mathcal{R}_P(\mathcal{I}),\ \mathcal{A}(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$$

avec

$$\begin{aligned}
\mathcal{I} &= \{i, j\} \\
\mathcal{D}(\mathcal{I}) &= \{1 \leq i \leq N-2,\ i \leq j \leq N-2\} \\
\mathcal{P}(\mathcal{I}) &= \mathrm{OWN}(X[i][j]) \\
\mathcal{A}(\mathcal{I}) &= X[i][j] = Z[i][j] + Y[i-1][j] + Y[i+1][j]
\end{aligned}$$

Le système de contraintes correspondants aux déclarations et distributions de la figure 3.6 est le suivant :

$$\begin{cases}
0 \leq kX \leq 3 \\
1 \leq i \leq 510 \\
i \leq j \leq 510 \\
128\,kX \leq i \leq 128\,kX + 127
\end{cases}$$

Il définit l'ensemble des points (kX, i, j) dans lesquels le vecteur d'itération (i, j) est tel que la référence $X[i][j]$ écrit dans le bloc kX de X. Ce polyèdre est parcouru par le nid de boucles

pour kX **de** 0 **à** 3
  pour i **de max**$(128\,kX,\ 1)$ **à min**$(128\,kX + 127,\ 510)$
    pour j **de** i **à** 510

L'ajout d'une garde sur le possesseur du bloc kX permet d'obtenir le code SPMD de calcul :

pour kX **de** 0 **à** 3
  si `myself` = possesseur du bloc kX de X **alors**
    pour i **de max**$(128\,kX,\ 1)$ **à min**$(128\,kX + 127,\ 510)$
      pour j **de** i **à** 510
        `access`$(desc_X,\ i,\ j)$ = `access`$(desc_Z,\ j,\ i)$ +
          `access`$(desc_Y,\ i-1,\ j)$ + `access`$(desc_Y,\ i+1,\ j)$

La tâche de l'exécutif est ici limitée à la résolution des accès aux éléments des tableaux (primitive `access`).

Cas des variables dupliquées

Comme pour le schéma de base une version simplifiée des opérations *Refresh_opt* et *Exec_opt* est à mettre en jeu lorsque la référence apparaissant dans le paramètre $\mathcal{R}_P(\mathcal{I})$ est une référence à une variable dupliquée sur tous les processus. Le travail du compilateur et celui de l'exécutif suivent les mêmes principes mais s'en trouvent simplifiés. Nous l'illustrons sur l'exemple de la compilation du nid de boucles suivant :

pour i **de** 0 **à** $N-1$
  pour j **de** i **à** $N-1$
    $a \;=\; a \;+\; Z[i][j]$

qui est traduit en

$Refresh\_opt(\mathcal{R}(\mathcal{I}),\ \mathcal{R}_P(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$
$Exec\_opt(\mathcal{R}_P(\mathcal{I}),\ \mathcal{A}(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$

où

$\mathcal{I} \;=\; \{i, j\}$
$\mathcal{D}(\mathcal{I}) \;=\; \{0 \leq i \leq N-1,\ i \leq j \leq N-1\}$
$\mathcal{R}(\mathcal{I}) \;=\; \{Z[i][j]\}$
$\mathcal{R}_P(\mathcal{I}) \;=\; \{a\}$
$\mathcal{A}(\mathcal{I}) \;=\; a \;=\; a \;+\; Z[i][j]$

Le scalaire est par nature dupliqué et le tableau Z est partitionné en P groupes de colonnes(voir figure 3.6).

Pour générer le code correspondant à l'opération *Refresh_opt*, le compilateur utilise les mêmes techniques que dans le cas où la variable affectée est distribuée, on aboutit au code SPMD suivant :

pour kZ **de** 0 **à** 3
  si `myself` = `owner_bloc`($desc_Z$, kZ) **alors**
    pour i **de** 0 **à** $128\,kZ\; + 127$
      pour j **de max**(i, $128\,kZ$) **à** $128\,kZ\; + 127$
        `elt_pack_bcast`($desc_Z$, j, i)
    `add_recver_bcast`($desc_Z$)

pour kZ **de** 0 **à** 3
  pZ = `owner_bloc`($desc_Z$, kZ)
  si `myself` $\neq\ pZ$ **alors**

`add_sender_bcast`($desc_Z$, pZ)

`exchange_bcast`($desc_Z$)

Les primitives `add_recver_bcast`(), `add_sender_bcast`() et `elt_pack_bcast`() servent à mettre à jour, respectivement, pour une variable V :

- un booléen indiquant si des éléments sont à diffuser B_V ;
- la liste des processus expéditeurs E_V ;
- la liste des segments à diffuser S_V.

Le booléen B_V sert à forcer la diffusion d'un message vide lorsque, dans le code d'émission, le test sur le possesseur du bloc kZ est passé mais que le nid (i, j) est vide. Cette diffusion est nécessaire car le code de réception dual indiquera, via la liste E_V, qu'un message est attendu.

L'échange effectif des données (diffusions et réceptions) est effectué dans la primitive `exchange_bcast` qui met en œuvre les mêmes mécanismes sur les segments que la primitive `exchange`. Les optimisations effectuées dans le cas des variables distribuées (agrégation de messages, communications directes et non redondantes) se retrouvent également ici.

Contrairement au cas où une référence à une variable distribuée constituait la partie gauche de l'affectation du nid de boucles, il n'y a dans le cas de la référence à une variable dupliquée pas de restriction du domaine d'itération, conformément à la règle des écritures locales. Le compilateur ne se sert donc que du paramètre $\mathcal{D}(\mathcal{I})$ et de l'affectation pour générer le code correspondant à l'opération *Exec_opt*. Sur l'exemple, on aura le code SPMD de calcul suivant :

pour i **de** 0 **à** 511
  pour j **de** i **à** 511
    $a = a +$ `access`($desc_Z$, j, i)

Le rôle de l'exécutif se limite là aussi à l'accès aux variables distribuées.

Allocation mémoire

Mécanisme d'allocation

Dans le schéma de base, la gestion des éléments reçus étaient effectuée simplement et efficacement à l'aide de scalaires temporaires. Pour le schéma optimisé, on doit également fournir un procédé efficace d'allocation et de libération de l'espace mémoire destiné aux éléments reçus. On peut faire les remarques suivantes :

- On ne sait pas au moment de la compilation combien d'éléments seront reçus. Ce nombre pouvant être grand, une gestion dynamique doit être mise en place.

- L'ordre des allocations et des libérations correspond à l'utilisation d'une pile pour les variables locales à une procédure en compilation séquentielle classique : on effectue une série d'allocations au moment de la réception des données distantes lors de l'exécution du *Refresh_opt*. Les données deviennent inutiles à la fin de la primitive *Exec_opt*, elles peuvent être toutes libérées en une seule fois.

Le mécanisme d'allocation utilisé fournit donc deux opérations : l'allocation d'une zone mémoire rendant une adresse (du type de l'allocation dans un tas) et la libération complète de l'espace alloué. Ce mécanisme est bâti au-dessus de l'allocation dans le tas de blocs de mémoire chaînés. L'espace mémoire nécessaire au stockage des segments reçus est alloué dans ces blocs, lorsqu'un bloc est plein, un nouveau bloc est alloué dans le tas. La libération complète consiste à parcourir la liste chaînée des blocs et à les libérer un à un.

Afin d'optimiser le remplissage des blocs, une seule chaîne de blocs reçoit les segments de toutes les variables lues impliquées dans le *Refresh_opt*. Un compromis à dû être fait pour déterminer la taille des blocs : des blocs trop grands gaspillent de la mémoire s'ils ne sont pas remplis, des blocs trop petits multiplient les allocations dans le tas, allocations qui restent des opérations coûteuses en temps. La taille des blocs T a été arbitrairement fixée en fonction de la variable dont les pages sont les plus grandes (cette variable v est déterminée à la compilation et indiquée à l'exécutif). T est égale au maximum entre la taille de deux de ces pages[6] et 10% de la taille de la partition locale de v.

Afin de supprimer le coût de son allocation — qui est faite dès que le processus a besoin d'un élément distant — le premier bloc de la chaîne est alloué statiquement (ceci est réalisé par la déclaration d'une variable statique dans une macro-instruction appelée au début de la boucle parallèle). Une série d'expérimentations nous a permis de découvrir que cette optimisation apportait un gain significatif pour les programmes optimisés possédant une bonne localité puisqu'aucune allocation dans le tas n'était alors nécessaire.

Allocation de segments

L'espace pour les éléments reçus est alloué par segments lors de la réception directe de ceux-ci, ou au moment de la recopie depuis le tampon de réception contenant les segments agrégés. Pour une page donnée et un processeur émetteur, on a un seul segment. Les opérations à effectuer diffèrent légèrement suivant que, pour la variable concernée, le nombre de possesseurs d'une page est égal à 1 ou 2.

- *1 possesseur par page*
  Il est certain que, lorsque que l'on reçoit un segment, la page correspondante n'a fait l'objet d'aucune réception antérieure et n'en fera pas l'objet par la

6. La taille d'un bloc ne peut évidemment pas descendre en dessous de la taille d'une page de la variable possédant les plus grandes pages.

suite. On alloue donc de l'espace uniquement pour le segment et l'entrée de la table des pages est décalée de la valeur de l'offset de début du segment (voir figure 3.8).

- *2 possesseurs par page*
  Il est possible dans ce cas qu'une page fasse l'objet de deux réceptions de segments, venant de deux processus différents. On doit donc garder la trace des pages dans lesquelles un segment a déjà été alloué. La première réception de segment fait l'objet d'une allocation similaire à celle du cas précédent. Lors de la deuxième réception de segment, on alloue de l'espace pour l'union des deux segments et on effectue une recopie du premier segment, l'espace alloué lors de la première réception étant perdu.

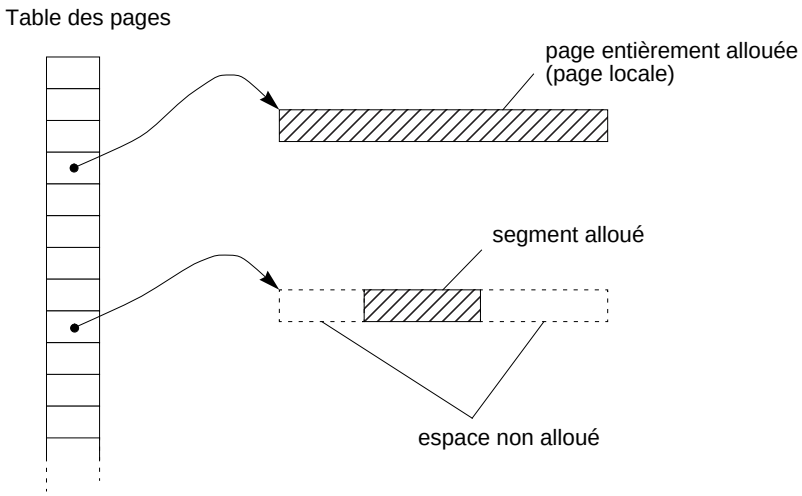
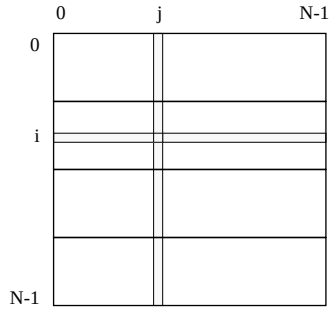


FIG. 3.8. – *Allocation mémoire pour un segment*

Cette allocation de segments permet de garder une consommation mémoire raisonnable même dans les cas tels que celui décrit dans la figure 3.9. Le nid de boucles destiné à calculer un élément (i, j) nécessite l'accès à la ligne i (locale) et à la colonne j (partiellement distante). Les zones d'éléments distants à recevoir sont perpendiculaires au sens des pages. L'allocation de pages entières entraînerait la duplication complète de la matrice.




```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
   for (k=0; k<N; k++)
    A[i][j] = B[i][k] + B[k][j];
```

FIG. 3.9. – *Nécessité d'accès à des segments distants perpendiculaires au sens des pages*

3.3 Mise en œuvre sur une mémoire virtuelle partagée

3.3.1 La mémoire virtuelle partagée Koan

Koan [77, 76] est une mémoire virtuelle partagée (MVP) intégrée dans le système d'exploitation NX de l'Intel iPSC/2. Koan offre la possibilité de partager un espace d'adressage virtuel entre les processus s'exécutant sur les différents nœuds de l'iPSC/2, permettant ainsi de voir la machine comme une machine multiprocesseur à mémoire partagée.

La MVP Koan s'appuie sur les services câblés de mémoire virtuelle de chaque nœud pour la transformation d'adresses virtuelles en adresses physiques et sur le réseau de communications pour les échanges de données entre mémoires locales. Lorsqu'un processeur désire accéder à un mot mémoire qui n'est pas présent dans sa mémoire locale, l'unité de gestion mémoire de la MVP se charge de le rapatrier. Les données sont rapatriées par blocs pour tirer parti de la localité spatiale et rentabiliser la latence du réseau. La taille des blocs correspond à la taille des pages de la mémoire virtuelle du processeur (4096 octets). Pour profiter de la localité temporelle, les copies de pages sont stockées localement dans les mémoires locales. Plusieurs protocoles assurent la cohérence de ces copies : cohérence forte, cohérence faible et gestion de données non modifiables.

L'implémentation de Koan permet d'utiliser également l'iPSC/2 comme une machine à messages : pour chaque processeur, seule une partie de la mémoire est partagée, l'autre partie demeurant privée, c'est-à-dire accessible à ce seul processeur. En outre, on dispose toujours des primitives de communication du système NX.

3.3.1.1 Protocoles de gestion de cohérence

Le protocole de base de gestion de la cohérence des copies de pages implanté dans la MVP Koan est un protocole de gestion de cohérence forte. Il est fondé sur la technique d'invalidation sur écriture : lorsqu'un processeur veut écrire dans une page, toutes les copies de celle-ci sont invalidées au préalable. Un seul processeur est donc propriétaire d'une page à un instant donné, ce propriétaire changeant en fonction des défauts de page.

La gestion de propriétaire est une gestion distribuée statique : chaque processeur est le gestionnaire d'un sous-ensemble des pages qui lui sont attribuées statiquement. Cette attribution peut être effectuée de deux façons : soit les pages sont distribuées cycliquement aux processeurs, soit un bloc de pages contiguës est attribué à chaque processeur. Lors d'un défaut de page, la requête est envoyée au gestionnaire de la page concernée, le gestionnaire la fait ensuite suivre au propriétaire.

Koan fournit la possibilité d'utiliser un protocole de cohérence faible. Il s'applique aux phases parallèles sans dépendances. Durant ces phases, il n'y a pas d'invalidation de pages. Chacun des processeurs n'effectuant des écritures que sur des variables différentes, le maintien de la cohérence peut être retardé jusqu'à la fin de la phase. Plusieurs processeurs peuvent donc écrire simultanément dans leur copie de la même page. A la fin de la phase sans dépendance, les processeurs se synchronisent et fusionnent les copies de pages.

Un troisième protocole de cohérence est applicable aux données partagées non modifiables. Pour ce genre de données, il n'y a pas de problème de cohérence. Seules sont gérées les duplications de pages. Dans ce cas, la notion de gestionnaire de pages est absente puisque le propriétaire ne change pas au cours du temps.

3.3.1.2 Régions partagées

La mémoire virtuelle partagée de Koan est structurée en régions. Chaque région est définie par une adresse virtuelle, une taille, un mode de gestion de propriétaire et un protocole de cohérence. Ces régions peuvent être créées et détruites dynamiquement. On a donc la possibilité de faire coexister plusieurs régions gérées différemment.

3.3.1.3 Autres services

Diffusion de pages

Afin d'améliorer l'efficacité des programmes présentant un schéma d'exécution *producteur-consommateurs*, un mécanisme de diffusion explicite de pages est fourni dans la MVP Koan. En effet lorsque que l'on sait à l'avance que plusieurs processeurs vont lire le même ensemble de données produit préalablement par un seul processeur, il est préférable que le producteur diffuse explicitement les pages concernées plutôt

que de laisser le système résoudre un par un les défauts de page provoqués par les consommateurs.

Outils de synchronisation

Koan offre deux outils de synchronisation en plus de la barrière de synchronisation fournie par le système NX :

- les sections critiques, par l'utilisation de sémaphores ;
- le verrouillage de pages qui permet d'obtenir l'usage exclusif d'une page en lecture-écriture.

3.3.1.4 Interface de programmation

La MVP Koan est accessible via une bibliothèque dont les fonctions sont appelables depuis un programme SPMD en langage *C*. Nous présentons brièvement les fonctions utilisées pour la mise en œuvre de la machine abstraite PANDORE.

void **init_koan**() initialise les structures de données nécessaires au fonctionnement de la MVP.

int **create_region**(*size*) permet d'allouer une région en précisant sa taille. On récupère alors un identificateur de région partagée.

void **free_region**(*id*) permet de libérer une région partagée donnée.

char ***map_region**(*id*, *fmap*, *cp*) définit, pour une région partagée donnée, la gestion de propriétaire *fmap* (`BLOCK` ou `MODULO`) et le protocole de cohérence *cp* (cohérence faible `WEAK`, cohérence forte `RW` ou lecture seule `RO`). On récupère l'adresse virtuelle du début de la région partagée.

void **begin_broadcast**(*id*, *node*, *start*, *end*) ouvre une section *producteur-consommateurs* en précisant le nœud producteur, le début et la fin de la zone mémoire qui sera à diffuser.

void **end_broadcast**() termine une section *producteur-consommateurs* en diffusant les pages concernées.

void **gsync**() synchronise l'ensemble de processeurs. Cette primitive est en fait une primitive du système NX.

3.3.2 Utilisation pour la machine abstraite Pandore

Le principe d'utilisation de la MVP Koan comme support d'exécution pour la machine abstraite PANDORE est d'exploiter les mécanismes de la MVP à la fois pour l'accès aux données distribuées et pour les communications.

On place les tableaux distribués dans les régions de mémoire virtuelle partagées de façon à attribuer à un processus les pages correspondant aux blocs possédés (au sens de PANDORE) par ce processus. Toutes les autres données (dupliquées) sont stockées dans la partie de mémoire privée. Comme les données distribuées sont placées en mémoire virtuelle partagée, on considère que tous les éléments sont accessibles sur l'ensemble des processus, il n'y a donc pas de distinction au niveau des accès entre données locales et données distantes.

C'est le protocole de cohérence forte qui est activé pour toutes les régions partagées, il permet d'effectuer des communications lors de la lecture d'une donnée distante. En effet, l'application de la règle des écritures locales garantit qu'un processus n'écrira que dans les pages qui lui ont été attribuées. En revanche, il est susceptible de lire une donnée se trouvant dans une page attribuée à un autre processus. On aura alors un défaut de page en lecture qui rapatriera la donnée sur le processus demandeur. Les autres protocoles ne sont pas utilisés. Le protocole de gestion de cache en lecture seule ne peut être utilisé puisque l'on effectue aussi des écritures dans les variables distribuées. Quant au protocole de cohérence faible, il n'est utile que lorsque l'on veut autoriser deux processeurs à écrire dans la même page sans qu'il y ait de défaut de page en écriture ; cette situation ne peut arriver puisque chaque page est attribuée à un seul processeur et que la règle des écritures locales évite tout défaut de page en écriture.

Nous verrons par la suite que la résolution de défaut de page en lecture n'est pas le seul moyen de communication employé, les mécanismes de diffusion de pages de Koan seront également mis à profit. Par ailleurs, pour des raisons d'efficacité, nous serons amenés à utiliser des communications explicites (en utilisant les primitives du système NX sous-jacent) au lieu de nous reposer exclusivement sur les mécanismes de communication internes à Koan.

3.3.3 Gestion des tableaux distribués

3.3.3.1 Principe

Comme la MVP Koan permet de gérer plusieurs régions partagées distinctes, on placera les éléments d'un tableau distribué donné dans une région partagée spécifique. Tous les éléments d'un tableau distribué sont en région partagée Koan, la propriété d'uniformité des accès est donc évidemment maintenue : par définition, qu'il s'agisse

d'éléments locaux ou rapatriés depuis un autre processeur lors d'un défaut de page, la forme de l'accès ne change pas.

Afin de maîtriser les défauts de page, et en particulier d'éviter les défauts de page en écriture, on impose qu'il n'y ait qu'un seul possesseur par page. On fera donc coïncider les pages sur les blocs, l'application de la règle des écritures locales garantissant l'absence de défaut de page en écriture.

Le problème est ici de définir la transformation $\mathcal{L}$ de l'espace multi-dimensionnel d'un tableau distribué PANDORE en l'espace mono-dimensionnel que représente la région partagée Koan et donc de définir l'accès à un élément de région partagée (adresse dans cette région) à partir d'un vecteur d'indices$(i_0, \ldots, i_{n-1})$ représentant l'accès au tableau original.

On choisit cette fonction de linéarisation $\mathcal{L}(i_0, \ldots, i_{n-1})$ afin de

- minimiser la taille des régions partagées créées ;
- minimiser le coût d'évaluation de la fonction d'accès dérivée de $\mathcal{L}$;
- conserver une partie de la contiguïté des éléments des tableaux distribués.

3.3.3.2 Prise en compte du placement

Si l'on impose qu'une page de région partagée Koan n'est possédée que par un seul processeur, il n'y aura pas de changement de propriétaire durant la phase distribuée. Il semble donc intéressant de faire en sorte que, pour une page donnée, ce propriétaire soit aussi le gestionnaire de la page : les requêtes de défaut de page en lecture qui sont adressées au gestionnaire de la page n'auront pas à transiter dans le réseau pour atteindre le propriétaire. Il n'est pas possible d'atteindre cette situation idéale simplement (sans compliquer coûteusement $\mathcal{L}$) dans le cas général car les possibilités de placements de pages Koan sont moins grandes que celles des blocs des tableaux distribués PANDORE, le placement de pages Koan se faisant suivant un seule dimension.

La méthode choisie consiste à utiliser le protocole de cohérence forte et à établir la correspondance possesseur–gestionnaire de page dans les cas où la distribution PANDORE est aussi exprimable par un placement Koan. On fait coïncider les fonctions de placement PANDORE et Koan (`wrapped` → `MODULO` et `regular` → `BLOCK`) et on effectue une permutation des coefficients de linéarisation en fonction de l'ordre défini dans les paramètres de la fonction de placement PANDORE. Cette permutation n'affecte en rien l'efficacité de $\mathcal{L}$.

On peut noter que la limitation des possibilités de Koan pour le placement de pages est purement arbitraire. Ont été implantées les deux politiques les plus couramment utilisées. A priori, n'importe quel placement est envisageable.

3.3.3.3 Définition de $\mathcal{L}$

Soit la déclaration de tableau distribué suivante

$$int\ V[h_0]\cdots[h_{n-1}]\ by\ block\ (s_0,\ldots,s_{n-1})\ map \left| \begin{array}{l} regular \\ wrapped \end{array} \right| (d_0,\ldots,d_{n-1})$$

De la même façon que pour la mise en œuvre sur la POM, on choisit une dimension particulière, δ, la dimension suivant laquelle le tableau n'est pas distribué. Si plusieurs dimensions sont non distribuées, on choisit celle correspondant à la plus grande taille de bloc. S'il n'en existe pas, δ est la dimension dans laquelle la taille des blocs est une puissance de deux et à défaut la plus grande.

On distingue deux cas pour la définition de $\mathcal{L}$, suivant qu'il existe une dimension non distribuée ou pas.

Cas où il existe une dimension non distribuée

Dans le cas où il existe une dimension non distribuée — cette dimension est alors la dimension δ —, on étend la taille du tableau dans cette dimension (figure 3.10) et on utilise pour $\mathcal{L}$ la fonction de linéarisation de C pour les tableaux multi-dimensionnels appliquée à une permutation du vecteur d'indices. Cette permutation place l'index correspondant à la dimension δ en dernière position et permute les autres index en fonction des paramètres de la fonction de placement.

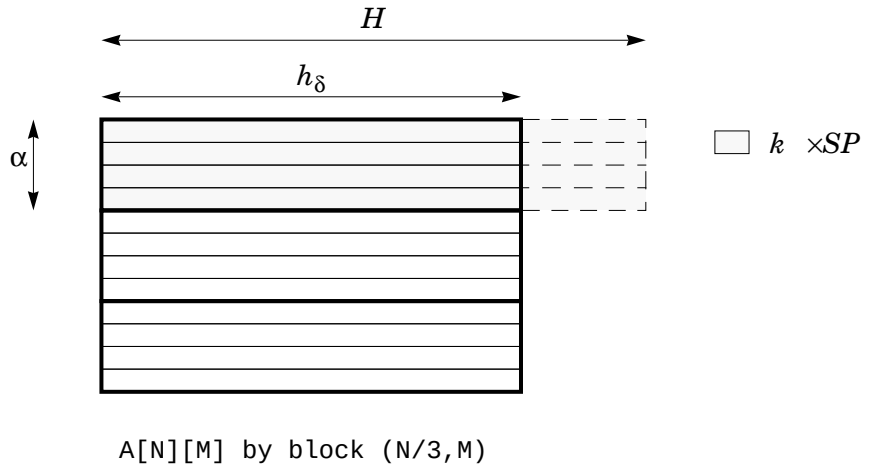


FIG. 3.10. – *Définition des pages avec une dimension non distribuée*

Pour une description plus formelle, on adopte les notations suivantes :

SP la taille de la page Koan (en nombre d'éléments)

$SB = \prod_k s_k$ la taille d'un bloc

$\alpha = \frac{SB}{s_\delta} = \prod_{k \neq \delta} h_k$

H la dimension h_δ étendue à déterminer

$SB' = \alpha H$ la taille d'un bloc étendu

On étend la dimension h_δ à H afin que le bloc étendu contienne un nombre entier de pages. SB' est donc le plus petit multiple de α et de SP plus grand ou égal à SB :

$$SB' = \left\lceil \frac{SB}{ppcm(\alpha, SP)} \right\rceil ppcm(\alpha, SP)$$

d'où

$$H = \left\lceil \frac{s_\delta \, pgcd(\alpha, SP)}{SP} \right\rceil \frac{SP}{pgcd(\alpha, SP)}$$

On obtient donc

$$\mathcal{L}(i_0, \ldots, i_{n-1}) = \sum_{k=0}^{n-1} \left(i'_k \prod_{l=k+1}^{n-1} h'_l \right)$$

où, si l'on pose $d_\gamma = \delta$,

$i'_{n-1} = i_\delta$ et $h'_{n-1} = H$
$\forall k \in 0, \ldots, \gamma - 1 \;\; i'_k = i_{d_k}$ et $h'_k = h_{d_k}$
$\forall k \in \gamma, \ldots, n-2 \;\; i'_k = i_{d_{k+1}}$ et $h'_k = h_{d_{k+1}}$

Cas où toutes les dimensions sont distribuées

Dans le cas où toutes les dimensions sont distribuées (figure 3.11), l'extension de la dimension δ ne se fait pas uniquement par l'extrémité, on génère des « trous » entre chaque limite de bloc. La fonction de linéarisation $\mathcal{L}$ est alors linéaire par morceaux.

La taille étendue H est donnée par le plus petit multiple de SP supérieur ou égal à h_δ :

$$H = \left\lceil \frac{h_\delta}{SP} \right\rceil SP$$

On obtient

$$\mathcal{L}(i_0, \ldots, i_{n-1}) = \sum_{k=0}^{n-1} \left(i'_k \prod_{l=k+1}^{n-1} h'_l \right)$$

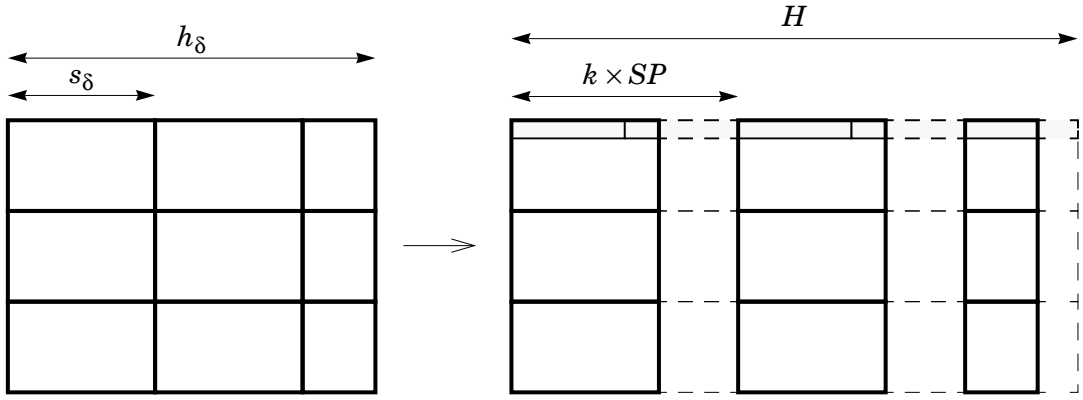


FIG. 3.11. – *Définition des pages avec toutes les dimensions distribuées*

où, avec $d_\gamma = \delta$,

$$i'_{n-1} = i_\delta + (i_\delta \, div \, s_\delta) \times \left(\left\lceil \frac{s_\delta}{SP} \right\rceil SP - s_\delta \right) \text{ et } h'_{n-1} = H$$
$$\forall k \in 0, \ldots, \gamma - 1 \;\; i'_k = i_{d_k} \text{ et } h'_k = h_{d_k}$$
$$\forall k \in \gamma, \ldots, n-2 \;\; i'_k = i_{d_{k+1}} \text{ et } h'_k = h_{d_{k+1}}$$

3.3.3.4 Mise en œuvre de l'allocation et de l'accès

On profite du fait que le code généré soit en C pour faire en sorte que la fonction de linéarisation $\mathcal{L}$ soit élaborée en grande partie par le compilateur cible. Pour cela, le compilateur transforme la déclaration d'un tableau distribué $V[h_0] \ldots [h_{n-1}]$ en l'équivalent dynamique de la déclaration d'un tableau $V'[h'_0] \ldots [h'_{n-1}]$. L'accès à un tableau distribué noté $V[i_0] \ldots [i_{n-1}]$ dans le programme source peut être alors exprimé comme une référence $V'[i'_0] \ldots [i'_{n-1}]$. Par exemple, le code PANDORE

```
float A[200][240][10] by block(200,60,1) map wrapped(2,1,0);

 ... A[i][j][k] ...
```

est traduit en

```
typedef float tA[240][256];
tA *vA;
int idA;

idA = create_region(600*4096);
```

```
vA = (tA*)map_region(idA, MODULO, RW);

... access(A[i][j][k]) ...
```

où l'appel à la macro `access(A[i][j][k])` s'expanse en `vA[k][j][i]`.

Contrairement à la mise en œuvre sur machine à messages, on ne contrôle pas complètement l'allocation mémoire correspondant aux tableaux distribués. Ici, c'est la MVP qui alloue la partition locale et l'espace pour les copies de pages temporaires, l'unité d'allocation étant toujours la page. L'extension de la taille du tableau dans la direction des pages (de 200 à 256 dans l'exemple) provoque donc un accroissement réel de l'occupation mémoire, ce qui n'était pas le cas lors de la mise en œuvre sur la POM où l'on évitait d'allouer de l'espace pour les éléments fictifs. C'est pour cette raison que l'on n'a pas étendu les autres dimensions à des puissances de deux, l'accélération du calcul de l'adresse aurait eu pour prix un trop grand surcoût mémoire.

3.3.3.5 Calcul de possesseur

La MVP Koan ne fournit pas d'accès à l'identité du propriétaire d'une page qui nous permettrait de construire la fonction rendant le possesseur d'un élément de tableau. C'est pourquoi celle-ci est définie de la même façon que pour la mise en œuvre sur machine à messages. Chaque processus stocke la table des possesseurs de pages qui indique, pour chaque page, le numéro du processus possédant cette page. La table est remplie au début de la phase distribuée à l'aide d'une fonction générée par le compilateur telle qu'elle est décrite dans la section 3.2.2, la situation étant ici plus simple car le possesseur d'une page est forcément unique. Pour la détermination du possesseur d'un élément $V[I]$, le compilateur produit à partir de I et des paramètres de distribution l'expression du numéro de page correspondant, expression dont la valeur servira d'index dans la table des possesseurs.

3.3.3.6 Initialisation des données

Au début de la phase distribuée, les éléments de chaque partition locale est éventuellement reçue depuis le processus hôte (cas des variables en mode `IN` et `OUT`). Bien que Koan fournisse un service permettant d'initialiser des régions partagées avec des données se trouvant sur l'hôte, ce service ne peut être utilisé car la correspondance possesseur PANDORE–gestionnaire Koan n'est pas systématiquement complète. Le transfert de données est donc réalisé explicitement par messages. Les pages sont construites sur l'hôte et envoyées au possesseur PANDORE et donc parfois à un processeur qui n'est pas le gestionnaire (le propriétaire initial). Dans ce cas, un défaut de page en écriture se produit et la correspondance propriétaire–possesseur est établie définitivement.

3.3.4 Schémas de compilation

3.3.4.1 Le schéma de base

Le schéma de compilation de base ne fait pas intervenir les mécanismes de transfert de données de la MVP Koan mais repose sur des communications par messages identiques à celles de la mise en œuvre sur la POM. Une affectation $\mathcal{A}$ produira donc la séquence d'opérations suivante :

Refresh_base(USE($\mathcal{A}$), DEF($\mathcal{A}$))
Exec_base(DEF($\mathcal{A}$), $\mathcal{A}$)

les opérations *Refresh_base* et *Exec_base* étant mises en œuvre de la même façon que pour la machine à messages POM, hormis l'accès aux éléments de tableaux distribués. Cette remarque vaut également pour le cas où les variables écrites sont des variables dupliquées.

Le choix de ce schéma est justifié par le fait que l'application du schéma utilisant les défauts de page comme moyen de communication est toujours moins efficace. En effet, considérons le schéma de compilation prévu pour la MVP, proposé au chapitre 2[7], dans lequel l'affectation $\mathcal{A}$ est transformée en

Sync_base(USE($\mathcal{A}$), DEF($\mathcal{A}$))
Exec_base(DEF($\mathcal{A}$), $\mathcal{A}$)
Sync_base(DEF($\mathcal{A}$), USE($\mathcal{A}$))

Pour simplifier considérons que les ensembles USE($\mathcal{A}$) et DEF($\mathcal{A}$) sont des singletons afin que l'opération *Sync_base* ne fasse intervenir qu'au plus deux processeurs. La méthode la plus efficace pour la synchronisation entre deux processeurs sur une machine à mémoire distribuée telle que l'iPSC/2 est l'échange de messages (l'utilisation d'une primitive de synchronisation globale de tous les processeurs comme `gsync` est à exclure ici). La mise œuvre de l'opération *Sync_base* prend donc la forme

Sync_base(v_1, v_2) ≡

si `myself` = *owner*(v_2) **alors**
  si *owner*(v_2) ≠ *owner*(v_2)
    alors envoyer message synchro à *owner*(v_1)
  fsi
fsi
si `myself` = *owner*(v_1) **alors**

7. On utilise ici pour simplifier des synchronisations asymétriques, l'utilisation d'opérations symétriques renforcerait le choix retenu.

si *owner*(v_1) $\neq$ *owner*(v_2)
alors recevoir message synchro de *owner*(v_2)
fsi
fsi

L'opération *Sync_base* ainsi mise en œuvre est quasiment identique à l'opération *Refresh_base*, la seule différence étant que le message transporté dans le *Refresh_base* contient une donnée, différence négligeable car il s'agit d'un seul élément de tableau (la latence prédomine très largement dans le temps de transfert).

Par ailleurs, le bien-fondé du choix de l'application du schéma *Refresh/Exec* est renforcé par le fait que les transferts de données se font forcément par pages si l'on applique le schéma *Sync/Exec/Sync*, ce qui peut occasionner l'échange de nombreux éléments superflus.

3.3.4.2 Le schéma optimisé

C'est lors de l'application du schéma de compilation optimisé que les spécificités de la MVP Koan sont utilisées. Lors de la compilation d'un nid commutatif comportant une affectation, le compilateur distingue le cas où la variable écrite est un tableau distribué de celui où il s'agit d'une variable dupliquée.

Cas des variables distribuées

Le schéma adopté est alors le schéma *Sync/Exec/Sync*. Par exemple le nid commutatif

pour i **de** 0 **à** N
pour j **de** i **à** N
$X[i][j] \ = \ X[i+1][j] \ + \ Y[i]$
fpour
fpour

sera traduit en

$$\begin{array}{l} \mathit{Sync\_opt}(\{X[i+1][j], Y[i], X[i]\}, \{0 \leq i \leq N, \ i \leq j \leq N\}) \\ \mathit{Exec\_opt}(\{X[i]\}, \{0 \leq i \leq N, \ i \leq j \leq N\}, X[i][j] \ = \ X[i+1][j] \ + \ Y[i]) \\ \mathit{Sync\_opt}(\{X[i+1][j], Y[i], X[i]\}, \{0 \leq i \leq N, \ i \leq j \leq N\}) \end{array}$$

L'opération *Sync_opt* effectue une synchronisation de tous les processus. Rigoureusement tous les processus n'ont pas à être synchronisés mais la synchronisation globale est efficacement mise en œuvre : elle correspond à une primitive système de bas niveau (la primitive `gsync` du système NX) et ne nécessite aucun calcul. En

effet, effectuer la synchronisation des seuls processus spécifiés dans les paramètres de *Sync_opt* nécessiterait un calcul préalable de l'ensemble de ces processus et le transfert d'un nombre éventuellement important de messages.

L'opération *Exec_opt* est mise en œuvre de la même façon que sur la POM : à partir des paramètres de l'opération (ensembles de références et domaine), le compilateur constitue un système de contraintes pour produire un code de parcours restreint, pour chaque processus, aux éléments à calculer. La forme du code généré ne diffère de celle du code généré pour la POM qu'en ce qui concerne les accès aux éléments de tableaux. L'exécution de ce code peut provoquer des défauts de page en lecture lorsqu'une ou plusieurs des opérandes en partie droite de l'affectation est distante ainsi que des invalidations lorsque un élément écrit pour la première fois dans la boucle avait été préalablement lu par un autre processus (à noter que cette lecture a forcément eu lieu avant la boucle).

Cas des variables dupliquées

Schéma

L'utilisation du schéma *Sync/Exec/Sync* qui emploie la résolution des défauts de page lors de l'*Exec* comme moyen de communication ne convient pas au cas où la variable écrite est une variable dupliquée. En effet, dans ce cas, tous les processus (sauf le possesseur de la variable distribuée lue) vont effectuer un défaut de page en lecture. Les communications sont donc en mode point à point et ne tirent pas parti des possibilités de diffusion rapide du système. De plus, l'exécution de l'opération *Exec_opt* se faisant souvent de manière relativement synchrone sur l'ensemble des processus, il s'en suit un goulot d'étranglement pour l'accès à la page sur le processus qui la possède puisque les demandes de pages y sont sérialisées.

Ce goulot d'étranglement peut être évité en explicitant les copies de données. On utilise le schéma *Sync/Get/Exec/Sync* décrit au chapitre 2. Par exemple, la compilation du nid de boucles suivant :

pour i **de** 0 **à** $N-1$
  pour j **de** i **à** $N-1$
    $a = a + Z[i][j]$
  fpour
fpour

produira

$$\mathit{Sync\_opt}(\mathcal{R}(\mathcal{I}) \cup \mathcal{R}_P(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$$
$$\mathit{Get\_opt}(\mathcal{R}(\mathcal{I}),\ \mathcal{R}_P(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$$
$$\mathit{Exec\_opt}(\mathcal{R}_P(\mathcal{I}),\ \mathcal{A}(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$$
$$\mathit{Sync\_opt}(\mathcal{R}(\mathcal{I}) \cup \mathcal{R}_P(\mathcal{I}),\ \mathcal{D}(\mathcal{I}))$$

où

$$\begin{aligned}
\mathcal{I} &= \{i, j\} \\
\mathcal{D}(\mathcal{I}) &= \{0 \leq i \leq N-1,\ i \leq j \leq N-1\} \\
\mathcal{R}(\mathcal{I}) &= \{Z[i][j]\} \\
\mathcal{R}_P(\mathcal{I}) &= \{a\} \\
\mathcal{A}(\mathcal{I}) &= a = a + Z[i][j]
\end{aligned}$$

Le scalaire a est par nature dupliqué et le tableau Z est partitionné en P groupes de colonnes (voir figure 3.6).

L'opération Sync_opt

Comme précédemment, l'opération *Sync_opt* effectue la synchronisation de tous les processus. En revanche, il n'y a pas ici de synchronisation d'un sur-ensemble des processeurs concernés puisque tous les processeurs possèdent la variable écrite.

L'opération Get_opt

L'opération *Get_opt* permet de diffuser les pages contenant des variables à rafraîchir. On ne peut cependant pas utiliser les primitives de diffusion du système NX car on n'a pas la possibilité de contrôler l'allocation des pages de région partagée, cette allocation étant strictement liée à la réception de page après défaut. Nous avons donc exploité les fonctions de diffusion de pages disponibles dans la MVP Koan. Malheureusement, les primitives fournies sont adaptées à un schéma *1 producteur / n consommateurs*. Il a donc été nécessaire de produire un code supplémentaire pour tenir compte du schéma *n producteurs / n consommateurs* qui est en vigueur dans les opérations *Get_opt* et *Exec_opt*. En effet, pour utiliser les primitives `begin_broadcast` et `end_broadcast` de Koan, il est nécessaire que tous les processus connaissent la sous-région qui est à diffuser. Chaque processus doit donc calculer la description des échanges concernant l'ensemble de processus.

Le travail du compilateur est similaire à celui effectué pour l'opération *Refresh_opt* sur la POM (il n'y a toutefois pas de test sur la possession des blocs afin que tous les processus calculent les ensembles d'éléments à diffuser). Sur l'exemple, le compilateur produit le code SPMD suivant :

pour kZ **de** 0 **à** 3
  pZ = `owner_bloc`($desc_Z$, kZ)
  pour i **de** 0 **à** $128\,kX + 127$
    pour j **de max**(i, $128\,kZ$) **à** $128\,kZ + 127$
      `elt_pack_bcast`($desc_Z$, j, i, pZ)

pour kZ **de** kX **à** 3

pZ = `owner_bloc`($desc_Z$, kZ)
`add_sender_bcast`($desc_Z$, pZ)

`exchange_bcast`($desc_Z$)

La primitive `add_sender_bcast`($desc_V$, p) ajoute le processus p à la liste des expéditeurs E_V. La primitive `elt_pack_bcast`($desc_V$, pg, of, p) permet de mettre à jour la liste des pages à diffuser par le processus p, $P_V[p]$. La primitive `exchange_bcast` provoque les mouvements de pages nécessaires en appelant successivement les primitives de diffusion de pages Koan `begin_broadcast` et `end_broadcast` pour chaque processus de la liste E_V.

L'opération Exec_opt

Le code de l'opération *Exec_opt* reste lui inchangé par rapport à la mise en œuvre sur la POM, mis à part l'expression des accès aux éléments de tableaux distribués. Évidemment, aucun défaut de page n'intervient durant l'exécution de l'*Exec_opt*, toutes les pages distantes nécessaires ayant été rapatriées explicitement durant l'opération *Get_opt*. Cependant, des invalidations peuvent également avoir lieu.

On peut noter que le compilateur utilise conjointement deux techniques de repérage d'un élément, d'une part par son adresse dans une région de mémoire virtuelle (lors de l'*Exec_opt*) et d'autre part, identiquement à la mise en œuvre sur la POM, par un couple (page, offset).

3.4 Comparaison des mises en œuvre

Les deux mises en œuvre des opérations des machines abstraites PANDORE que nous avons présentées, *i.e.* sur la POM et sur la MVP Koan, ont de nombreux points communs dans la mesure où elles sont définies dans un même cadre, exposé au chapitre 2. Elles comportent cependant des différences à plusieurs égards, différences que nous détaillons dans la suite.

Effort de développement dans le compilateur et l'exécutif

Les mêmes techniques de compilation et en particulier d'optimisations sont utilisées dans les deux mises en œuvre. La construction de polyèdres et la génération de leur parcours s'avèrent être nécessaires dans les deux cas. Ceci est évidemment valable pour la mise en œuvre de l'opération *Exec*, commune aux deux mises en œuvre, mais aussi pour les opérations de rafraîchissement de variables *Refresh* et *Get*. On notera toutefois que le travail du compilateur est facilité dans le cas de l'application du schéma *Sync/Exec/Sync* sur MVP puisque les ensembles d'éléments à émettre et recevoir ne sont pas à calculer.

L'exécutif de la version s'appuyant sur la MVP est plus simple que celui utilisant la POM dans la mesure ou une grande partie des primitives de communication sont réalisés par le système de MVP.

Accès aux données distribuées

La rapidité d'accès aux éléments de tableaux distribuée était un objectif majeur lors du développement des machines abstraites sur la POM et sur la MVP Koan.

Dans les deux cas, un accès efficace est obtenu. Il peut être considéré comme optimal sur la MVP dans la mesure le temps d'accès à un élément de tableau distribué est identique à celui d'une tableau non distribué. On profite dans ce cas des mécanismes cablés de l'unité de gestion de la mémoire.

L'adressage paginé logiciel mis en œuvre pour la POM peut occasionner un léger surcoût que l'on ne retrouve pas lors de l'utilisation de la MVP. Il offre cependant un gain certain par rapport aux adressages proposés par les autres projets de recherche dans le domaine, dont le coût peut être prohibitif dans de nombreux cas.

Communications

Les mécanismes de la MVP n'ont pas été considérés pour la mise en œuvre des communications lors de l'application des schémas de base. On a tenu compte du fait que la MVP Koan n'interdit pas l'utilisation de l'iPSC/2 comme machine à messages. L'utilisation stricte de la MVP pour les communications dans ce cadre peut entraîner un surcoût notable du fait de la granularité forcée des échanges par résolution de défaut de page (risque de transfert d'une page entière pour quelques éléments nécessaires).

Pour ce qui est de la mise en œuvre des schémas optimisés, l'ordonnancement des communications point à point diffère suivant que l'on utilise le schéma *Refresh/Exec*, sur la POM, ou le schéma *Sync/Exec/Sync*, sur la MVP. On a des communications effectuées « à la demande » lors de l'*Exec_opt* sur la MVP alors qu'elles sont groupées dans le *Refresh_opt* dans la mise en œuvre sur la POM. A priori, on ne peut décider de la supériorité d'un schéma sur l'autre. Le regroupement des communications dans le *Refresh* permet une meilleure vectorisation. Les communications par résolution de défauts de page, en vigueur dans la mise en œuvre sur la MVP, ne sont pas contrôlables, elles peuvent être effectuées de manière très parallèle ou conduire à des goulots d'étranglement lorsque de plusieurs défauts d'une même page interviennent au même moment.

Dans les deux cas, les communications redondantes sont évitées. Cela est fait via la mémorisation de segments sur la POM. En ce qui concerne la MVP, une fois qu'un élément a été communiqué via la résolution d'un défaut de page, la copie

de la page reste cohérente pendant toute la durée de l'exécution du nid de boucle, l'élément n'est donc pas re-transférer lors d'un accès ultérieur.

On peut avoir des communications superflues dans les deux mises en œuvre. Le risque de transférer des éléments en trop est toutefois accru sur la MVP parce que les communications se font nécessairement par pages entières.

L'utilisation de la diffusion est plus directe dans la mise en œuvre sur la POM. Elle ne constitue en fait qu'un cas simplifié lors de la mise en œuvre du *Refresh_opt*. En revanche, pour utiliser la diffusion sur la MVP, il a fallu mettre en œuvre des techniques spécifiques (*i.e.* s'adapter aux particularités des primitives de diffusion de Koan) qui étaient inutiles pour les communications point à point. La diffusion n'est pas un mécanisme de base de la MVP, elle constitue en fait une extension relevant de l'adjonction de services propres à une machine à messages.

Allocation mémoire

La mise en œuvre sur la POM permet une gestion très fine de la mémoire, permettant d'éviter d'allouer inutilement de la mémoire à la fois pour les données locales et pour les données reçues. La possibilité de contrôler la communication directe et l'agrégation de messages permet d'adopter un compromis temps de transfert/taille de tampons adapté à une plate-forme donnée.

La gestion de la mémoire est bien moins souple dans le cas de l'utilisation de la MVP. L'adaptation des pages PANDORE sur les pages de la MVP peut être coûteuse. En outre, le contrôle de l'allocation mémoire pour les éléments distants est impossible. La gestion de cette mémoire suit forcément la politique de gestion de la mémoire cache prévu dans Koan (taille de cache fixée et politique de remplacement de type LRU). Il peut s'en suivre une consommation mémoire très importante qui limite la taille des problèmes traités.

De façon générale, on constate que la mise en œuvre de la machine abstraite PANDORE sue la POM permet un contrôle plus important sur l'exécution du programme et offre potentiellement de meilleures garanties d'efficacité. L'utilisation de la MVP doit permettre d'atteindre dans certains cas une plus grande efficacité mais le fait que peu de contrôle statique soit autorisé rend possible un écroulement des performances.

Il ne semble pas efficace d'utiliser une machine n'offrant que des services de MVP. La mixité des supports d'exécution (passage de messages et MVP) sur une même plate-forme paraît en tous cas une caractéristique utile. Cette mixité pourrait être plus poussée que dans le cas expérimenté ici (NX et Koan sur iPSC/2) afin d'éviter le doublement de structures de données (*e.g.* table des pages et table des possesseurs).

Des expérimentations et plus généralement un travail d'évaluation des perfor-

mances des programmes générés sont indispensables à la confirmation de ces hypothèses. Le chapitre 4 tentera d'apporter des éléments de réponse en la matière.

Chapitre 4

Expérimentation et évaluation de performance

L'étude des performances des programmes produits par compilation dirigée par les données est une activité importante de la plupart des projets qui développent des prototypes d'environnements dans ce domaine.

Après avoir exposé les raisons de cette activité, nous présenterons dans ce chapitre une série d'expérimentions effectuées avec le système de compilation PANDORE. Celles-ci nous permettront de vérifier l'efficacité des techniques décrites au chapitre précédent ainsi que de détailler certains aspects de programmation avec un langage tel que le langage PANDORE.

Dans la dernière partie du chapitre, nous nous attacherons à préciser la notion d'*évaluation* de performances dans le contexte de la compilation par distribution de données. Cette notion recouvre des aspects concernant la mesure ou l'estimation d'indices de performances (l'indice de base étant le temps d'exécution du programme) ainsi que des aspects touchant à l'interprétation de ces indices. Nous décrirons des travaux réalisés par d'autres projets dans ce domaine avant de détailler les outils développés dans l'environnement PANDORE.

4.1 Problématique

L'approche de la compilation dirigée par les données vise essentiellement à paralléliser les applications de calcul scientifique. Ces applications effectuent des calculs réguliers sur de très grandes données elles-mêmes régulières. L'obtention de bonnes performances est un critère prépondérant dans le développement de ce genre d'applications, il est donc crucial que les compilateurs soient effectivement capables de répondre aux exigences de performance. Si la validité de l'approche de la paral-

lélisation par distribution de données est maintenant reconnue, les prototypes de compilateurs doivent encore faire leur preuve en matière de performance.

Ceci passe par des expérimentations complètes (compilation, exécution) sur des machines parallèles réelles pour prendre en compte tous les éléments du système de compilation.

Afin de comparer les choix de mise en œuvre, un certain nombre de noyaux d'applications (*benchmarks*) sont utilisés par les différents projets développant des compilateurs afin de tester leur capacité à obtenir des performances pour des codes sensés être représentatifs des applications de calcul scientifique.

Toutefois, l'obtention de bonnes performances sur les noyaux d'application n'est qu'une première étape de la validation de prototypes de compilateurs. Il est nécessaire d'effectuer des expérimentations sur des applications « réelles » pour prendre en compte le facteur d'échelle, c'est-à-dire le passage à des codes beaucoup plus longs, plus complexes, faisant intervenir des données plus grandes ou plus nombreuses. Des considérations telles que le temps de compilation, la taille du code obtenu et la quantité de mémoire nécessaire à son exécution deviennent alors importantes.

L'évaluation des performances des programmes très simples est une chose relativement facile ; mesurer le temps d'exécution parallèle et le comparer au temps d'exécution séquentiel suffit souvent. Hélas, dans le cas général, il est bien plus difficile de dire si les performances obtenues sont bonnes, sachant que l'on ne peut rarement atteindre la performance idéale (le temps d'exécution séquentiel est divisé par P pour une exécution parallèle sur P processeurs). Cet optimal théorique ne peut même pas être connu si la taille des données ou des contraintes de temps ne permettent pas une exécution séquentielle.

La difficulté provient d'abord du fait qu'il s'agit d'analyser les performances de programmes parallèles. La complexité engendrée par la présence de plusieurs flots de contrôles et de plusieurs espaces de données ainsi que l'indéterminisme des exécutions rendent difficiles la compréhension du comportement du programme et l'interprétation de ses performances.

Le fait que ces programmes ne soient pas écrits manuellement mais générés par un système de compilation ajoute à la difficulté. La distance entre le programme soumis au compilateur et celui qui est effectivement exécuté est grande. Les facteurs influant sur les performances du programme sont multiples (code, distribution des données, compilateur, exécutif, système et architecture cibles).

Il est nécessaire de développer des techniques et des outils aidant à l'évaluation des performances, à la fois pour le programmeur et les développeurs du système de compilation. La dernière partie de ce chapitre détaillera ces aspects et présentera le travail réalisé dans le cadre du système Pandore.

4.2 Expérimentation

4.2.1 Expérimentation sur machine à messages

Un certain nombre d'expérimentations ont été menées sur l'Intel iPSC/2 disponible à l'Irisa. Cette machine parallèle comporte 32 nœuds organisés suivant une topologie hypercube et reliés à un hôte chargé des entrées/sorties. Les nœuds sont construits autour de microprocesseurs 80386 et comprennent 4 Mo de mémoire chacun.

Les performances ont été évaluées en mesurant d'une part le temps du programme séquentiel écrit en *C* et d'autre part le temps parallèle du programme généré par PANDORE, c'est-à-dire le maximum des temps pris pour la phase distribuée sur les nœuds. Le programme séquentiel a été exécuté sur un des nœuds pour éviter les perturbations dues aux processus système tournant sur l'hôte. La limitation de la mémoire sur un nœud ne nous a pas permis d'exécuter les programmes séquentiels pour toutes les tailles de données considérées. Les temps pour ces tailles ont été extrapolés par calcul de la complexité des programmes et en tenant compte des facteurs constants déduits des temps mesurés sur les tailles plus petites, cette méthode s'avérant, sur cette plate-forme, plus fiable que l'extrapolation graphique.

Les performances sont exprimées en terme d'*accélération* définie comme suit :

$$Acc = \frac{Ts}{Tp} \quad \text{où} \quad Ts \text{ est le temps séquentiel et } Tp \text{ le temps parallèle.}$$

Afin de pouvoir comparer les performances indépendamment du nombre de processeurs utilisés, on utilisera l'*efficacité* :

$$Eff = \frac{Acc}{P} \quad \text{où } P \text{ est le nombre de processeurs.}$$

4.2.1.1 Noyaux

Jacobi

La méthode de relaxation itérative de Jacobi peut être utilisée pour calculer une approximation de la solution d'une équation différentielle partielle. La solution est discrétisée sur une grille. À chaque pas de temps, l'approximation courante est mise à jour en calculant, pour chaque point de la grille, la moyenne pondérée des valeurs du point et de ses quatre voisins. On considère ici le cœur de cet algorithme qui consiste en deux nids de boucles opérant sur deux tableaux bi-dimensionnels A et B. Le premier nid de boucles calcule dans A l'approximation courante à partir des valeurs stockées dans le tableau B qui représente la dernière approximation. Le second nid de boucles transfère les éléments de A dans B.

Le programme PANDORE est identique au programme séquentiel mis à part la spécification de distribution (voir figure 4.1). Les tableaux A et B sont tous les deux distribués en groupes de lignes (blocs de taille $N/P \times N$) avec un groupe par processeur. L'attribution des blocs aux processeurs (fonction `map`) n'est pas significative puisqu'il n'y a qu'un seul bloc par processeur. Avec cette distribution, la charge de calcul est répartie équitablement sur les processeurs et le deuxième nid de boucles s'exécute sans communication puisque A et B sont exactement alignés. Des communications sont nécessaires dans le premier nid de boucles. En effet, le calcul des éléments situés sur la frontière de chaque bloc nécessite l'accès à un élément possédé par un processeur voisin. La symétrie des accès aurait permis une distribution par groupe de colonnes qui eût entraîné le même coût de communication. Cependant, comme on accède aux éléments d'abord le long des lignes (la boucle sur i est externe), la localité des accès est mieux exploitée par une distribution par lignes[1].

```
dist jacobi(double B[N][N] by block(N/P,N) map regular(0,1) mode INOUT)
            double A[N][N] by block(N/P,N) map regular(0,1);
{
 int i,j;

 for (i=1; i<N-1; i++)
   for (j=1; j<N-1; j++)
     A[i][j] = V*B[i][j] + W*(B[i-1][j]+B[i+1][j]+B[i][j-1]+B[i][j+1]);

 for (i=1; i<(N-1); i++)
   for (j=1; j<(N-1); j++)
     B[i][j] = A[i][j];
}
```

FIG. 4.1. – *Code* PANDORE *pour le Jacobi*

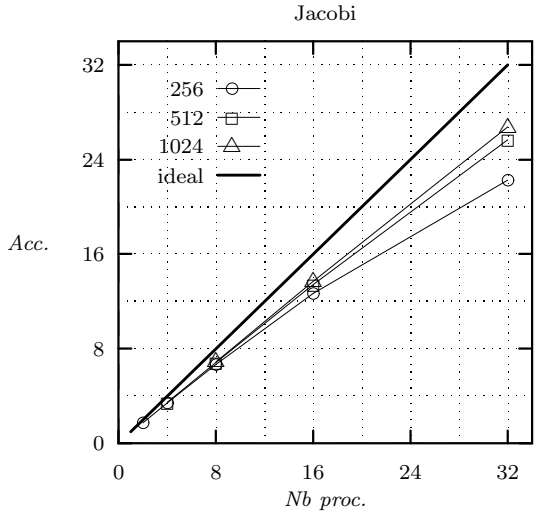
Ce code est entièrement optimisé par le compilateur PANDORE. En particulier :

- Les domaines d'itérations sont statiquement restreints.
- Toutes les communications sont directes (2 messages par processeur).
- La conversion d'indice est réduite à l'identité car une seule dimension est distribuée (le numéro de page est le numéro de la ligne et l'offset dans la page est le numéro de colonne).

Les performances sont présentées dans la figure 4.2. On observe que l'efficacité est très correcte même pour de petites tailles de tableaux et un nombre relativement élevé de processeurs : on obtient par exemple 70% d'efficacité pour $N = 256$ sur 32 processeurs, cas dans lequel un quart des éléments sont communiqués. On observe un plafond de l'efficacité d'environ 86%. Ceci est dû à l'incapacité du compilateur

1. Le schéma de compilation optimisé ne perturbe pas l'ordre des boucles du source.

cible à optimiser totalement la boucle de calcul générée (l'accès aux éléments de tableaux via la table des pages perturbe l'allocation de registres) alors qu'il le fait sur le code séquentiel. Une optimisation manuelle de cette allocation de registre montre qu'une efficacité de plus de 98% peut être atteinte.



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
2	0.970	1.7	87%
4	0.495	3.4	86%
8	0.253	6.7	84%
16	0.132	12.8	80%
32	0.075	22.4	70%
$N = 512$			
2	—	—	—
4	1.961	3.4	86%
8	0.999	6.8	85%
16	0.503	13.4	84%
32	0.261	25.9	81%
$N = 1024$			
2	—	—	—
4	—	—	—
8	3.931	6.9	86%
16	1.976	13.7	86%
32	1.013	26.7	84%

FIG. 4.2. – *Performances du Jacobi sur POM-iPSC/2*

Red-Black

L'algorithme de relaxation Red-black met en œuvre une variante de méthode de résolution itérative d'équation différentielle partielle travaillant sur une seule grille. Les points sont partagés en deux catégories (les rouges et les noirs) qui alternent sur la grille. Le calcul des points est divisé en deux étapes : chaque point rouge est d'abord calculé en fonction des quatre points noirs voisins puis les points noirs sont calculés en fonction des points rouges de façon identique.

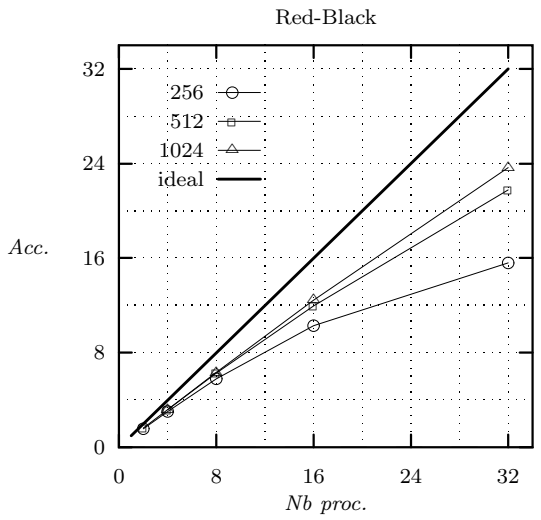
Le code du programme PANDORE est montré sur la figure 4.3. Il comporte quatre nids de boucles, les points de chaque couleur étant calculés par deux nids de boucles successifs. Afin d'obtenir des boucles normalisées, les pas ont été ramenés à 1 et les accès aux tableaux modifiés en conséquence. Le tableau A est distribué en groupe de N/P lignes ; comme pour le Jacobi, la fonction `map` n'est pas significative ici. Cette distribution répartit uniformément la charge de calcul. Des communications sont nécessaires à chaque nid de boucle pour satisfaire les accès aux éléments jouxtant les frontières de bloc : la moitié des éléments de la ligne nord ou de la ligne sud sont à échanger.

Tous les nids de boucles sont optimisés, les domaines de calculs sont restreints et les communications vectorisées. Pour chaque nid, des communications directes sont effectuées mais au prix du transfert d'éléments superflus. En effet, une ligne entière

est communiquée alors que la moitié de ses éléments sont utilisés. Ceci explique la baisse de performances par rapport au Jacobi (voir figure 4.4).

```
dist RedBlack(double A[N][N] by block(N/P,N) map regular(0,1) mode INOUT)
{
 int i,j ;

 for (i=0 ; i<(N-1)/2 ; i++)
   for (j=0 ; j<(N-1)/2 ; j++)
     A[2*i+1][2*j+1] = (1-W) * A[2*i+1][2*j+1] + (W/4) *
         (A[2*i][2*j+1] + A[2*i+2][2*j+1] + A[2*i+1][2*j] + A[2*i+1][2*j+2]) ;

 for (i=1 ; i<(N-1)/2+1 ; i++)
   for (j=1 ; j<(N-1)/2+1 ; j++)
     A[2*i][2*j] = (1-W) * A[2*i][2*j] + (W/4) *
         (A[2*i-1][2*j] + A[2*i+1][2*j] + A[2*i][2*j-1] + A[2*i][2*j+1]) ;

 for (i=1 ; i<(N-1)/2+1 ; i++)
   for (j=0 ; j<(N-1)/2 ; j++)
     A[2*i][2*j+1] = (1-W) * A[2*i][2*j+1] + (W/4) *
         (A[2*i-1][2*j+1] + A[2*i+1][2*j+1] + A[2*i][2*j] + A[2*i][2*j+2]) ;

 for (i=0 ; i<(N-1)/2 ; i++)
   for (j=1 ; j<(N-1)/2+1 ; j++)
     A[2*i+1][2*j] = (1-W) * A[2*i+1][2*j] + (W/4) *
         (A[2*i][2*j] + A[2*i+2][2*j] + A[2*i+1][2*j-1] + A[2*i+1][2*j+1]) ;
}
```

FIG. 4.3. – *Code* PANDORE *pour le Red-Black*



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
2	0.965	1.6	78%
4	0.501	3.0	76%
8	0.261	5.8	72%
16	0.147	10.3	64%
32	0.097	15.6	48%
$N = 512$			
2	3.831	1.6	81%
4	1.942	3.2	80%
8	0.992	6.3	78%
16	0.520	12.0	75%
32	0.286	21.7	68%
$N = 1024$			
2	—	—	—
4	7.709	3.2	80%
8	3.903	6.3	79%
16	1.992	12.4	78%
32	1.046	23.7	74%

FIG. 4.4. – *Performances du Red-Black sur POM-iPSC/2*

Produit de matrices

Nous utilisons l'algorithme usuel du produit de matrices opérant par produits scalaires [52, page 11] pour calculer le produit $C = AB$ où A, B et C sont des matrices carrées de taille $N \times N$. Le code de la phase distribuée PANDORE est donné dans la figure 4.5. Le programme met d'abord la matrice C à 0 avant d'accumuler les produits scalaires. Les opérandes et le résultat sont tous distribués : A et C sont distribuées par groupes de lignes alors que B est distribuée par groupe de colonnes. Ceci permet d'avoir un équilibrage de charge optimum. Les données nécessaires au calcul d'un élément $C[i][j]$ n'étant pas toutes locales (une partie de la colonne j de B est distante), des communications sont nécessaires.

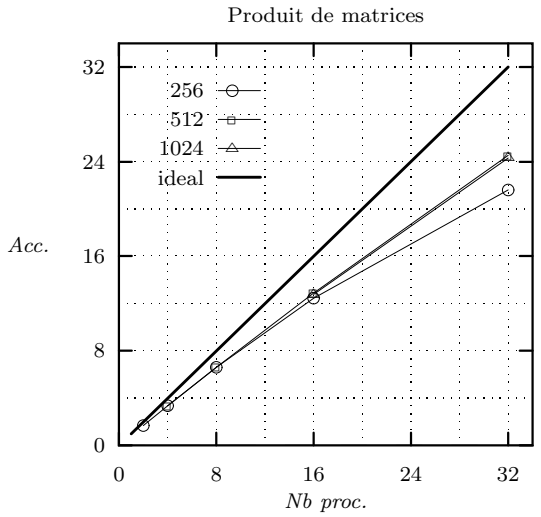
```
dist matprod(double A[N][N] by block(N/P,N) map regular(0,1) mode IN,
             double B[N][N] by block(N,N/P) map regular(0,1) mode IN,
             double C[N][N] by block(N/P,N) map regular(0,1) mode OUT
           )
{
 int i,j,k;
 double colj[N];

 — Mise à 0 de C —
 for (i=0; i<N; i++)
   for (j=0; j<N; j++)
     C[i][j] = 0.0;

 for (j=0; j<N; j++) {

   — Copie de la colonne j de B —
   for (k=0; k<N; k++)
     colj[k] = B[k][j];

   — Calcul de la colonne j de C —
   for (i=0; i<N; i++)
     for (k=0; k<N; k++)
       C[i][j] = C[i][j] + A[i][k]*colj[k];
 }
}
```

FIG. 4.5. – *Code* PANDORE *pour le produit de matrices*

Le calcul de C est organisé en une boucle séquentielle sur les colonnes. Pour chaque colonne j, on commence par copier dans le vecteur $colj$ la colonne j de B puis on fait les N produits scalaires permettant le calcul de la colonne j de C. Comme le vecteur $colj$ est dupliqué, la boucle de copie de la colonne permet d'effectuer les communications nécessaires par des diffusions plutôt que par des messages point à point. Tous les produits scalaires sont ensuite calculés entièrement localement car les tableaux A et C sont alignés.

Le schéma de compilation optimisé s'applique pour toutes les boucles sauf la boucle sur j qui reste séquentielle. Pour ces boucles, les domaines de calcul sont restreints. Les diffusions intervenant lors de la boucle d'affectation de $colj$ sont effectuées par communications directes.

Les performances obtenues (voir figure 4.6) sont bonnes malgré le volume important de communications (la matrice B est par l'intermédiaire du vecteur $colj$ globalement entièrement diffusée). On peut également noter que la baisse de performance est minime lorsque l'on augmente le nombre de processeurs, même pour les matrices les plus petites.



P	*temps (sec)*	*accélération*	*efficacité*
	$N = 256$		
2	120.284	1.7	85%
4	60.818	3.3	84%
8	30.786	6.6	83%
16	16.330	12.4	78%
32	9.418	21.6	67%
	$N = 512$		
2	—	—	—
4	481.113	3.3	84%
8	243.339	6.6	83%
16	124.976	12.9	80%
32	65.469	24.5	77%
	$N = 1024$		
2	—	—	—
4	—	—	—
8	—	—	—
16	1074.786	12.1	76%
32	551.037	23.6	74%

FIG. 4.6. – *Performances du produit de matrices sur POM-iPSC/2*

Factorisation LU

La factorisation LU permet de décomposer une matrice A en un produit de deux matrices, l'une triangulaire inférieure (L) et l'autre triangulaire supérieure (U). Ceci permet de transformer la résolution d'un système linéaire $Ax = b$ en la résolution de deux systèmes triangulaires plus simples : $Ly = b$ et $Ux = y$.

Nous utilisons la version kji de l'algorithme d'élimination de Gauss [52, page 98]. Le programme PANDORE est présenté dans la figure 4.7. Il consiste en une boucle sur k parcourant les éléments diagonaux. Pour chaque k, on calcule le vecteur de Gauss en modifiant la fin de la colonne k, on transforme ensuite la sous-matrice de coin supérieur gauche $(k+1, k+1)$. Chaque élément de cette sous-matrice est calculé en fonction des éléments correspondants dans la ligne k et la colonne k.

Une décomposition par lignes est choisie pour A afin de répartir équitablement la charge à la fois pour la boucle de calcul du vecteur de Gauss et pour le nid de

boucles appliquant la transformation de la sous-matrice. Un attribution cyclique des lignes est nécessaire pour répartir la charge.

Comme pour le produit de matrice, on introduit une boucle de copie de la fin de la ligne k dans une variable dupliquée (*gvec*) pour que des diffusions soient utilisées.

La boucle sur k, qui comporte des dépendances, reste séquentielle. Le schéma de base est appliqué à l'affectation du pivot, en revanche, tous les nids de boucles internes à la boucle sur j sont optimisés par le compilateur : les domaines de calculs sont restreints et les diffusions vectorisées. Le mécanisme de transfert de segments s'avère être ici utile car seule une partie de la ligne k doit être diffusée.

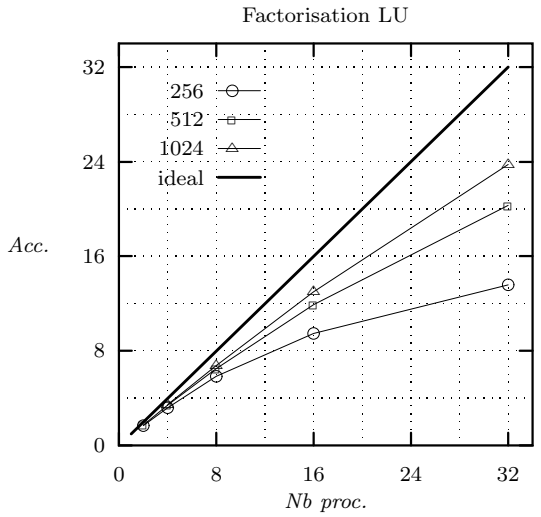
```
dist LU(double A[N][N] by block(1,N) map wrapped(1,0) mode INOUT)
{
 int i,j,k;
 double gvec[N], pivot;

 for (k=0; k<(N-1); k++) {
   — calcul du vecteur de Gauss —
   pivot = A[k][k];
   for (i=k+1; i<N; i++)
     A[i][k] = A[i][k] / pivot;
   — copie de la fin de la ligne k (diffusion) —
   for (j=k+1; j<N; j++)
     gvec[j] = A[k][j];
   — application de la transformation de Gauss
        à la sous-matrice d'origine (k+1, k+1)
   for (i=k+1; i<N; i++)
     for (j=k+1; j<N; j++)
       A[i][j] = A[i][j] - A[i][k] * gvec[j];
 }
}
```

FIG. 4.7. – *Code* PANDORE *pour la factorisation LU*

On constate sur la figure 4.8 un fléchissement des courbes d'accélération plus important que dans le cas du produit de matrices. On peut en donner deux explications :

- La distribution cyclique induisant un nombre assez important de blocs, elle entraîne un coût plus important lors du parcours restreint du domaine de calcul (pendant l'*Exec*) ainsi que lors du calcul de l'ensemble d'éléments à communiquer (pendant le *Refresh*) car ceux-ci sont organisés selon un parcours des blocs. On constate toutefois que ce coût n'est pas prohibitif.
- La quantité de calcul à effectuer diminue lorsque k augmente. Le coût de parallélisation, qui comporte une partie fixe, prend donc une part relative de

plus en plus importante. Lorsque k est proche de N, le coût de parallélisation devient même plus important que le coût de calcul lui-même.



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
2	43.628	1.7	84%
4	22.869	3.2	80%
8	12.535	5.9	73%
16	7.752	9.5	59%
32	5.412	13.6	42%
$N = 512$			
2	339.429	1.7	87%
4	173.693	3.4	85%
8	90.948	6.5	81%
16	49.545	11.9	75%
32	29.117	20.3	63%
$N = 1024$			
2	—	—	—
4	1358.917	3.5	87%
8	694.665	6.8	85%
16	362.944	13.0	81%
32	198.349	23.8	74%

FIG. 4.8. – *Performances de la factorisation LU sur POM-iPSC/2*

Orthonormalisation de Gram-Schmidt

Étant donné un ensemble de vecteurs de $\mathbb{R}^n$, l'algorithme de Gram-Schmidt modifié produit une base orthonormale de l'espace généré par ces vecteurs [52, page 219]. La base est construite pas à pas, chaque vecteur nouvellement calculé remplaçant l'ancien. Le code PANDORE est donné dans la figure 4.9. L'ensemble des vecteurs est stocké dans un tableau bi-dimensionnel v. L'algorithme est constitué d'une boucle principale sur les lignes de v ; à chaque ligne i, on normalise le vecteur ligne i puis on corrige tous les vecteurs lignes j tels que $j > i$. La correction du vecteur j se fait en calculant le produit scalaire du vecteur i et du vecteur j.

Le tableau v est décomposé en lignes attribuées cycliquement aux processeurs afin d'équilibrer la charge correspondant à la boucle de correction. Pour pouvoir distribuer le calcul des normes et celui des produits scalaires, on utilise deux tableaux mono-dimensionnels *xnorm* et *sdot* alignés sur v (on a expansé les scalaires utilisés dans la version séquentielle du programme). La ligne i est copiée dans une variable dupliquée *vc* pour utiliser la diffusion et rendre complètement locale la correction. Cette dernière est composée d'une succession de trois nids de boucles parallèles comportant une instruction. Ces boucles sont le résultat de la distribution d'une unique boucle sur j afin de permettre l'application du schéma de compilation optimisé sur des domaines 2D.

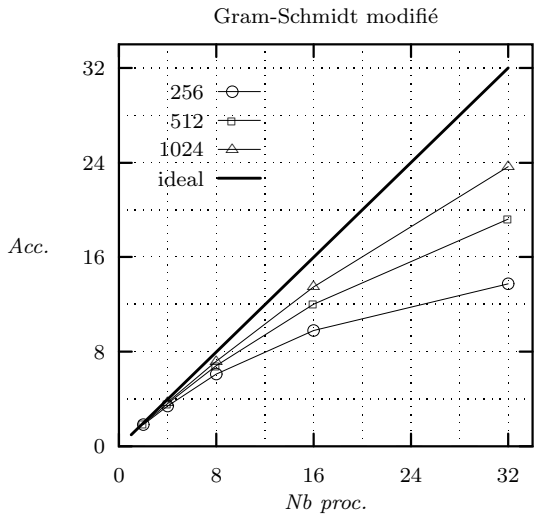
Le compilateur applique donc le schéma optimisé à tous les nids de boucles internes à la boucle sur i, celle-ci restant séquentielle. On peur noter le fait que seule

la correction est parallélisée ; en particulier, pour un i donné, toute la normalisation est effectuée par un seul processeur puisqu'elle n'opère que sur une seule ligne.

```
dist MGS(double v[N][N] by block(1,N) map wrapped(0,1) mode INOUT)
   double xnorm[N] by block(1) map wrapped(0);
   double sdot[N] by block(1) map wrapped(0);
{
 int i,j,k;
 double vc[N];

 for (i=0;i<N;i++) {
   — normalisation —
   xnorm[i] = 0.0;
   for (k=0;k<N;k++)
     xnorm[i] = xnorm[i] + v[i][k]*v[i][k];
   xnorm[i] = 1.0/sqrt(xnorm[i]);

   for (k=0;k<N;k++)
     v[i][k]=v[i][k]*xnorm[i];

   — copie de la ligne i (diffusion) —
   for (k=0;k<N;k++)
     vc[k]=v[i][k];

   — correction —
   for (j=i+1;j<N;j++)
     sdot[j]=0.0;

   for (j=i+1;j<N;j++)
     for (k=0;k<N;k++)
       sdot[j]=sdot[j]+vc[k]*v[j][k];

   for (j=i+1;j<N;j++)
     for (k=0;k<N;k++)
       v[j][k]=v[j][k] - sdot[j]*vc[k];
 }
}
```

FIG. 4.9. – *Code* PANDORE *pour l'algorithme de Gram-Schmidt modifié*

Les performances sont similaires à celles de la factorisation LU. On note toutefois que les courbes d'accélération démarrent à un plus haut niveau et baissent plus rapidement quand on augmente le nombre de processeurs. Ceci peut s'expliquer par le fait que la diminution de la quantité de calcul à effectuer est ici plus faible lorsque l'on avance dans la boucle séquentielle principale, mais ceci est compensé par l'absence de parallélisation de la normalisation qui devient relativement coûteuse quand on augmente le nombre de processeurs.



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
2	122.654	1.8	92%
4	65.428	3.4	86%
8	36.812	6.1	76%
16	22.955	9.8	61%
32	16.365	13.7	43%
$N = 512$			
2	957.302	1.9	94%
4	495.147	3.6	91%
8	263.660	6.8	85%
16	149.208	12.0	75%
32	937.495	19.2	60%
$N = 1024$			
2	—	—	—
4	3863.675	3.7	93%
8	1998.343	7.2	90%
16	1064.103	13.5	84%
32	607.173	23.7	74%

FIG. 4.10. – *Performances du Gram-Schmidt modifié sur POM-iPSC/2*

Daxpy

Étant donnés deux vecteurs x et y et un scalaire α, l'algorithme Daxpy calcule le vecteur $z = \alpha x + y$, les réels manipulés étant en double précision [78]. Dans le programme utilisé (voir figure 4.11), le résultat est écrit dans y.

Les tableaux X et Y sont distribués en P blocs pour une exécution sur P processeurs. Les calculs sont donc entièrement locaux, aucune communication n'est nécessaire.

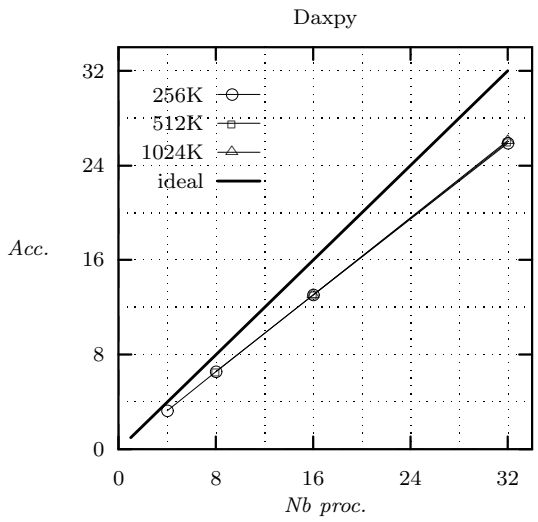
```
dist daxpy(double alpha mode IN,
           double X[N] by block(N/P) map regular(0) mode IN,
           double Y[N] by block(N/P) map regular(0) mode INOUT)
{
 int i ;

 for (i=0 ; i<N ; i++)
   Y[i] = alpha * X[i] + Y[i] ;
}
```

FIG. 4.11. – *Code* PANDORE *pour le Daxpy*

On observe que les courbes d'accélération sont linéaires quelque soit la taille des données. Ceci s'explique par le fait que le schéma de compilation optimisé qui est ici appliqué, n'engendre effectivement aucune communication et restreint les domaines de calculs de façon optimale.

L'efficacité plafonne autour de 80% alors qu'on pourrait attendre une efficacité quasi-optimale. Ceci est dû à l'accès aux éléments de tableaux. Dans le programme

séquentiel, il s'agit d'un accès à un tableau mono-dimensionnel alors que dans la version distribuée, on utilise l'adressage paginé qui comporte l'équivalent d'un accès à un tableau 2D. Les différences d'optimisations appliquées dans chaque cas par le compilateur cible — discutées dans le cas du Jacobi — creusent l'écart entre les deux types d'accès. On notera toutefois que cet écart reste acceptable.



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256K$			
2	—	—	—
4	1.095	3.3	82%
8	0.548	6.5	82%
16	0.274	13.0	81%
32	0.138	25.9	81%
$N = 512K$			
2	—	—	—
4	—	—	—
8	1.096	6.5	82%
16	0.548	13.0	82%
32	0.275	26.0	81%
$N = 1024K$			
2	—	—	—
4	—	—	—
8	—	—	—
16	1.097	13.0	82%
32	0.550	26.1	82%

FIG. 4.12. – *Performances du Daxpy sur POM-iPSC/2*

4.2.1.2 Application de propagation d'ondes

Application de propagation d'ondes

Nous avons utilisé l'environnement PANDORE pour étudier la parallélisation d'une application de propagation d'ondes développée par l'Institut Français des Pétroles. Le cœur du programme est d'une longueur d'environ un millier de lignes, il simule la propagation d'une onde dans un milieu bi-dimensionnel borné. Les ondes sont générées par une explosion déclenchée en un point donné de l'espace considéré. Le programme étudie l'évolution temporelle des ondes en plusieurs points où l'on a placé des capteurs. Il prend en entrée plusieurs paramètres de simulation tels que les pas de temps et d'espace, la fréquence de la source d'explosion et la position des capteurs.

L'évolution temporelle de la propagation des ondes est régie par un schéma numérique dit « à différences finies ». Ce schéma correspond à une discrétisation du second ordre en temps ($V(t = n + 1) = F(V(t = n), V(t = n - 1))$) et du second ordre des dérivées partielles spatiales du système continu d'équations aux dérivées partielles décrivant la propagation des ondes élastiques en milieu hétérogène.

Les résultats fournis par l'algorithme sont les valeurs des champs de déplacements horizontaux et verticaux à chaque pas de temps. Celles-ci sont récupérées au niveau de chaque capteur. Le programme se décompose en deux phases :

- une phase d'initialisation dans laquelle sont définies les conditions initiales de l'explosion ainsi que les contraintes liées au milieu de propagation ;
- une phase de calcul qui consiste en une boucle principale sur le temps. À chaque pas de temps, les mouvements horizontaux et verticaux sont calculés pour chaque point de la grille représentant l'espace bi-dimensionnel. Un échantillonage spatial de la grille est effectué pour stocker certaines valeurs des mouvements.

On note U^t (resp. W^t) les mouvements horizontaux (resp. verticaux) au temps t. Les dépendances temporelles sont exprimées par :

$$(U^{t+1}, W^{t+1}) = f(U^t, U^{t-1}, W^t, W^{t-1})$$

La boucle sur le temps est parcourue par pas de deux, on dispose de quatre tableaux bi-dimensionnels afin de mémoriser les mouvements à $t-1$ et t nécessaires au calcul des mouvements au temps $t+1$: U_m, U_p et W_m, W_p. Le corps de la boucle effectue la suite d'opérations suivante :

$$(U_p, W_p) = f(U_m, U_p, W_m, W_p)$$
$$(U_m, W_m) = f(U_p, U_m, W_p, W_m)$$

La fonction f comprend deux séries de nids de boucles :

- La première série est constituée de nids boucles de profondeur 2 opérant sur l'intérieur de la grille. Ces boucles sont comparables à celles de la première partie du Jacobi présenté page 4.2.1.1. Elles sont de la forme :

  pour i **de** 1 **à** $n-2$
  pour j de 1 à $n-2$
  $U_p[i][j] = f_1(U_p[i][j],$
  $U_m[i][j], U_m[i-1][j], U_m[i+1][j], U_m[i-1][j+1], U_m[i+1][j],$
  $W_m[i][j], W_m[i-1][j], W_m[i][j-1], W_m[i-1][j-1])$
  fpour
  fpour

- La deuxième est une série de boucles de profondeur 1 agissant sur les bords supérieurs des tableaux. Elles sont de la forme :

  pour j **de** 0 **à** $n-1$
  $U_p[0][j] = f_2(U_m[0][j])$
  fpour

Distribution du programme

Nous décrivons maintenant les différentes étapes de la transformation du code séquentiel initial en un programme PANDORE. Une seule phase distribuée est nécessaire pour cette application, elle correspond à la partie effectuant l'initialisation suivie de la partie de calcul. Il n'y a pas lieu de faire un découpage en plusieurs phases distribuées car il n'est pas nécessaire d'utiliser des distributions différentes à chaque partie. Seules les entrées-sorties sont placées hors des phases distribuées, ce qui fait que tous les calculs effectués par l'application se voient distribués sur les nœuds de la plate-forme parallèle.

Le corps de la phase distribué a été légèrement modifié de façon à faire apparaître des nids de boucles conformes aux conditions sous lesquelles le compilateur utilise le schéma optimisé. Ceci est obtenu facilement pour les parties de calcul principales décrites au paragraphe précédent car il s'agit de nids de boucles parallèles où les bornes et les références aux tableaux sont affines.

La tâche principale réside dans le choix des distributions de tableaux. Les tableaux principaux sont des tableaux 2D représentant l'espace de propagation. Ils sont utilisés conjointement avec une dizaine de tableaux temporaires 2D du même type. Huit autres tableaux 1D servent au calcul sur le bord de la grille. Nous avons opté pour une distribution en groupes de colonnes pour tous les tableaux 2D car,

- dans le calcul des valeurs associées à la partie intérieure de la grille, les dépendances sont similaires à celles du Jacobi où la distribution en groupes de colonnes et la distribution en groupes de lignes sont les plus efficaces ;

- les boucles de profondeur 1 opérant sur le bord supérieur de la grille nécessitent une distribution par colonnes pour équilibrer la charge de calcul ;

- les nids de boucles de profondeur 2 parcourent les tableaux dans le sens des colonnes, un découpage en colonnes renforce donc la localité d'accès pour la plus grande partie du calcul.

Nous choisissons donc, pour P processeurs, un partitionnement en blocs de taille $(N/P, N)$ de tous les tableaux de taille $N \times N$ et un partitionnement en blocs de taille N/P pour les tableaux mono-dimensionnels de taille N. Il est à noter que le programme PANDORE obtenu est très peu différent du programme séquentiel original.

Résultats

Le programme a été compilé et exécuté sur l'iPSC/2 pour plusieurs tailles de tableaux ($N = 128$, 256 et 512). Le temps de compilation (sur une station Sun SS10) s'est élevé à une dizaine de minutes (environ quarante nids de boucles ont été optimisés), le code exécutable obtenu avait une taille d'environ 1 Mo.

Les performances obtenues sont présentées dans la table 4.1 (les calculs sont en double précision). On peut remarquer que les performances restent à un niveau acceptable même pour les petites tailles de tableaux. Ceci est important car le nombre de tableaux mis en jeu dans l'application fait en sorte que l'on manipule un grand volume de données même lorsque les tailles de tableaux sont petites (20,5 Mo pour $N = 512$ par exemple).

$N = 128$		
P	*temps (sec)*	*efficacité*
2	136	75%
4	71	72%
8	38	66%
16	22	56%
32	17	38%

$N = 256$		
P	*temps (sec)*	*efficacité*
2	—	—
4	280	71%
8	145	69%
16	77	65%
32	42	59%

$N = 512$		
P	*temps (sec)*	*efficacité*
2	—	—
4	—	—
8	—	—
16	289	68%
32	150	66%

TAB. 4.1. – *Performances de l'application de propagation d'ondes sur POM-iPSC/2*

Optimisations supplémentaires

Dans le programme PANDORE obtenu par la distribution du programme séquentiel, les boucles parallèles qui effectuent l'échantillonage de la grille pour la constitution du résultat entraînent des communications qui pourraient être évitées. Ceci est seulement dû au fait que l'alignement ne peut pas être exprimé dans le langage PANDORE ; les tableaux stockant les mouvements calculés au niveau des capteurs ne sont pas alignés avec les tableaux représentant la grille. Cependant, une renumérotation manuelle triviale des points des capteurs est possible pour réaliser l'alignement. Après cette transformation, les boucles d'échantillonage s'exécutent sans communication. Les performances globales ne sont pas beaucoup améliorées car l'échantillonage ne représente qu'environ 5% du temps total d'exécution.

Pendant l'exécution, certains éléments sont envoyés plusieurs fois au même processeur sans avoir été modifiés. Ces transferts redondants peuvent être évités en modifiant le source PANDORE afin d'utiliser des tableaux auxiliaires stockant ces éléments après la première réception. Nous avons expérimenté cette optimisation, qui s'est révélée difficile à mettre en œuvre et qui n'améliore le temps d'exécution global que d'environ 5%. Ce gain n'apparaît pas significatif, étant donné le coût mémoire induit, dans une application où cette ressource est en fait critique.

4.2.2 Expérimentation sur machine à MVP

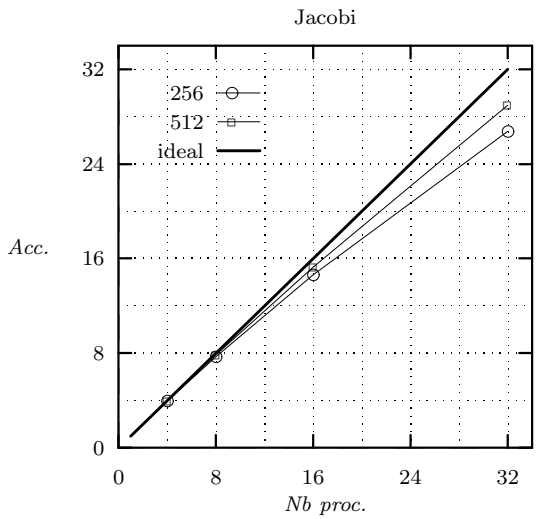
Les noyaux présentés lors de l'étude de performances sur machine à messages ont été utilisés pour expérimenter le compilateur PANDORE générant du code pour la MVP Koan. Seules les distributions laissant une dimension non distribuée sont prises en compte dans la version actuellement implantée. La version 2.1 de Koan a été utilisée sur un hypercube Intel iPSC/2 de 32 nœuds avec 4 Mo de mémoire par nœud.

Nous présentons ici les performances obtenues pour quatre d'entre eux : le Jacobi, le Red-Black, le produit de matrices et la factorisation LU (figures 4.13 à 4.16).

Les performances sont globalement similaires à celles obtenus sur la POM, ce qui s'explique par la régularité des algorithmes et la relative adéquation entre les distributions employées et la taille de page de la MVP Koan (4 Ko). Des expériences plus complètes seraient nécessaires pour comparer les performances des deux approches pour des programmes et des tailles de données plus variés.

Les programmes mettant en jeu des communications point à point (Jacobi et Red-Black) exhibent des performances légèrement supérieures avec la MVP : les communications sont identiques et l'accès aux données est plus rapide. En ce qui concernent les programmes induisant des diffusions, on constate que les performances se dégradent plus fortement lorsque l'on augmente le nombre de processeurs, la dégradation étant très nette pour les tailles de données les plus petites.

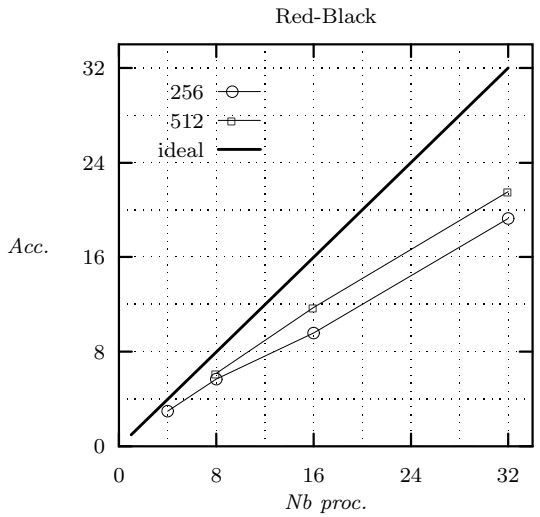
Bien que cela n'apparaisse pas explicitement dans les courbes présentées, le volume de mémoire requis à l'exécution pour les programmes PANDORE tournant sur Koan est plus important que pour les programmes tournant sur la POM[2]. Ceci est dû à l'impossibilité de gérer très finement l'allocation mémoire dans la version sur Koan, notamment en ce qui concerne l'attribution des pages pour les partitions locales.



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
2	—	—	—
4	0.43	4.0	99%
8	0.22	7.7	96%
16	0.12	14.6	92%
32	0.06	26.7	84%
$N = 512$			
2	—	—	—
4	1.69	4.0	99%
8	0.86	7.8	98%
16	0.44	15.3	96%
32	0.23	29.0	91%

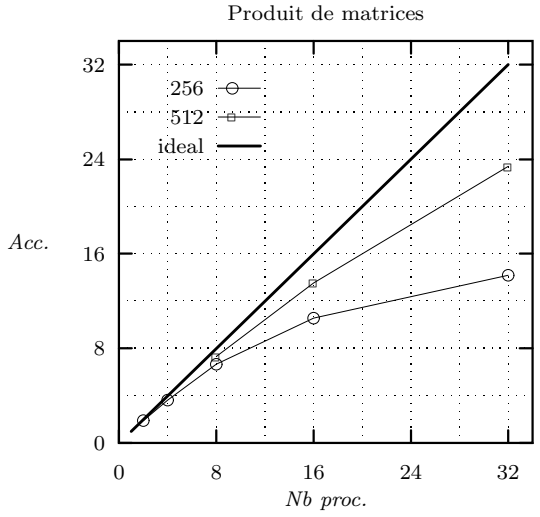
FIG. 4.13. – *Performances du Jacobi sur Koan-iPSC/2*

2. On peut remarquer que, pour les plus grandes tailles de tableaux, plusieurs exécutions sur un petit nombre de processeurs n'ont pas pu être effectuées.



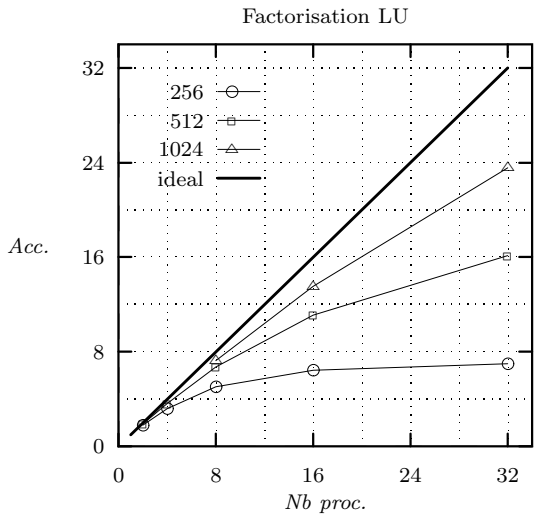
P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
2	—	—	—
4	0.50	3.0	75%
8	0.27	5.7	71%
16	0.16	9.6	60%
32	0.08	19.3	60%
$N = 512$			
2	—	—	—
4	—	—	—
8	1.01	6.2	77%
16	0.53	11.8	74%
32	0.29	21.5	67%

FIG. 4.14. – *Performances du Red-Black sur Koan-iPSC/2*



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
2	107.19	1.9	95%
4	56.16	3.6	91%
8	30.776	6.6	83%
16	19.45	10.4	65%
32	14.92	14.3	45%
$N = 512$			
2	—	—	—
4	—	—	—
8	219.21	7.3	92%
16	118.53	13.6	85%
32	68.65	23.4	73%

FIG. 4.15. – *Performances du produit de matrices sur Koan-iPSC/2*



P	*temps (sec)*	*accélération*	*efficacité*
$N = 256$			
2	40.95	1.8	90%
4	23.03	3.2	80%
8	14.52	5.1	63%
16	11.42	6.4	40%
32	10.48	7.0	22%
$N = 512$			
2	313.11	1.9	94%
4	162.88	3.6	91%
8	88.23	6.7	84%
16	53.13	11.1	70%
32	36.63	16.1	50%
$N = 1024$			
2	—	—	—
4	—	—	—
8	648.57	7.3	91%
16	348.17	13.5	85%
32	200.34	23.5	74%

FIG. 4.16. – *Performances de la factorisation LU sur Koan-iPSC/2*

4.3 Évaluation de performance et compilation par distribution de données

4.3.1 Performances des programmes parallèles compilés par distribution de données

Les programmes générés par les systèmes de compilation par distribution de données tels que PANDORE, sont des programmes distribués, dont les performances sont difficiles à évaluer précisément. Le fait qu'ils ne soient pas écrits directement par le programmeur mais générés automatiquement par un compilateur ajoute à la difficulté d'évaluation des performances. Si le programmeur peut influer sur la performance du programme généré, il n'en maîtrise pas tous les aspects, le compilateur et l'exécutif jouant un rôle important.

4.3.1.1 Influence du programmeur

Le principal axe selon lequel le programmeur peut intervenir sur la performance de son programme est le choix de la distribution des variables. En effet, comme on l'a vu dans la description des expérimentations effectuées avec le compilateur PANDORE, les performances du programme dépendent de l'adéquation de la distribution des données avec l'algorithme utilisé dans la mesure où celle-ci influe sur la répartition de charge et les communications.

- C'est la distribution des variables qui guide le compilateur pour la distribution du code, puisque celui-ci applique la règle des écritures locales. Le pro-

grammeur indique donc implicitement, pour un code donné, comment seront réparties les instructions et donc comment la charge de calcul sera répartie sur les processeurs. Une charge mal équilibrée (répartie inéquitablement sur l'ensemble des processeurs) baissera les performances car le temps d'exécution du programme parallèle est le plus grand des temps mesuré sur chacun des processeurs.

- La distribution des variables détermine également la localisation des données dans les mémoires associées aux processeurs. Pour un code donné, le programmeur indique donc implicitement si les accès aux variables qu'il a spécifiés sont locaux ou distants. Dans le cas d'accès distants, des communications seront nécessaires, communications qui ralentiront l'exécution du programme.

Il faut également considérer le fait que le programmeur dispose parfois d'une certaine latitude dans l'expression du code lui-même. Ceci est le cas quand il part d'un problème dont la solution peut s'exprimer sous la forme de plusieurs algorithmes séquentiels (*e.g.* les noyaux d'applications d'algèbre linéaire que nous avons présentés au début de ce chapitre), le choix de l'algorithme a évidemment des conséquences sur la nature des accès aux données et donc sur la répartition de charge et les communications.

4.3.1.2 Influence du compilateur et de l'exécutif

Idéalement, le temps d'exécution d'un programme distribué SPMD appliquant la règle des écritures locales pourrait être défini en tenant compte uniquement de la répartition de charge et du nombre d'accès distants, en considérant, pour chaque processeur, le temps correspondant à la charge de calcul (accès mémoire compris) auquel on ajoute un surcoût dépendant du nombre d'accès distants. Ce temps serait caractéristique du programme compilé par distribution des données, indépendamment du compilateur utilisé.

Dans la pratique, le compilateur et l'exécutif rajoutent un certain nombre de surcoûts dus notamment :

- aux transformations d'adresses ;
- aux gardes de possession de données ;
- aux communications superflues ;
- aux synchronisations induites par les communications.

Ces surcoûts sont inhérents à l'approche de la compilation par distribution de données. Cependant, comme on l'a vu précédemment, le compilateur et l'exécutif peuvent effectuer plusieurs optimisations afin de les réduire (accès optimisé aux données, restrictions de domaines, vectorisation des communications, recouvrement calcul/communications,...). Ces surcoûts prennent donc des valeurs variables selon les systèmes de compilation et pour un système donné, selon les stratégies adoptées.

On peut remarquer que la connaissance des stratégies d'optimisation du compilateur et de l'exécutif peuvent aider le programmeur dans l'écriture de son programme. En transformant quand c'est possible son code ou la distribution des données pour se ramener aux cas optimisables, le programmeur obtiendra certainement de meilleures performances. Toutefois, on ne peut envisager ce genre de transformations que dans des cas simples, tels que ceux qui ont été décrits dans l'étude des expérimentations en PANDORE dans le paragraphe 4.2. Un exemple typique est celui du produit de matrices dans lequel on a simplement introduit un vecteur supplémentaire pour éviter la duplication d'une des matrices.

La plupart des prototypes de système de compilation par distribution de données produisent des codes sources qui sont compilés eux-mêmes par un compilateur cible. Ces compilateurs cibles offrent fréquemment une variété d'optimisations telles que le réordonnancement d'instructions, l'élimination de variables d'induction et d'expressions communes, la propagation de constantes ou la vectorisation automatique [1, 102]. L'effet de ces optimisations par exemple sur l'utilisation du cache local ou le comportement du pipeline du processeur peut changer les performances globales du programme de façon significative.

4.3.1.3 Buts de l'évaluation

On distinguera deux buts principaux dans l'évaluation des performances des programmes générés par un compilateur par distribution de données :

- Tout d'abord, cette évaluation doit aider le programmeur à obtenir le programme le plus efficace possible. Ceci peut être fait pour un compilateur et une plate-forme d'exécution données mais, pour des raisons de portabilité, il est préférable d'évaluer le programme de façon plus indépendante en faisant abstraction notamment des spécificités de la machine. Le but recherché par le programmeur est principalement d'être en mesure de choisir parmi différentes versions d'un même programme.

- L'évaluation sert également aux développeurs du système de compilation, elle doit permettre de mieux comprendre l'effet des stratégies mises en œuvre tant dans le compilateur que dans l'exécutif. L'évaluation est plus intéressante lorsqu'elle fournit des indices de performances indépendants d'une plate-forme d'exécution donnée et non pas seulement des données spécifiques.

Ces deux buts de l'évaluation ne sont pas forcément clairement distingués dans les outils existants. En effet, bon nombre de résultats d'évaluation servent à la fois aux programmeurs et aux concepteurs des compilateurs.

4.3.1.4 Méthodes d'évaluation

Il est possible d'utiliser deux approches pour évaluer les performances des programmes parallèles générés par un compilateur dirigé par la distribution des données.

L'approche la plus simple consiste à utiliser un outil conçu pour des programmes parallèles explicites, on évalue alors le programme généré par le compilateur, en faisant abstraction du fait qu'il a été produit par un compilateur. Cette approche présente l'avantage évident de pouvoir utiliser des outils généraux dont la conception et le développement bénéficient des travaux d'une plus grande communauté. Il faut tout de même faire en sorte que le code généré par le compilateur puisse s'exprimer dans le paradigme de programmation supporté par l'outil, ce qui peut faire l'objet du portage d'une partie du système de compilation.

L'inconvénient majeur de cette approche est l'abstraction qui est faite du programme de départ, qui rend très difficile l'interprétation des résultats fournis puisqu'ils ne tiennent pas compte des spécificités des codes générés et restent forcément à bas niveau. De plus, l'absence de certaines informations, en particulier celles contenues dans le compilateur, peut également diminuer la qualité de l'évaluation.

L'alternative consiste à utiliser des outils conçus et réalisés conjointement avec le système de compilation. Cette approche implique un effort de développement sans doute plus important ; en contrepartie, l'outil d'évaluation de performance peut avoir à sa disposition les informations contenues dans le programme source (en particulier la distribution des données) et dans le compilateur (*e.g.* la nature des optimisations des schémas de compilation). Les résultats d'évaluation sont potentiellement plus précis et surtout plus facilement interprétables. C'est cette approche qui a été choisie pour PANDORE.

Les deux approches pré-citées mettent en jeu des techniques communes. Nous distinguerons les techniques d'estimation de performances et les techniques de mesure de performances qui seront rapidement décrites à l'occasion de la présentation d'outils. Nous présentons dans le paragraphe 4.3.2 des outils de mesure généraux *i.e.* des outils considérant des programmes parallèles explicites. Le paragraphe 4.3.3 décrira des outils intégrés à des environnements de compilation par distribution de données comprenant des outils de mesure et des outils d'estimation.

4.3.2 Outils généraux

Trois outils de mesure de performances sont ici présentés. Deux d'entre eux (Picl/-ParaGraph et Topsys) ont fait l'objet d'une utilisation avec une version antérieure du système PANDORE [37]. À cet effet, l'exécutif avait fait l'objet d'un portage sur les bibliothèques supportées par ces outils. Le dernier (Alpes) a en commun avec les outils PANDORE un mécanisme de calcul de temps global [87].

Avant la présentation des outils proprement dits, nous précisons brièvement quelques notions sur la mesure de performance. Une description plus complète des techniques et outils d'observation des exécutions de programmes parallèles peut être trouvée dans [98].

L'évaluation des indices de performances peut se faire par mesures directes durant l'exécution du programme, lors de l'occurrence de certains événements (émission ou réception de messages, calculs, synchronisation,...). Les données de mesures sont acquises par un *moniteur* matériel, logiciel ou hybride. Les événements sont détectés par des *sondes* qui déclenchent alors un traitement spécifique. L'introduction des sondes dans le programme constitue l'instrumentation. Celle-ci peut-être faite à plusieurs niveaux : au niveau du programme source, dans une bibliothèque (notamment une bibliothèque de communication), dans le code objet ou à l'intérieur même du système. L'exécution du traitement associé aux sondes peut occasionner une perturbation de l'exécution du programme : ralentissement, voire changement dans l'ordre des événements ; on parle alors d'intrusion. Des techniques existent cependant pour la limiter [14].

Nous distinguerons deux méthodes d'acquisition de données de performance : la génération de trace et le *profiling*.

- La génération de traces consiste, durant l'exécution d'un programme, à enregistrer sur chaque processeur certains événements auxquels sont assignés au moins un type et une estampille. Cette estampille est donnée par une horloge physique ou logique et peut être locale ou globale. En plus des analyses statistiques, les traces recueillies permettent la ré-exécution (simulée) du programme[3] et donc une analyse fine de son comportement pour en déduire des indices de performance. Le principal inconvénient de cette méthode est le volume de trace généré, qui croît avec le temps d'exécution, nécessitant une gestion coûteuse de l'espace de stockage des traces.

- Le *profiling* permet de maintenir à jour durant l'exécution un nombre fixe de compteurs reliés à des événements. Étant donné que le *profiling* collecte uniquement des totaux, le volume d'informations conservé pendant l'exécution est limité. De plus l'intrusion est généralement plus faible que lors de la génération de traces car les traitements effectués par les sondes sont très simples (*e.g.* incrémentation d'un compteur). En revanche, le *profiling* recueille moins d'information et ne permet notamment pas la ré-exécution. L'objectif est plutôt de rassembler suffisamment de statistiques afin d'en déduire des indices de performance.

L'évaluation de performance par mesures peut conduire à l'acquisition d'un volume important de données, souvent de bas niveau, qu'il faut ensuite analyser. La façon dont sont présentées ces données à l'utilisateur est un aspect important des

3. On dit aussi que l'exécution est *rejouée*.

outils d'évaluation de performances. C'est la représentation graphique, complétant une représentation textuelle qui prévaut dans les outils actuels.

4.3.2.1 Picl/ParaGraph

Picl [50] (Portable Instrumented Communication Library) est une bibliothèque de communication instrumentée qui génère des traces exploitables par l'outil de visualisation graphique de performances ParaGraph [60].

Picl est une bibliothèque développée initialement pour les hypercubes iPSC et Ncube. Elle offre plusieurs primitives de communication (envoi et réception de message, diffusion), de synchronisation et d'opérations globales (réductions). Cette bibliothèque est instrumentée afin de générer différents types de traces suivant un format ASCII ou binaire [100].

Les traces relatives aux événements de communication et de synchronisation sont générées de façon transparente par les primitives concernées. Le programmeur peut toutefois contrôler leur génération et leur sauvegarde dans un fichier en utilisant des primitives ad hoc.

Le programmeur peut insérer dans le code des appels à une primitive générant une trace de type quelconque, enregistrant ainsi les événements de son choix.

Un *profiler* permet de cumuler des statistiques sur l'activité du CPU et les communications (nombre de messages, volume échangé, nombre de scrutations de files). Ces statistiques sont automatiquement calculées et délivrées dans une trace spécifique.

Tous les événements sont datés suivant une approximation du temps global obtenue par synchronisation des horloges locales au début de l'exécution du programme.

ParaGraph est un outil graphique permettant de rejouer l'exécution du programme parallèle d'après les informations contenues dans les traces générées par Picl. ParaGraph fournit plus d'une vingtaine de représentations différentes qui peuvent être affichées simultanément. La plupart des visualisations sont animées, elles sont mises à jour à chaque nouvel événement lu dans la trace. Le déroulement de la ré-exécution peut être arrêté, repris au départ et effectué pas à pas.

Parmi les visualisations on trouve :

- Les visualisations liées aux communications :
  - le diagramme espace-temps représentant les processus par des lignes parallèles dont certains points sont joints par une flèche représentant une communication point à point ;
  - le graphe des communications animé dans lequel les sommets représentent les processus — leur état est codé par une couleur — et les arcs les

communications (une vue sous forme de matrice donne des informations similaires).

- Les visualisations liées à l'activité des processeurs, présentées sous la forme d'histogramme ou de kiviat (polygone inscrit dans un cercle donc chaque rayon représente un processeur). Un processeur est considéré comme actif, inactif ou effectuant une communication.

Une présentation détaillée des vues est donnée dans [61].

Une version antérieure du compilateur PANDORE produisait du code faisant appel aux primitives de la bibliothèque instrumentée Picl, permettant ainsi d'utiliser ParaGraph pour analyser les performances. L'interfaçage de PANDORE avec Picl ne pose pas de problème mais les possibilités d'interprétation des performances sont peu satisfaisantes, l'analyse se faisant à bas niveau sans que l'on puisse relier facilement les résultats au code source PANDORE. La profusion des vues n'apporte pas une aide significative dans ce contexte. De plus, le contrôle réduit de la ré-exécution offre peu de souplesse d'utilisation.

4.3.2.2 Topsys

Topsys (TOols for Parallel SYStems) est un environnement de programmation des machines parallèles à mémoire distribuée par passage de message développé à l'université de Munich depuis 1986 [19]. Il se compose d'un noyau chargé sur l'ensemble des nœuds de la machine parallèle, d'un outil de placement, de moniteurs interchangeables (matériel sur l'iPSC/2, logiciel et hybride) chargés de collecter des informations pendant l'exécution des programmes, et de plusieurs outils graphiques exploitant ces informations (outil de débogage, d'évaluation de performance et de visualisation d'exécution).

Topsys offre un modèle de programmation fondé sur l'utilisation d'objets actifs (tâches) et passifs (boîtes aux lettres, sémaphores et zones mémoire) distribués sur les nœuds de la machines parallèle. Ces objets sont créés statiquement ou dynamiquement et peuvent être identifiés de façon globale, permettant leur partage par plusieurs nœuds.

L'approche adoptée pour l'observation de l'exécution de programmes est une approche « à la volée » (*on-line*), les outils d'analyse Vistop et Patop s'exécutant en parallèle avec le programme observé.

L'outil Vistop visualise le comportement dynamique du programme en se concentrant sur les événements de communication et de synchronisation. Les tâches, boîtes aux lettres et sémaphores sont représentés sous forme iconifiée, des flèches relient les icones afin de visualiser les interactions entre les objets. En plus de l'observation à la volée, on peut rejouer l'exécution du programme en mode continu ou pas à pas (en avant et arrière).

L'outil Patop [20] est plus particulièrement orienté vers l'évaluation de performance. L'accent est mis sur l'imbrication des niveaux d'abstractions : le système dans sa totalité, les nœuds puis les objets. Des vues multiples sont affichées sous la forme de courbes et d'histogrammes pour chacun des niveaux.

- Pour le système entier : le temps d'inactivité, le temps moyen d'attente des tâches ;
- Pour les nœuds : le temps d'inactivité, le temps moyen d'attente des tâches, le volume de données émis vers d'autres nœuds, le temps moyen de temps passé par les tâches en opérations distantes ;
- Pour les tâches : le temps d'attente pour les émissions et réceptions, le taux d'occupation CPU ;
- Pour les boîtes aux lettres : le volume émis ou reçu vers ou depuis chaque nœud, le temps d'attente des tâches en réception ;
- Pour les sémaphores : le temps d'attente.

Les indices de performances évalués par Topsys sont similaires à ceux de Picl/-ParaGraph. L'aspect hiérarchique qui caractérise les outils de visualisation est un atout appréciable, notamment par l'introduction du niveau tâche. Le modèle de programmation offert est peu adapté à la compilation par distribution de données et constitue un frein à une utilisation efficace dans ce contexte.

4.3.2.3 Alpes

ALPES (ALgorithms, Parallelism and Evaluation of Systems) est un projet visant à développer des outils d'évaluation de différents aspects de la mise en œuvre et de l'exécution de programmes parallèles sur machines à mémoire distribuée [75]. Trois axes sont conjointement étudiés : la génération de programmes parallèles synthétiques, l'acquisition de traces et la visualisation des données de performances, axes qui donnent lieu au développement d'outils en cours d'intégration.

La base de l'approche est l'observation de l'exécution d'un *programme synthétique* sur une machine parallèle. Ce programme synthétique modélise à la fois :

- le programme à évaluer ;
- les stratégies d'implémentation (*e.g.* les stratégies de placement) ;
- la machine parallèle cible sur laquelle le programme sera exécuté (éventuellement différente de la machine d'expérimentation).

Les programmes sont manuellement modélisés sous la forme d'un graphe de précédence de tâches annotées à des fins d'évaluation de performances [74]. Les annotations sont des annotations d'entrée/sortie qui détaillent les dépendances de données liées aux arcs de précédences (*e.g.* communications) et des annotations de calcul qui modélisent le traitement effectué par les tâches. Des coûts sont associés

aux annotations, ils peuvent être constants, suivre une distribution aléatoires ou dépendre d'autres coûts.

La machine parallèle cible est modélisée par un certain nombre de paramètres relatifs aux processeurs (*e.g.* le temps d'une opération flottante), à la mémoire (*e.g.* la taille d'un entier) et au réseau d'interconnexion (*e.g.* la topologie du réseau, la bande passante d'un lien).

À partir du modèle du programme parallèle et de celui de l'architecture cible, un programme synthétique est automatiquement produit en tenant compte d'une stratégie de mise en œuvre. Ce programme n'effectue pas de réels calculs mais simule la charge de calcul (notamment par des boucles vides) et de communication (par des transferts de messages dont seule la longueur est significative). Le programme synthétique peut ensuite être instrumenté et exécuté sur une machine parallèle.

Un outil d'instrumentation de programmes parallèles (écrits à l'aide de la bibliothèque de communication PVM [18]) a été développé afin de pouvoir générer des traces exploitables par analyse *post-mortem* [86]. Les programmes sont instrumentés par un préprocesseur. L'apport principal de cet outil est l'utilisation d'un mécanisme de datation cohérente des événements suivant un temps global. Une méthode statistique et non intrusive est utilisée [87] : elle est fondée sur la détermination des décalages et des dérives des différentes horloges locales par rapport à l'horloge d'un des nœuds de la machine (l'horloge de référence). Les mesures nécessaires à l'évaluation des décalages et des dérives des horloges étant effectuées seulement avant et après l'exécution de l'application proprement dite, celle-ci ne subit aucune perturbation due au mécanisme de datation globale. En contrepartie, il est nécessaire d'attendre la fin de l'exécution répartie pour que les dates globales puissent être calculées. Par ailleurs plusieurs techniques sont utilisées pour réduire au maximum l'intrusion de la génération de traces (compaction, réduction du nombre des messages et tâches additionnels). À l'heure actuelle, les traces générées peuvent être mises au format Picl pour être exploitées grâce à ParaGraph.

Par ailleurs, un outil interactif d'analyse graphique de traces appelé Scope a été mis au point [9]. Scope est un outil facilement extensible qui sépare les notions de ré-exécution et de visualisation. Le temps simulé peut être contrôlé de plusieurs façons (déroulement en avant, en arrière, enregistrement de repères, positionnement sur un repère). Scope offre un certain nombre de visualisations statiques et animées telles que les histogrammes, les kiviats, les diagrammes espace-temps ou les graphes de communication. Les représentations graphiques sont personnalisables (couleurs, formes, étiquettes, échelles...), on peut également y appliquer des filtres et des regroupements afin de diminuer la complexité des affichages. Un des points forts de cet outil est à notre avis son adaptabilité, qu'on retrouve aussi dans un outil comme Pablo [92] ; il permet une utilisation dans un contexte plus large que celui d'Alpes.

Si l'utilisation directe de l'environnement Alpes dans le contexte de la compilation par distribution de données paraît difficile au vu de la modélisation des

programmes requise, plusieurs des techniques mises en œuvre comme la production de programme synthétique peuvent être réutilisées. C'est le cas également pour le mécanisme de calcul de temps global dont l'algorithme a été utilisé dans les outils d'observation de l'environnement PANDORE.

4.3.3 Outils associés à des compilateurs dirigés par les données

Nous présentons dans ce paragraphe trois outils intégrés dans des environnements de compilation par distribution de données. Deux d'entre eux sont des outils d'estimation de performance (estimateur de Fortran D et PPPT) alors que le troisième, associé à l'environnement EPPP, utilise des techniques de mesure de performance.

Avant de décrire ces outils, nous présentons succintement les différents aspects relatifs à l'estimation de performance dans le cadre de la compilation par distribution de données.

L'estimation de performance vise à prédire la performance d'un programme parallèle sur une machine cible sans l'exécuter. Le résultat de l'estimation consiste principalement en un temps d'exécution attendu du programme ou d'une partie du programme ; d'autres paramètres peuvent également être estimés. La base de la technique repose en général sur l'utilisation de modèles de performance de certaines opérations élémentaires, et sur la détermination du nombre de fois où chacune de ces opérations élémentaires sera exécutée.

- *Modélisation des coûts des opérations élémentaires.*
  Ces opérations doivent couvrir l'ensemble des opérations de calcul et de communication générées par le compilateur et leur coût doit être prédit avec précision. Le fait que les opérations soient considérées individuellement rend très difficile la prise en compte de caractéristiques « fines » telles que la hiérarchie mémoire ou la contention sur le réseau de communication. Les modèles peuvent être des modèles analytiques ou des modèles issus de l'exécution de jeux d'essais.

- *Détermination du nombre d'opérations.*
  La complexité du flot de contrôle est un obstacle à la précision de l'estimation. En effet, chaque branchement dont la direction est inconnue à la compilation induit une incertitude sur le nombre d'opérations réellement exécuté[4]. Il en résulte que les estimations sont plus précises pour les parties de codes SPMD optimisées qui comportent peu ou pas de tests. Même si le flot de contrôle ne dépend pas de données connues seulement à l'exécution, il est parfois difficile de déterminer le nombre d'opérations effectuées. C'est le cas par exemple pour

4. Une alternative pour prendre en compte n'importe quel flot de contrôle est d'utiliser des techniques de simulation pour l'estimation [43].

les nids de boucles dont les bornes ne sont pas constantes et pour lesquels les techniques de comptage exact peuvent être très coûteuses [94].

Un estimateur de performance peut être utilisé comme un composant du système de compilation. Il permet de comparer les performances obtenues pour plusieurs stratégies de compilation et donc de choisir la meilleure. Dans cette optique, il peut être utilisé pour la compilation automatique de programmes séquentiels en aidant à la détermination d'une distribution de données [57, 11].

4.3.3.1 Estimateur de performance de Fortran D

Dans le cadre du projet Fortran D [63] a été développé un prototype d'outil d'évaluation de performance. Il s'agit d'un estimateur statique de performance dont le but est de guider le programmeur dans ses choix de distribution des données [12].

Cet outil est basé sur l'élaboration d'un ensemble de données statistiques modélisant les différents coûts de calcul et de communication sur une architecture donnée (*Training Set Method*). Un certain nombre d'instructions de calcul élémentaires et d'appels à des primitives de communication sont rassemblées dans un programme appelé le *training set*. Ce programme est exécuté sur l'architecture cible et les temps des opérations sont mesurés. En ce qui concerne les communications, les opérations considérées sont des opérations de haut niveau (permutation circulaire d'éléments et de vecteurs, diffusions, réductions globales, etc.) étudiées pour plusieurs tailles et configuration de données. Un important volume de données de performance est ainsi recueilli. Dans un deuxième temps, ces données sont analysées afin de réduire leur volume : les performances des opérations de communication sont modélisées par des fonctions linéaires par morceaux (en utilisant une variante de la méthode du χ^2) ; les performances des opérations arithmétiques sont résumées par des moyennes.

La procédure d'estimation de performance est appliquée sur une portion du code généré par le compilateur et donne en résultat le temps d'exécution estimé ainsi que la part de communication dans ce temps. Elle consiste à appliquer les modèles de performances aux opérations rencontrées après avoir déterminer le nombre d'occurrence de ces opérations à l'aide des bornes de boucles si elles sont constantes. Les valeurs des variables symboliques sont demandées interactivement à l'utilisateur. Il en est de même pour les probabilités de branchement des conditionnelles.

La méthode employée simplifie la tâche d'estimation des performances car l'utilisation des *training sets* permet de s'affranchir de l'élaboration d'un modèle analytique complexe de l'architecture cible. Cependant, la difficulté de sa mise en œuvre reste grande car il faut élaborer des *training sets* en tenant compte de tous les schémas de communications susceptibles d'être générés par le compilateur.

Le manque de précision de la méthode fait que cet outil n'est pas destiné à effectuer des estimations absolues des temps d'exécution des programmes générés mais plutôt à classer grossièrement les versions obtenues à partir d'un même pro-

gramme pour différentes distributions des données afin d'éliminer les distributions qui, manifestement, ne permettent pas d'obtenir des performances.

4.3.3.2 PPPT

PPPT (*Parameter Based Performance Prediction Tool*) est un un outil d'estimation de performance développé par Thomas Fahringer [44, 42]. Cet outil est associé à VFCS, le système de compilation de Vienna Fortran [103].

L'utilisation de cet outil se déroule en trois temps.

1) Le programme source (programme séquentiel annoté par la distribution des données) exprimé en Vienna Fortran est analysé et instrumenté par un outil nommé *Weight Finder*. Le programme – séquentiel – simplifié obtenu est exécuté afin de déterminer un certain nombre de caractéristiques quantitatives, à savoir : la fréquence de passage dans les branches de chaque conditionnelle, le nombre de tour de chaque itération, la fréquence de chaque instruction (nombre de fois où elle est exécutée). Le *Weight Finder* détermine également les zones du programme dans lesquelles la plupart des calculs sont fait, zones sur lesquelles l'effort d'optimisation devra être concentré.
2) Le programme source est compilé par VFCS qui produit un programme explicitement parallèle.
3) Les statistiques calculées par le *Weight Finder*, le programme parallèle généré et la distribution des données sont passés à PPPT qui détermine plusieurs paramètres caractéristiques des performances du programme parallèle.

Les résultats de PPPT sont donnés au programmeur[5]. Ils devraient également pouvoir servir à guider les stratégies d'optimisations du compilateur, voire à modifier les distributions de données, les étapes 2) et 3) s'enchaînant jusqu'à obtention de performances satisfaisantes. Les paramètres calculés par PPPT sont de deux natures :

- Les paramètres indépendants de la machine cible.
  Il s'agit de la répartition de charge, du nombre de messages et du volume transféré. Le calcul de ces paramètres est fondé sur l'analyse des distributions de variables, des domaines d'itérations et des domaines d'accès ; les résultats de l'analyse de recouvrement effectuée par VFCS, qui a été décrite au chapitre 2, sont également considérés par le calcul. La mise en œuvre effectue des unions englobantes et des intersections de polyèdres ainsi que des calculs de volume de polyèdres par triangularisation.
- Les paramètres spécifiques à une machine cible.
  Ce sont le temps de communication, la contention réseau (définie, pour une

5. Les valeurs des paramètres sont présentées en regard du code intermédiaire généré par le compilateur-restructureur interactif de VFCS.

boucle, par le nombre de fois où au moins deux messages empruntent le même lien de communication lors de l'exécution de la boucle) et le nombre de défauts de cache. Pour ces paramètres, un modèle de la machine cible (intel iPSC/860) est construit, il prend en compte la topologie, la latence et le débit des liens de communication (en tenant compte du routage) ainsi que le nombre et la taille des lignes de cache.

Les expérimentations conduites avec PPPT ont montré que pour plusieurs noyaux, le classement des versions des programmes (obtenus en modifiant la distribution des variables et le nombre de processeur) par temps d'exécution effectif pouvait être prédit par classement suivant les valeurs des paramètres estimés ; le temps de transfert, le nombre de messages puis la répartition de charge s'avérant être les paramètres prépondérants dans VFCS.

4.3.3.3 Débogueur de performance d'EPPP

L'environnement de compilation EPPP (*Environment for Portable Parallel Programming* [31]) comprend un outil d'analyse de performance intégré [65]. Sa particularité est que sa conception a été guidée par le fait qu'il devait être adapté au paradigme de programmation par distribution de données. Le principe de l'outil est la représentation graphique de données de performances obtenues par analyse symbolique du code source et par analyse et ré-exécution de traces d'exécution.

L'outil est en effet capable d'afficher de façon interactive plusieurs types d'informations concernant certains objets du code source (fonctions, boucles, variables...).

- La distribution des données, telle qu'elle a été indiquée dans le programme source, peut être graphiquement représentée. Ceci permet de vérifier les spécifications complexes. Cette représentation est également utile pour connaître les distributions que l'optimiseur a pu choisir en remplacement de celle spécifiée dans le programme source[6].
- La succession des accès aux données peuvent constituer une animation des représentations graphiques des tableaux distribués. Différentes couleurs représentent les accès en lecture et en écriture, pour les données locales ou distantes. L'objectif est de pouvoir localiser les éléments de tableau impliqués dans une phase de calcul donnée et de mieux appréhender les motifs de communications mis en jeu. Une version modifiée de cette animation peut être utilisée pour visualiser les accès simultanés sur différents processeurs permettant ainsi d'apprécier la répartition de charge.
- Des statistiques sur les communications peuvent être fournies, il s'agit notamment du nombre de messages émis et reçus et du volume transféré.

6. Après analyse des accès, le compilateur d'EPPP est capable de choisir, dans certaines boucles, une distribution plus efficace que celle proposée par le programmeur [90].

Pour générer ces représentations graphiques, le débogueur de performance exploite des informations fournies par le compilateur (texte source, distribution des variables, topologie de processeurs virtuels, liste des fonctions et variables...) et contenues dans les traces d'exécution. Ces traces sont obtenues après exécution d'une version instrumentée du programme distribué. L'instrumentation est faite à deux niveaux :

- Au niveau de la bibliothèque de communication.
  La bibliothèque employée est une version étendue de Picl [50] qui permet notamment de rajouter des informations liées aux objets du programme source dans les traces. Ceci est réalisé par l'adjonction de primitives spécifiques générant des traces qui peuvent être mises en relation avec les traces standard automatiquement générées par les primitives Picl.
- Au niveau du compilateur.
  Afin de rassembler des informations qui ne sont pas liées aux communications (statistiques sur les fonctions, accès aux données distribuées), le compilateur introduit des sondes spécifiques.

Par rapport aux outils classiques de type ParaGraph, cet outil apporte des éléments de visualisation nouveaux, spécifiques à l'approche de la compilation par distribution de données. L'analyse graphique des mouvements de données semble très utile à la compréhension de l'impact de la distribution sur les performances. L'instrumentation est actuellement limitée aux portions de code non optimisées ; cela réduit l'intérêt des analyses puisque l'influence des optimisations du schéma de compilation ne peuvent pas être prises en compte.

4.3.4 Outils d'évaluation dans Pandore

4.3.4.1 Motivations

L'axe de recherche sur l'évaluation de performance dans l'environnement Pandore est celui du développement d'outils fondés sur l'exploitation de mesures effectuées durant l'exécution distribuée des programmes Pandore.

L'objectif visé n'est pas de proposer des outils complets mais plutôt d'expérimenter des techniques d'acquisition de mesures et de visualisation ainsi que d'étudier l'intégration de ces techniques dans un environnement de compilation dirigée par les données. L'utilisation de techniques de mesure permet l'obtention de données précises qui tiennent compte de tous les facteurs influençant les performances (algorithme, distribution des données, compilateur, exécutif, machine parallèle cible). Un certain nombre d'outils d'instrumentation et d'analyse ont été mis au point et partiellement intégrés à l'environnement. Une intégration plus complète et une phase d'utilisation systématique de ces outils sur des programmes Pandore doit permettre d'identifier les avantages et les manques des techniques mises en œuvre.

Les outils développés ont deux utilisations possibles :

- Ils permettent à l'utilisateur de comparer plusieurs versions d'un programme PANDORE. L'évaluation permet également d'identifier les phases, instructions, variables ou accès responsables d'éventuelles mauvaises performances. À cette fin, les résultats d'évaluation sont reliés aux objets du programme source.
- Pour les développeurs du système de compilation PANDORE, l'évaluation des performances de noyaux de programmes et la comparaison avec des résultats attendus permettent d'apprécier l'impact et de chiffrer les améliorations des différentes stratégies d'optimisation mises en œuvre dans le compilateur et l'exécutif.

L'évaluation se fait suivant deux axes : l'analyse de données quantitative et la compréhension du comportement dynamique du programme. Le premier axe met en œuvre une méthode de *profiling* alors que le deuxième exploite des techniques de génération et d'analyse de traces[7].

4.3.4.2 Profiler

Le *profiler* PANDORE permet de collecter et d'analyser graphiquement un certain nombre de mesures quantitatives sur l'exécution des programmes générés par le compilateur, ceci avec une intervention minimum du programmeur.

Instrumentation

L'instrumentation est réalisée à deux niveaux, au niveau du compilateur et à l'intérieur de l'exécutif.

- Lorsque l'utilisateur désire évaluer les performances de son programme, le compilateur remplace les appels aux primitives standard de l'exécutif par des appels à des primitives instrumentées. Ceci est fait systématiquement pour les primitives relatives aux phases distribuées (primitives de déclenchement des phases, de passage de paramètres...). Pour les primitives utilisées dans la traductions des opérations *Refresh* et *Exec*, le remplacement est effectué dans les portions de code correspondant à des instructions du programme source situées à l'intérieur de *zones d'instrumentation*. Ces zones sont choisies par l'utilisateur et indiquées sur la ligne de commande de compilation.

  Le compilateur génère par ailleurs des tables de description du programme source : tables des zones d'instrumentation, des phases distribuées, des variables distribuées et dupliquées. Ces tables contiennent notamment des identificateurs et des repères dans le fichier source.

7. Les techniques liées aux traces font l'objet d'autres utilisations au sein de l'équipe Pampa, dépassant le cadre de la compilation par distribution de données [15, 71].

- Les versions instrumentées des primitives de l'exécutif mettent en œuvre une méthode de *profiling* étendu, pour certains événements, en plus de compter leur occurrence, la sonde cumule leur durée [73]. C'est le cas pour les événements de réception (on considère que l'événement en question est un *événement composé* [98], distinguant le moment où la primitive de réception est appelée et celui où le message est arrivé).

Les mesures sont effectuées sur chaque nœud, rapatriées sur l'hôte à la fin de l'exécution puis écrites dans un fichier au format ASCII. L'utilisation de la technique du *profiling* restreint la quantité de mémoire supplémentaire nécessaire et supprime les problèmes de stockage externes durant l'exécution du programme, le nombre de compteurs mis à jours étant fixé à la compilation et de l'ordre du nombre de variables apparaissant dans le programme. L'intrusion reste très faible (inférieure à quelques pourcents dans tous les noyaux testés), elle est également réduite par le fait que le programme n'est pas entièrement instrumenté.

Mesures

Outre les temps d'exécution, les principaux résultats fournis par le *profiler* concernent la répartition de charge, les communications et les synchronisations. Les synchronisations sont exprimées en terme de temps d'attente sur les réceptions bloquantes, c'est-à-dire que l'on mesure le temps entre le moment où la primitive de réception est invoquée et celui où le message est disponible dans la file d'attente.

Les résultats de mesures peuvent être classés en deux catégories : les résultats propres aux phases distribuées et les mesures concernant les affectations situées dans les zones d'instrumentations.

Mesures propres aux phases distribuées

Les phases distribuées PANDORE sont des phases exécutées en parallèle sur les nœuds de la machine. Elles ne peuvent être appelées que depuis le programme principal. C'est le processeur hôte qui déclenche l'exécution d'une phase sur les nœuds, et qui envoie les données correspondant aux variables passées en paramètre d'entrée (mode IN). De même, les nœuds envoient à l'hôte les valeurs correspondant aux paramètres en sortie (mode OUT). Le *profiler* mesure, pour chaque phase distribué :

- le temps d'attente sur le déclenchement de la phase ;
- le temps d'attente sur réception de chaque partition locale[8] de tableau et de chaque scalaire passé en paramètre en mode IN ;
- le temps total de réception des paramètres passés en mode IN ;
- le temps total d'envoi des paramètres passés en mode OUT.

8. La partition locale d'un tableau est l'ensemble des éléments possédés par un processeur donné.

Mesures sur les affectations

Elles rendent compte de la répartition de charge, de la localité des accès, des communications point à point et des diffusions pour chaque zone d'instrumentation.

- *Répartition de charge.*
  On mesure le nombre d'affectations effectuées en faisant la distinction entre les affectations de variables distribuées et les affectations de variables dupliquées (effectuées par tous les processeurs).

- *Localité.*
  On compte le nombre d'accès locaux et le nombre d'accès distants.

- *Communications point à point.*
  L'affectation d'un élément de tableau distribué par une expression contenant une référence à un élément de tableau distribué peut conduire à un certain nombre d'accès distants mis en œuvre par communication. Le but des mesures est de construire globalement le graphe de communication entre partitions locales. Les sommets de ce graphe sont les partitions désignées par un nom de variable et un numéro de processeur. Les arcs du graphe décrivent les transferts d'éléments entre les partitions. Les arcs sont valués par le nombre de messages, le volume transféré ou le temps d'attente sur les réceptions. Par exemple, l'affectation $A[3][5] = B[4]$ augmente la valeur de l'arc $(B1 \rightarrow A2)$ si l'élément $A[3][5]$ est situé sur le processeur 2 et $B[4]$ sur le processeur 1.

- *Diffusions.*
  De même, l'affectation d'une variable dupliquée par une expression où apparaît une variable distribuée peut donner lieu à une diffusion depuis le possesseur de l'élément de la variable distribuée. Sur une zone d'instrumentation, les diffusions sont décrites par le nombre de messages, le volume transféré ou le temps d'attente sur réception relatif à chaque couple (*var*,*part*) où *var* désigne une variable dupliquée[9] et *part* désigne partition source de la diffusion. Par exemple, l'affectation $x = A[3][5]$ augmente la valeur des compteurs liés au couple (x, $A1$) si A[3][5] est possédé par le processeur 1.

Visualisation

Un outil permet de visualiser graphiquement toutes les mesures décrites plus haut, offrant un confort d'analyse plus grand que la consultation directe des données brutes. La plupart des données de performance sont représentées à la demande par des tableaux, des histogrammes et des kiviats afin de rendre compte des valeurs mesurées sur chaque processeur (les moyennes et écarts types sont également précisés pour chaque graphique). Un certain nombre de sélections sont applicables pour réduire la quantité de données présentées.

9. Si la variable dupliquée est un tableau, seul l'identificateur du tableau est donné par *var*.

L'intérêt principal de cet outil graphique est la possibilité de visualiser le graphe de communication entre partitions qui rend compte des communications point à point. C'est sur le graphe de communication que l'utilisateur peut effectuer le plus de manipulations. Des sélections et regroupements d'arcs et de sommets sont possibles, ainsi que le choix parmi trois valuations (nombre de messages, volume transféré ou temps d'attente sur réception). Par ailleurs, des fonctions de déplacements et de zoom du graphe sont utilisables. L'analyse des motifs des graphes obtenus pour différentes zones du programme permettent de mieux repérer les causes de mauvaises performances ; de plus, comme les sommets du graphes sont des partitions, il est relativement facile de relier les performances des communications aux distributions de variables.

Exemple d'utilisation

Pour illustrer l'usage du graphe produit par le *profiler* afin de choisir une distribution de données, considérons le programme PANDORE de la figure 4.17 exécuté sur 4 processeurs ($P = 4$ et $N = 128$). L'examen du premier nid de boucles conduit à décomposer le vecteur V en blocs de 32 éléments et la matrice A en groupes de 32 lignes (distribution a utilisée dans la figure 4.17). Un autre choix serait de prendre en compte d'abord le deuxième nid de boucle. Cela conduirait à décomposer A en groupes de 32 colonnes (distribution b). On peut également envisager une solution intermédiaire : la décomposition de A en blocs de taille $(64, 64)$ (distribution c). La figure 4.18 montre les graphes de communication obtenus avec le *profiler* pour les trois distributions (on a une seule zone d'instrumentation regroupant les deux nids de boucles). C'est la distribution en ligne qui apparaît la meilleure au vu du nombre de messages, ceci est confirmé par le graphe des temps d'attente sur réception qui montre une forte synchronisation pour les distributions en colonnes et en blocs.

La figure 4.19 montre un exemple de session d'étude de performance avec notre outil graphique. Il porte sur le programme de la figure 4.17 dans lequel on a modifié la taille des données ($N = 256$) et la distribution de A (partitionnement en groupes de colonnes *i.e.* blocs de $N \times N/P$). Le programme a été exécuté sur 8 processeurs.

Exploitation statistique

Une autre voie d'exploitation des résultats du *profiler* PANDORE est envisagée, elle repose sur une analyse statistique permettant de dégager des relations entre les caractéristiques (statiques et dynamiques) du programme [88].

La méthode consiste en plusieurs phases :

- Une analyse statique du programme PANDORE (ou d'un fragment du programme, un nid de boucles par exemple) et de la distribution des données est effectuée afin d'obtenir une description en termes de valeurs de paramètres tels que la taille des tableaux, la taille des blocs, la stratégie d'attribution des

```
#define N 128
#define P 4

float A[N][N], V[N] ;

dist calc( float A[N][N] by block(N/P,N) map regular(0,1) mode INOUT,
           float V[N] by block(N/P) map regular(0) mode INOUT)
{
   int i,j ;

   for (i=0 ; i<N ; i++)
      for (j=0 ; i<N ; i++)
         V[i] = f(V[i],A[i][j]) ;

   for (i=0 ; i<N ; i++)
      for (j=1 ; i<N-1 ; i++)
         A[i][j] = g(A[i+1][j],A[i-1][j]) ;
}

main()
{ calc(A,V) ; }
```

FIG. 4.17. – *Exemple de programme source* PANDORE

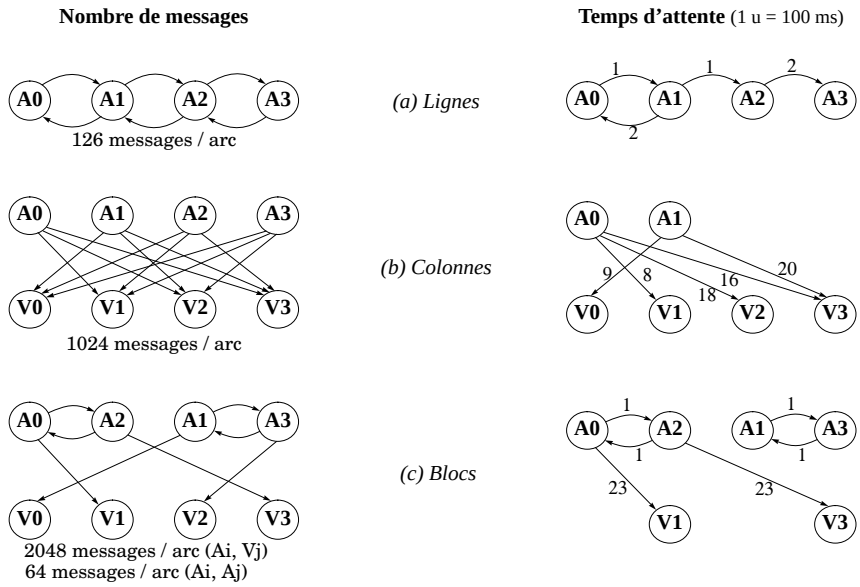


FIG. 4.18. – *Graphe de communication point à point pour trois distributions*

blocs aux processeurs, la profondeur des nids de boucles et l'ordre des indices de boucles. Le nombre de processeurs est également inclus à la liste de ces paramètres dits statiques.

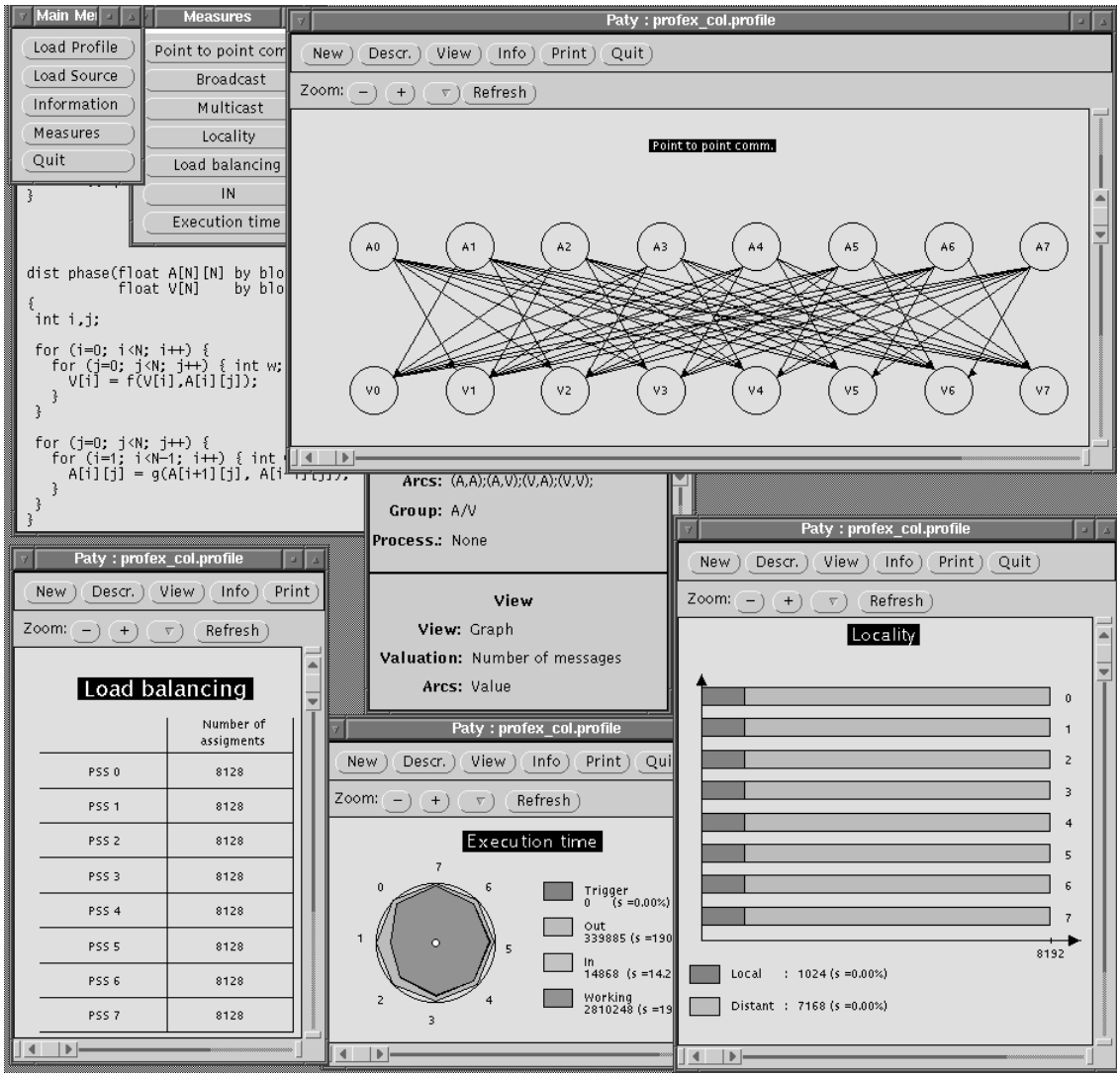


FIG. 4.19. – *Exemple de session avec l'outil graphique associé au profiler* PANDORE.

- Le programme PANDORE est compilé et exécuté (en version instrumentée) en faisant varier chacun des paramètres statiques. Ceci permet d'obtenir un certain nombre de paramètres dynamiques, concernant la répartition des calculs et les communications (*i.e.* tous les résultats de mesures du *profiler*).
- On considère l'espace multi-dimensionnel défini par les paramètres statiques et dynamiques. Un point de cet espace caractérise une exécution particulière. Les relations entre les paramètres sont étudiées en appliquant plusieurs analyses statistiques.
  - On réduit tout d'abord la complexité du modèle en constituant la matrice de corrélation entre paramètres, celle-ci permet d'éliminer les paramètres fortement corrélés.

– On applique ensuite une analyse de groupe (*cluster analysis*) qui permet de regrouper certains points d'après un critère de similitude, chaque groupe étant caractérisé par son centre. Une description qualitative des différents groupes obtenus peut enfin être déduite, description qui relie les paramètres statiques aux paramètres dynamiques. Ainsi, une analyse de groupes sur l'exemple du Jacobi permet d'isoler le groupe des exécutions aux caractéristiques suivantes : distribution cyclique, nombre important de blocs par processeur, grand nombre de messages transférés et long temps d'exécution. On vérifie ainsi que ce genre de distribution ne convient pas à l'algorithme.

4.3.4.3 Génération et analyse de traces

L'environnement PANDORE dispose d'outils de génération et d'analyse de traces d'exécution. L'objectif principal du développement de ces outils est la compréhension du comportement des programmes et particulièrement l'étude de la structure causale des programmes, structure qui rend compte des performances du programme car elle reflète des notions comme la répartition de charge ou les synchronisations.

La collecte des traces s'appuie sur la bibliothèque de communication et d'observation POM, introduite au paragraphe 3.2.1. L'analyse des traces recueillies est essentiellement fondée sur l'exploitation graphique des dépendances causales entre événements.

La POM

En plus des services de communications qui ont été déjà présentés, la POM offre plusieurs services d'observations : l'utilisation d'un nœud observateur, le traçage d'événements, l'estampillage automatique et la datation globale.

Nœud observateur

La POM permet d'associer aux nœuds d'application un *nœud observateur* ayant pour fonction de collecter et d'exploiter les informations de traces relatives au comportement de l'application. Elle offre au programmeur un ensemble de primitives permettant le développement des programmes d'observation. Ces primitives permettent de réceptionner les messages de trace de manière déterministe ou non déterministe, et d'en extraire les différentes composantes significatives. Le nœud observateur peut ainsi procéder à une analyse des informations reçues « à la volée », ou se contenter de les stocker pour une utilisation ultérieure (analyse *post–mortem*).

Génération de traces

Il demeure à la charge du programmeur d'application de spécifier quels événements doivent être tracés en insérant des points d'observations (*i.e.* des appels à la pri-

mitive APS_trace). Lors de l'exécution, le passage sur un point d'observation va se traduire par l'envoi d'un message de trace vers le nœud observateur. Un message de trace contient des informations précisées par l'utilisateur et des informations liées à l'estampillage et la datation de l'événement tracé, ces dernières étant générées automatiquement.

Estampillage

L'estampillage consiste à associer une date logique aux événements en faisant abstraction du temps physique. Ceci permet de déterminer les dépendances causales entre événements. Dans sa version actuelle, la POM propose à l'utilisateur de choisir entre deux types d'estampilles : des estampilles vectorielles [45] dont la taille est constante au cours d'une exécution et déterminée par le nombre de nœuds d'application, ou des estampilles dites « adaptatives » dont la taille peut varier en cours d'exécution [68]. L'intrusion occasionnée par l'estampillage n'est pas négligeable puisque les estampilles sont ajoutées à chaque message de l'application. Toutefois ceci ne pose pas de problème car la causalité n'est pas altérée, les programmes PANDORE possédant un déterminisme particulier qui fait que la causalité n'est pas modifiée d'une exécution à l'autre [15].

Datation physique globale

Le mécanisme de datation physique utilisé par défaut réalise une datation des événements en fonction du temps local à chaque nœud d'application (valeur retournée par l'horloge locale de chaque processeur). La POM intègre également un mécanisme de datation globale des événements. L'approche choisie est fondée sur une méthode statistique d'estimation des décalages et des dérives des différentes horloge locales des nœuds d'application par rapport à une horloge de référence [58]. L'algorithme est le même que celui utilisé dans l'environnement Alpes présenté au paragraphe 4.3.2. C'est une approche non intrusive (aucun calcul n'est effectué pendant l'application proprement dite) mais les dates globales ne sont connues qu'après l'exécution. Le mécanisme de datation globale mis en œuvre dans la POM n'est donc approprié que pour l'analyse de traces *post-mortem*.

Utilisation dans Pandore

L'insertion des appels à la bibliothèque POM pour la génération de traces s'appuie sur l'instrumentation automatique réalisée par le *profiler* PANDORE. L'utilisateur définit des zones d'instrumentation pour lesquelles des versions instrumentées des primitives de l'exécutif sont invoquées. Ces primitives font appel à la routine APS_trace. Ceci permet de tracer automatiquement les événements de communications et de calcul.

Traitement des traces

L'ensemble des traces générées par routine APS_trace est collecté et traité par un observateur générique qui fournit un ensemble d'événements aux outils de visualisation en tenant compte des choix de l'utilisateur lors du lancement de l'exécution du programme PANDORE. Une description plus détaillée de ces aspects est faite dans [15].

- Si l'option d'estampillage vectoriel a été choisie, l'observateur stocke momentanément une partie des événements jusqu'à ce que tous leurs prédécesseurs (au sens de la causalité) soient arrivés. Il peut ainsi utiliser un algorithme de linéarisation qui permet de délivrer les événements au visualiseur dans un ordre qui soit une extension linéaire.

- Si l'utilisateur a choisi d'utiliser le temps global, l'observateur stocke tous les événements et corrige leur date en fonction des dérives d'horloges calculées en fin d'exécution. Il délivre ensuite l'ensemble d'événements classés au visualiseur.

Visualisations

Le visualiseur reçoit un ordre partiel et l'affiche sous la forme d'un diagramme de Hasse ou d'un chronogramme s'il dispose de dates globales. L'ordre partiel peut également faire l'objet de la construction (à la volée) du treillis des idéaux.

Dans le diagramme de Hasse, on associe à chaque processeur un axe vertical sur lequel on fait figurer (de bas en haut) par des points les événements tracés. Un événement est relié à un événement situé plus bas (sur le même axe ou sur un axe différent) s'il en dépend causalement. Lorsque les événements tracés ont une date physique globale, une échelle de temps peut être ajoutée au diagramme de Hasse, on obtient alors un chronogramme. Les ordonnées des points ne correspondent plus seulement à la hauteur dans l'ordre partiel mais à la date d'exécution. La figure 4.20 montre à gauche un diagramme de Hasse obtenu pour l'exécution sur Paragon du produit de matrices sur 4 processeurs (les émissions et réceptions sont tracées), et à droite le chronogramme correspondant. Ce dernier indique que les communications prennent peu de temps par rapport aux calculs.

Le treillis représente l'ensemble des comportements possibles du programme. C'est un graphe dans lequel chaque sommet représente un instant de l'exécution et les arcs sortant sont les actions qui peuvent être effectuées à cet instant par les processeurs qui ne sont pas bloqués. Ainsi, chaque chemin du sommet initial au somment final correspond à un entrelacement possible de la suite des actions du programme. Par exemple, à partir du programme suivant où $x[i]$ et $y[j]$ sont placés sur le processeur p_1 et $z[i]$ sur le processeur p_2,

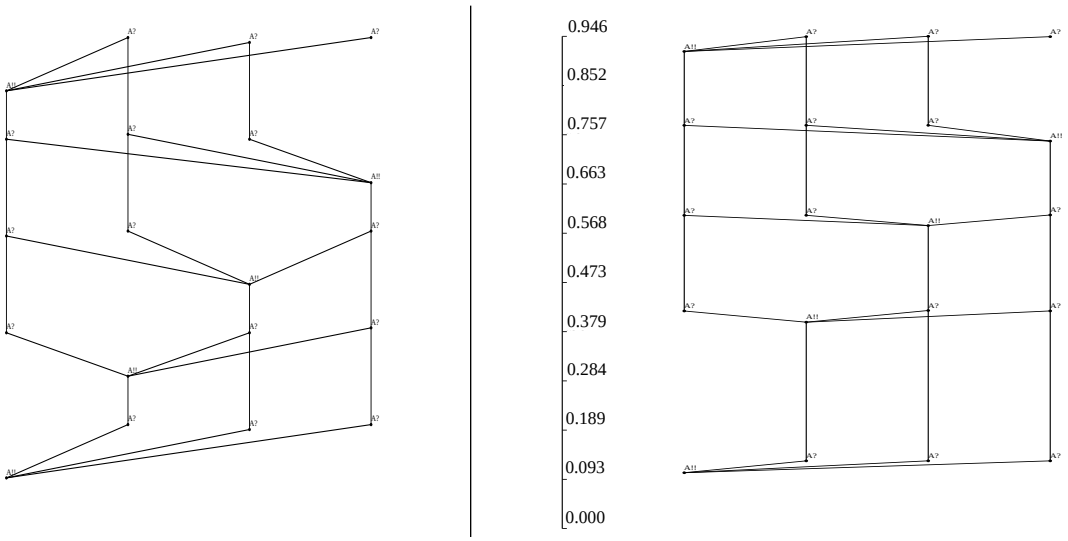
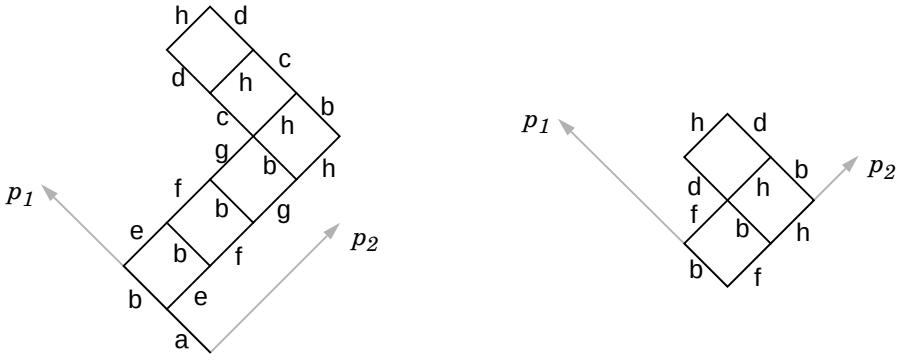


FIG. 4.20. – *Diagramme de Hasse et chronogramme pour le produit de matrices*

Code source	Exécution sur p_1	Exécution sur p_2
$z[i] = y[j] + 1$	(a) envoyer $y[j]$ à p_2	(e) recevoir $y[j]$ de p_1
$x[i] = 5$	(b) $x[i] = 5$	(f) $z[i] = y[j] + 1$
$y[j] = x[i] + 3 * z[i]$	(c) recevoir $z[i]$ de p_2	(g) envoyer $z[i]$ à p_1
$z[i] = z[i] - 3$	(d) $y[j] = x[i] + 3 * z[i]$	(g) $z[i] = z[i] - 3$

on obtient, en assignant une direction à chaque processeur, le treillis de la figure 4.21(i). On peut abstraire certains événements afin de réduire la taille du treillis. Par exemple, on peut observer uniquement les affectations sans perte de précision ; on obtient le treillis de la figure 4.21(ii).



(i) treillis complet (ii) treillis des affectations

FIG. 4.21. – *Treillis des idéaux*

La figure 4.22 montre les treillis obtenus après exécution du produit de matrices sur 4 processeurs en utilisant trois distributions différentes (le schéma de compilation de base a été utilisé). Les événements observés sont les affectations. Pour la distribution (c), beaucoup de points n'ont qu'un ou deux arcs sortant : le parallélisme y donc est très faible. En revanche la distribution (a) montre une largeur plus grande, signe de parallélisme potentiel, malgré des resserrements correspondant à des synchronisations.

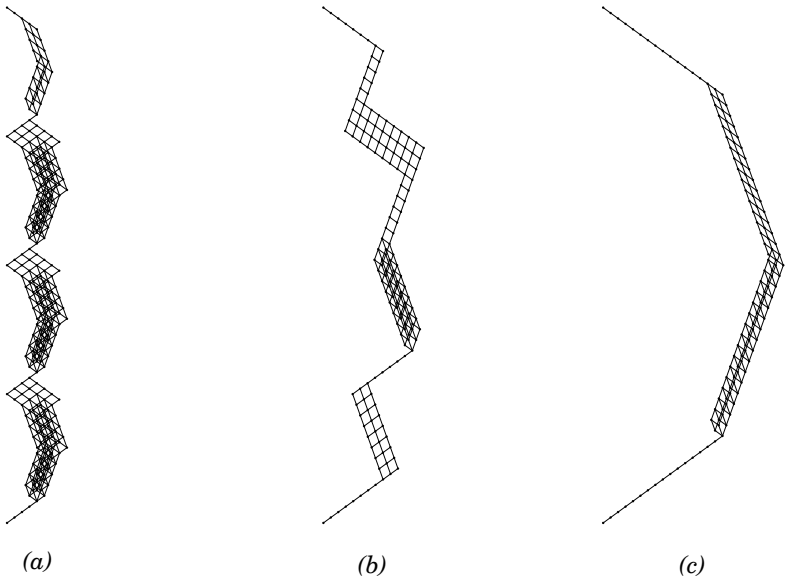


FIG. 4.22. – *Treillis obtenus pour le produit de matrices avec trois distributions*

Une série d'expériences préliminaires sur l'utilisation des outils d'évaluation PANDORE (*profiling* et analyse de traces) a permis de vérifier en partie les explications qui avaient été formulées à propos des performances des noyaux d'applications étudiés au paragraphe 4.2. L'emploi de ces outils a également confirmé nos intuitions sur le comportement dynamique de ces programmes. Des expérimentations systématiques seront possibles après une intégration plus complète des outils dans l'environnement.

Chapitre 5

Bilan et perspectives

5.1 Bilan

Parmi les approches de programmation des APMD, la compilation par distribution de données apparaît prometteuse. Elle offre un mode de programmation séquentiel apprécié des programmeurs scientifiques et elle permet l'obtention de bonnes performances pour une large classe d'applications.

La mise en œuvre de cette approche nécessite des environnements complets permettant le développement et la mise au point de programmes efficaces et portables. Le projet PANDORE constitue un cadre de recherches sur la conception et la réalisation de tels environnements. À travers le travail effectué sur le prototype PANDORE, nous avons apporté, sous divers angles, une contribution conceptuelle et expérimentale à l'étude des environnements de compilation par distribution de données.

- Nous avons développé un support d'exécution complet pour le compilateur PANDORE. Cet exécutif permet l'exécution du code généré par le compilateur suivant un schéma de compilation de base et un schéma optimisé en offrant un certain nombre de services tels que la gestion de la distribution des données, le transfert d'ensembles de données, l'accès aux données distribuées ou l'allocation mémoire. On a montré que les optimisations mises en œuvre dans ces opérations étaient nécessaires pour l'obtention de performances globales des programmes.

  L'exécutif repose sur une bibliothèque de communication portable et efficace, la POM, dont nous avons contribué à la conception et au développement sur plusieurs plate-formes parallèles.

- Nous avons défini et mis en œuvre (dans le compilateur et dans l'exécutif) une gestion des tableaux distribués originale. Celle-ci est fondée sur la pagination des tableaux guidée par la distribution. En plus d'offrir un compromis temps d'accès / occupation mémoire intéressant, cette gestion des tableaux possède

plusieurs caractéristiques remarquables : l'accès aux éléments locaux et aux éléments reçus depuis d'autres processeurs se fait de façon uniforme, la définition de la représentation mémoire et du mécanisme d'accès ne dépend que du partitionnement du tableau et la contiguïté des éléments de tableaux est en partie conservée. Ces caractéristiques font que cette gestion de tableau est utilisable dans un contexte plus général que celui de PANDORE, elle peut être facilement adaptée à d'autres environnements de compilation par distribution de données. On peut également envisager son utilisation dans un contexte de programmation parallèle explicite, la gestion de tableau par pages offrant un mécanisme d'accès global transparent.

- Un travail d'expérimentation a été mené pour démontrer la validité des optimisations développées dans le compilateur et dans l'exécutif. Ces expérimentations ont porté sur plusieurs noyaux de programmes scientifiques et sur une application de propagation d'ondes sismiques. Elles ont permis de préciser la nature du travail à réaliser pour porter un algorithme séquentiel dans un langage data-parallèle tel que le langage PANDORE.
- L'évaluation des performances des programmes générés par les compilateurs est une étape nécessaire à la fois pour le programmeur d'application et pour les développeurs des systèmes de compilation. Durant ce travail de thèse, des techniques d'évaluation de performances des programmes produits par la compilation dirigée par les données ont été également étudiées. Nous avons jeté les bases d'un outil de mesure de performances intégré à l'environnement PANDORE. Celui-ci adopte une méthode de *profiling* ; il est fondé sur l'instrumentation du code généré, la collecte non intrusive de données statistiques et la visualisation graphique des résultats. Un des points clés de l'outil proposé est la relation entre les données de performances présentées et les objets du programme source, en particulier les données et leur distribution.

  Nous avons par ailleurs contribué à la définition et à la mise en œuvre de services d'observation dans la POM. Ces services permettent la génération et la collecte de traces d'exécution décrivant des événements pouvant être datés logiquement ou suivant un temps physique global. L'interfaçage que nous avons commencé à mettre en place entre l'exécutif PANDORE avec les services d'observation de la POM ouvrent un large champ d'étude des comportements des programmes compilés par distribution de données.
- Enfin, par la définition d'un cadre général pour décrire les schémas de compilation et par une mise en œuvre d'une version spécifique du compilateur et de l'exécutif PANDORE, nous avons amorcé une étude de la compilation de programmes séquentiels par distribution de données pour machines dotées d'une mémoire virtuelle partagée. Le principe de l'approche repose sur le placement judicieux des blocs de tableaux distribués dans la MVP. Ceci permet l'utilisation des mécanismes de la MVP pour l'accès rapide aux données et pour effectuer des communications vectorisées. Une première série d'expériences

avec la MVP Koan a permis de montrer la faisabilité de l'approche pour des cas réguliers.

5.2 Perspectives

Le travail que nous avons réalisé dans le cadre du projet PANDORE offre plusieurs perspectives de recherches à plus ou moins long terme. Nous développons dans les paragraphes qui suivent quelques unes d'entre elles ayant trait à la technique de pagination de tableaux distribués, aux outils d'évaluation de performances et à l'utilisation de la MVP.

- L'alignement, présent dans la plupart des langages data-parallèles et notamment HPF, apparaît comme une fonction de distribution utile pour le programmeur et permettant d'améliorer les performances des codes générés. Cette fonction n'a jusqu'à présent pas été implantée dans le système PANDORE, il semble toutefois que les techniques d'optimisation du schéma de compilation puissent être utilisées pour les tableaux alignés [79]. En ce qui concerne la représentation des tableaux distribués et leur l'accès, l'extension de la technique de pagination présentée au chapitre 3 ne pose pas de réels problèmes car les mécanismes présents dans le compilateur et dans l'exécutif peuvent être très largement réutilisés.

  Seuls les cas où toutes les dimensions sont alignées et distribuées nécessitent une adaptation des mécanismes de pagination. L'alignement entraîne un découpage des tableaux en blocs de tailles inégales ; cependant, la définition de la pagination décrite au paragraphe 3.2.2 peut être appliquée en considérant les tailles maximales des blocs. Le calcul du possesseur d'un élément implique un calcul plus compliqué qu'en l'absence d'alignement mais dans la mise en œuvre que nous avons proposée, ce calcul peut être fait entièrement à la compilation, la détermination du possesseur s'effectuant à l'exécution par consultation de tables. On garde ainsi, même pour les tableaux alignés, les avantages de la pagination et notamment un temps d'accès rapide.
- Le schéma de compilation optimisé actuellement mis en œuvre dans le compilateur PANDORE traite les nids de boucles réguliers (les bornes de boucles et les indices d'accès aux tableaux sont des fonctions affines des indices englobants). Pour prendre en compte des cas irréguliers, c'est-à-dire les cas où le compilateur ne peut pas déterminer statiquement les communications et les calculs à effectuer, on peut envisager l'utilisation de techniques d'optimisation dynamique comme l'inspecteur-exécuteur. À notre connaissance, cette technique est toujours utilisée avec le support fourni par la bibliothèque Parti [41] (voir l'exemple de Vienna Fortran au paragraphe 2.6.1.2) qui prend en charge également (à un coût non négligeable) la représentation des tableaux

distribués. La gestion des tableaux distribués par pages, et les optimisations de communications qu'elle permet, peut constituer un atout pour une mise en œuvre plus efficace de l'inspecteur-exécuteur.

  - Les accès peuvent rester globaux, et permettent donc l'économie de fonctions de conversion d'indices *global vers local*.
  - La notion de segment peut être utilisée de façon similaire à celle employée dans les cas réguliers pour éliminer les envois redondants, rendant inutile la construction de table de hachage.
  - Les communications directes sont possibles.
  - L'uniformité des accès entre les éléments locaux et reçus est par définition assurée.

- Le développement des outils d'évaluation de performance réalisé jusqu'à présent dans l'environnement PANDORE constitue une base pour l'expérimentation et l'étude de l'intégration dans un système de compilation par distribution de données de différentes techniques de mesures et d'interprétation de données de performances. Il s'agit d'une première étape qui offre plusieurs perspectives.

  Une phase d'expérimentation plus systématique est en cours à l'aide du *profiler* PANDORE, elle doit porter sur une série de programmes divers, l'objectif étant d'affiner les métriques choisies et de les compléter. La visualisation des données collectées doit être également enrichie. Une identification précise des indices de performances utiles d'une part pour le programmeur et d'autre part pour les développeurs du système de compilation est souhaitable. Parmi les pistes d'extension, on peut citer la prise en compte du coût en mémoire des représentations des tableaux et des optimisations de communication ainsi que la visualisation des mouvements de données sur une représentation graphique des tableaux distribués.

  Par ailleurs, l'automatisation de la production de traces d'exécution des programmes PANDORE par l'utilisation de la POM peut être développée, l'objectif étant à terme de disposer d'un ensemble complet d'outils interactifs autorisant le pilotage de l'instrumentation des programmes et de l'observation des exécutions distribuées, en relation constante avec le programme source.

- Les expériences que nous avons menées sur l'utilisation de la MVP Koan comme support d'exécution de codes générés par compilation dirigée par les données ont montré que l'on pouvait obtenir de bonnes performances dans les cas réguliers. Les premiers résultats que nous avons obtenus peuvent laisser penser que le manque de souplesse d'une MVP par rapport à une machine à messages a tendance à dégrader les performances, en forçant par exemple le grain des communications ou des allocations mémoire. Il convient d'affiner ce jugement par des expérimentations plus poussées, sur une gamme plus large de codes. L'utilisation de MVP câblées apporterait sans doute des informations pertinentes en la matière.

Par ailleurs, les expériences menées peuvent avoir un impact sur la définition des services offerts par les futures MVP. Les services proposés par la MVP Koan ont été conçus essentiellement pour une programmation manuelle. Ils ne sont donc pas forcément adaptés à une utilisation dans un code généré automatiquement. C'est par exemple le cas pour la diffusion de pages qui a été prévue pour un schéma 1 producteur–n consommateurs alors qu'elle est utilisée dans un cadre plus large dans le code produit par le compilateur PANDORE. Les services fournis par la MVP pourraient être adaptés, souvent sans grand bouleversement, pour prendre en compte ce genre d'utilisations. Dans le même esprit, on peut envisager que des accès plus fins aux mécanismes de la MVP soient possibles, permettant par exemple l'accès aux possesseurs des pages, le contrôle de l'allocation au sein de la mémoire virtuelle, voire le contrôle de la cohérence.

Les travaux décrits dans cette thèse laissent entrevoir la possibilité d'intégrer efficacement la compilation par distribution du contrôle et la compilation par distribution de données au sein d'un même système de compilation, le choix de la stratégie de compilation pouvant être guidé par les indications de l'utilisateur ou la forme des accès (réguliers ou irréguliers par exemple).

Ceci peut être un premier pas vers la définition d'environnements de programmation des APMD à la fois généraux, efficaces et portables, environnements dont la disponibilité est un élément essentiel à la diffusion des APMD.

Bibliographie

[1] A. Aho, R. Sethi et J. Ullman. – *Compilateurs - Principes, techniques et outils*. – InterEditions, 1989.

[2] R. Allen et K. Kennedy. – Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages*, vol. 9, n° 4, octobre 1987, pp. 491–542.

[3] F. André, O. Chéron, M. Le Fur, Y. Mahéo et J.-L. Pazat. – Programmation des machines à mémoire distribuée par distribution des données : langages et compilateurs. *Techniques et Sciences Informatiques*, vol. 12, Numéro spécial "Langages à parallélisme de données", n° 5, octobre 1993, pp. 563–596.

[4] F. André, O. Chéron et J.-L. Pazat. – Compiling Sequential Programs for Distributed Memory Parallel Computers with Pandore II. *In : International Worshop on Environments and Tools for Parallel Scientific Computing*, Saint Hilaire du Touvet, septembre 1993. pp. 293–308, North Holland.

[5] F. André, O. Chéron, J.-L. Pazat et H. Thomas. – Efficient Code Generation for Distributed Memory Machines. *In : International Conference on Parallel Computing*, Londres, Angleterre, septembre 1991. Advances in Parallel Computing n° 4, North Holland.

[6] F. André, M. Le Fur, Y. Mahéo et J.-L. Pazat. – The Pandore Data Parallel Compiler and its Portable Runtime. *In : International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe '95*, Milan, Italie, mai 1995. LNCS n° 919, Springer Verlag.

[7] F. André, M. Le Fur, Y. Mahéo et J.-L. Pazat. – Parallelization of a Wave Propagation Application using a Data Parallel Compiler. *In : 9th International Parallel Processing Symposium*, Santa Barbara, Californie, avril 1995.

[8] F. André, J.-L. Pazat et H. Thomas. – Data Distribution with Pandore. *In : 5th Distributed Memory Computing Conference*, Charleston, Caroline du Sud, avril 1990.

[9] Y. Arrouye. – *Performance Evaluation of Parallel Systems : the Scope Extensible Interactive Environment.* – Rapport Apache 15, Institut d'Informatique et de Mathématiques Appliquées de Grenoble, janvier 1995.

[10] V. Balasundaram. – A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools : The Data Access Descriptor. *Journal of Parallel and Distributed Computing*, vol. 9, avril 1990, pp. 154–170.

[11] V. Balasundaram, G. Fox, K. Kennedy et U. Kremer. – An Interactive Environnement for Data Partitioning and Distribution. *In : 5th Distributed Memory Computing Conference*, Charleston, Caroline du Sud, avril 1990.

[12] V. Balasundaram, G. Fox, K. Kennedy et U. Kremer. – A Static Performance Estimator to Guide Data Partitioning Decisions. *In : 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, Virginia, juin 1991.

[13] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley et J. Subhlock. – The Parascope Editor : an Interactive Parallel Programming Tool. *In : Supercomputing'89*, Reno, Nevada, novembre 1989.

[14] T. Ball et J. R. Larus. – *Optimally Profiling and Tracing Programs.* – Rapport technique 1031, Computer Science Department - University of Wisconsin, Madison, septembre 1991.

[15] C. Bareau. – *Distribution automatique de programmes séquentiels : étude structurelle et expérimentale.* – Thèse de doctorat, IFSIC / Université de Rennes I, juillet 1995.

[16] C. Bareau, B. Caillaud, C. Jard et R. Thoraval. – Correctness of Automated Distribution of Sequential Programs. *In : Parallel Architectures and Languages Europe*, juin 1993. LNCS n° 694, pp. 517–528, Springer Verlag.

[17] C. Bareau, Y. Mahéo et J.-L. Pazat. – Parallel Program Performance Debugging with the Pandore II Environment. *In : International Conference on Parallel Computing*, Grenoble, septembre 1993. North Holland.

[18] A. Beguelin, G.A. Geist, W. Jiang, R. Manchek, K. Moore et V. Sunderam. – *The PVM Project.* – Rapport technique, Oak Ridge National Laboratory, Tennessee, février 1993.

[19] T. Bemmerl, A. Bode, P. Braun, O. Hansen, P. Liksch et R. Wismüller. – *TOPSYS - Tools for Parallel Systems - User's Overview and User's Manuals.* – Rapport technique TUM-I9047, Institut für Informatik der Technischen Universität München, Allemagne, 1990.

[20] T. Bemmerl et T. Ludwig. – PATOP for Performance Tuning of Parallel Programs. *In : Joint International Conference on Vector and Parallel Processing, CONPAR 90 -VAPP IV*, Zurich, Suisse, septembre 1990. LNCS n° 457, pp. 840–851, Springer Verlag.

[21] A.-D. Benalia. – *Un environnement graphique pour le développement de codes HPF.* – Rapport technique ERA-153, Laboratoire d'Informatique Fondamentale de Lille, mai 1994.

[22] A.-D. Benalia, J.-L. Dekeyser et P. Marquet. – HelpDraw Graphical Environment : A Step Beyond Data-Parallel Programming Languages. *In : Human-Computer Interaction : Sofware and Hardware Interfaces*, 1993, pp. 591–596, Elsevier Science Publishers B.V.

[23] S. Benkner. – *Vienna Fortran 90 and its Compilation.* – Thèse de doctorat, Université de Vienne, Autriche, septembre 1994.

[24] S. Benkner, P. Brezany et H. Zima. – Compiling High Performance Fortran in the Prepare Environment. *In : 4th International Workshop on Compilers for Parallel Computers*, Delft, Pays-Bas, décembre 1993.

[25] S. Benkner, P. Brezany et H. Zima. – Processing Array Statements and Procedure Interfaces in the PREPARE High Performance Fortran Compiler. *In : 5th International Conference on Compiler Construction*, avril 1994. LNCS n° 786, pp. 324–338, Springer-Verlag.

[26] F. Bodin, L. Kervella et T. Priol. – Fortran-S : a Fortran Interface for Shared Virtual Memory Architectures. *In : Supercomputing'93*, Portland, Oregon, novembre 1993.

[27] T. Brandes. – Compiling Data Parallel Programs to Message Passing Programs for Massively Parallel MIMD Systems. *In : Programming Models for Massively Parallel Computers*, Berlin, septembre 1993, IEEE, Computer Society Press.

[28] T. Brandes et F. Zimmermann. – *ADAPTOR Distributed Array Library.* – Adaptor Report 4, GMD, Sankt Augustin, Allemagne, mars 1994.

[29] P. Brezany, O. Chéron, K. Sanjari et E. van Konijnenburg. – Processing Irregular Codes Containing Arrays with Multi-Dimensional Distributions by the PREPARE HPF Compiler. *In : International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe '95*, Milan, Italie, mai 1995. LNCS n° 919, Springer Verlag.

[30] P. Brezany, M. Gerndt et V. Sipkova. – *SVM Support in the Vienna Fortran Compilation System.* – Rapport technique KFA-ZAM-IB-9401, KFA Juelich, Allemagne, 1994.

[31] V. Van Dongen C. Bonello, G. Hurteau et G. R. Gao. – *EPPP – An Environment for Portable Parallel Programming.* – Rapport technique EPPP-94/04-12, Centre de Recherche Informatique de Montréal, Canada, juillet 1994.

[32] B. Caillaud. – *Contribution à la modélisation du SPMD : distribution asynchrone d'automates.* – Thèse de doctorat, IFSIC / Université de Rennes I, juin 1994.

[33] D. Calahan et K. Kennedy. – Compiling Programs for Distributed Memory Multiprocessors. *The Journal of Supercomputing*, vol. 2, octobre 1988, pp. 151–169.

[34] R. Calkin, R. Hempel, H.-S. Hoppe et P. Wypior. – Portable Programming with the PARMACS Message-Passing Library. *Parallel Computing*, 1994.

[35] B. Chapman, P. Mehrotra et H. Zima. – *Vienna Fortran : A Fortran Language Extension for Distributed Memory Multiprocessors.* – Rapport technique 91-72, ICASE, septembre 1991.

[36] S. Chatterjee, J.R. Gilbert, F.J.E. Schreiber et S.H. Teng. – Generating Local Adresses and Communication Sets for Data-Parallel Program. *In : 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, juillet 1993, pp. 149–158.

[37] O. Chéron. – *Pandore II : un compilateur dirigé par la distribution des données.* – Thèse de doctorat, IFSIC / Université de Rennes I, juillet 1993.

[38] J. Choi, J. Dongarra, R. Pozo et D.W. Walker. – Scalapack : a Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. *In : Frontiers of Massively Parallel Computers*, Mc Lean, Virginia, octobre 1992.

[39] F. Darema-Rogers, V. Norton et G. Pfister. – *Using a Single-Program-Multiple-Data Computational Model for Parallel Execution of Scientific Applications.* – Rapport technique RC11552, IBM T.J. Watson Research Center, Yorktown Heights, New-York, novembre 1985.

[40] R. Das, R. Ponnusamy, J. Saltz et D. Mavriplis. – Distributed Memory Compiler Methods for Irregular Problems – Data Copy Reuse and Runtime Partitioning. *In : 3rd Workshop on Compilers for Parallel Computers*, Vienne, Autriche, juillet 1992, pp. 173–206.

[41] R. Das, J. Saltz et H. Berryman. – *A Manual for PARTI Runtime Primitives - Revision 1.* – Rapport technique, ICASE Langley Research Center, Hampton, Virginie, décembre 1992.

[42] T. Fahringer. – *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. – Thèse de doctorat, Université de Vienne, Autriche, septembre 1993.

[43] T. Fahringer, R. Blasko et H. Zima. – Automatic Performance Prediction to Support Parallelization of Fortran Programs for Massively Parallel Systems. *In : International Conference on Supercomputing*, Washington, D.C., juillet 1992, pp. 347–356, ACM press.

[44] T. Fahringer et H. Zima. – A Static Parameter Based Performance Prediction Tool for Parallel Programs. *In : International Conference on Supercomputing*, Tokyo, Japon, juillet 1993, ACM press.

[45] C.J. Fidge. – Logical Time in Distributed Computing Systems. *IEEE Computer*, vol. 24, n° 8, août 1991, pp. 28–33.

[46] M.J. Flynn. – Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, vol. C-21, n° 9, septembre 1972, pp. 948–960.

[47] High Performance Fortran Forum. – *High Performance Fortran Language Specification, Version 1.0*. – Rapport technique, Rice University, Houston, Texas, 1993.

[48] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C-W. Tseng et M. Wu. – *The Fortran D Language Specification*. – Rapport technique TR90141, Center for Research on Parallel Computation, Rice University, Houston, Texas, avril 1991.

[49] P. Fritzson et N. Andersson. – Generating Parallel Code from Equations in the ObjectMath Programming Environment. *In : 2nd International Austrian Center for Parallel Computation Conference*, Gmunden, Autriche, octobre 1993. LNCS n° 734, pp. 217–232, Springer-Verlag.

[50] G.A. Geist, M.T. Heath, B.W. Peyton et P.H. Worley. – *A User's Guide to PICL - A Portable Instrumented Communication Library*. – Rapport technique ORNL/TM-11616, Oak Ridge National Laboratory, Tennessee, mai 1992.

[51] M. Gerndt. – *Automatic Parallelization for Distributed-Memory Multiprocessor Systems*. – Thèse de doctorat, Université de Bonn, janvier 1990.

[52] G.H. Golub et C.F.Van Loan. – *Matrix Computation*. – The John Hopkins University Press, 1990, 2^{nde} édition.

[53] F. Guidec. – *Un cadre conceptuel pour la programmation par objets des architectures parallèles distribuées : application à l'algèbre linéaire*. – Thèse de doctorat, IFSIC / Université de Rennes I, juin 1995.

[54] F. Guidec et Y. Mahéo. – POM : une machine virtuelle parallèle incorporant des mécanismes d'observation. *Calculateurs Parallèles*, vol. 7, Numéro thématique "Environnements d'exécution de programmes parallèles", n° 2, juin 1995, pp. 101–118.

[55] F. Guidec et Y. Mahéo. – POM : a Parallel Observable Machine. *In : International Conference on Parallel Computing*, Gand, Belgique, septembre 1995.

[56] F. Guidec et Y. Mahéo. – POM : a Virtual Parallel Machine Featuring Observation Mechanisms. *In : International Conference on High Performance Computing*, New Delhi, Inde, décembre 1995.

[57] M. Gupta. – *Automatic Data Partitioning on Distributed Memory Multicomputers*. – Thèse de doctorat, University of Illinois at Urbana-Champaign, septembre 1992.

[58] Y. Haddad. – *Performance dans les systèmes répartis : des outils pour les mesures*. – Thèse de doctorat, Université de Paris-Sud, Centre Orsay, Paris, septembre 1988.

[59] R. von Hanleden, K. Kennedy, C. Koelbel, R. Das et J. Saltz. – Compiler Analysis for Irregular Problems in Fortran D. *In : 3rd Workshop on Compilers for Parallel Computers*, Vienne, Autriche, juillet 1992.

[60] M. Heath et J. Etheridge. – Vizualizing Performance of Parallel Programs. *IEEE Software*, vol. 5, n° 8, septembre 1991, pp. 29–39.

[61] M. Heath et J. Etheridge. – *Vizualizing Performance of Parallel Programs*. – Rapport technique ORNL/TM-11813, Oak Ridge National Laboratory, Tennesse, mai 1991.

[62] S. Hiranandani, K. Kennedy, J.M. Crummy et A. Sethi. – *Advanced Compilation Techniques for Fortran D*. – Rapport technique TR93338, Center for Research on Parallel Computation, Rice University, Houston, Texas, octobre 1993.

[63] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer et C.W. Tseng. – *An Overview of the Fortran D Programming System*. – Rapport technique TR91121, Center for Research on Parallel Computation, Rice University, Houston, Texas, mars 1991.

[64] S. Hiranandani, K. Kennedy et C-W. Tseng. – Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, vol. 35, n° 8, août 1992.

[65] G. Hurteau, A. Singh, M. Hancu, V. Van Dongen et H. Hum. – A Performance Debugger for a Language Supporting Data Distribution Primitives. *In : International Conference on Parallel Processing*, Chicago, Illinois, août 1994.

[66] Intel. – *Paragon^{TM} OSF/1 User's Guide.* – Technical Report 312489-001, Intel Corporation, avril 1993.

[67] F. Irigoin, C. Ancourt, F. Coelho et R. Keryell. – A Linear Algebra Framework for Static HPF Code Distribution. *In : 4th International Workshop on Compilers for Parallel Computers*, Delft, Pays-Bas, décembre 1993.

[68] C. Jard et G.-V. Jourdan. – *Dependency Tracking and Filtering in Distributed Computations.* – Publication interne 851, Irisa, août 1994.

[69] L. Jerid, F. André, O. Chéron, J.-L. Pazat et T. Ernst. – *HPF to C-Pandore Translator.* – Rapport technique 2283, Inria, mai 1994.

[70] J.-M. Jézéquel. – EPEE : an Eiffel Environment to Program Distributed Memory Parallel Computers. *Journal of Object Oriented Programming*, vol. 6, n° 2, mai 1993, pp. 48–54.

[71] G.-V. Jourdan. – *L'analyse d'exécutions réparties en utilisant la théorie de l'ordre.* – Thèse de doctorat, IFSIC / Université de Rennes I, octobre 1995.

[72] Kendall Square Research, Waltham, Massachusetts. – *KSR Technical Summary*, 1992.

[73] C. Kesselman. – *Tools and Techniques for Performance Measurment and Performance Improvement in Parallel Programs.* – Thèse de doctorat, University of California at Los Angeles, juillet 1991.

[74] J. P. Kitajima. – *Modèles quantitatifs d'algorithmes parallèles.* – Thèse de doctorat, Institut National Polytechnique de Grenoble, octobre 1994.

[75] J. P. Kitajima, C. Tron et B. Plateau. – ALPES : a Tool for the Performance Evaluation of Parallel Programs. *In : Environments and Tools for Parallel Scientific Computing*, 1993. pp. 213–228, Elsevier Science Publishers B.V.

[76] Z. Lahjomri. – *Conception et évaluation d'un mécanisme de mémoire virtuelle partagée sur une machine multiprocesseur à mémoire distribuée.* – Thèse de doctorat, IFSIC / Université de Rennes I, janvier 1994.

[77] Z. Lahjomri et T. Priol. – Koan : a Shared Virtual Memory for the iPSC/2 Hypercube. *In : 2nd Joint International Conference on Vector and Parallel Processing, CONPAR 92 - VAPP V*, Lyon, septembre 1992. LNCS n° 634, pp. 441–452, Springer Verlag.

[78] C. Lawson, R. Hanson D. Kincaid et F. Krogh. – Basic Linear Algebra Subprograms for Fortran. *ACM Transactions on Math. Software*, vol. 14, 1989, pp. 308–325.

[79] M. Le Fur. – *Compilation de boucles dirigée par la distribution des données.* – Thèse de doctorat, IFSIC / Université de Rennes I, juillet 1995.

[80] M. Le Fur, J-L. Pazat et F. André. – *Static Domain Analysis for Compiling Commutative Loop Nests.* – Rapport de Recherche 2067, Inria, septembre 1993.

[81] M. Le Fur, J.-L. Pazat et F. André. – An Array Partitioning Analysis for Parallel Loop Distribution. *In : International Conference on Parallel Processing, Euro-Par'95*, Stockholm, Suède, août 1995. LNCS, Springer Verlag.

[82] J. Li et M. Chen. – Compiling Communication-Efficient Programs for Massively Parallel Machines. *Journal of Parallel and Distributed Computing*, vol. 2, n° 3, juillet 1991, pp. 361–376.

[83] K. Li et R. Shaefer. – A Hypercube Shared Virtual Memory System. *In : International Conference on Parallel Processing*, University Park, Pennsylvanie, 1989, pp. 121–131.

[84] Y. Mahéo et J.-L. Pazat. – Distributed Array Management for HPF Compilers. *In : High Performance Computing Symposium*, Montréal, Canada, juillet 1995.

[85] Y. Mahéo et J.-L. Pazat. – Distributed Array Management Scheme for Data-Parallel Compilers. *In : 5th Workshop on Compilers for Parallel Computers*, Malaga, Espagne, mai 1995.

[86] E. Maillet. – *Tape/PVM an Efficient Performance Monitor for PVM Applications – User Guide.* – Institut d'Informatique et de Mathématiques Appliquées de Grenoble, mars 1995.

[87] E. Maillet et C. Tron. – On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 1994.

[88] L. Massari et Y. Mahéo. – *Performance Analysis of Automatically Generated Data-Parallel Programs.* – Publication interne, Irisa, 1995. À paraître.

[89] Message Passing Interface Forum. – *Document for a Standard Message-Passing Interface.* – Technical Report CS-93-214, University of Tennessee, novembre 1993.

[90] Q. Ning, V. Van Dongen et G. R. Gao. – Automatic Data and Computation Decomposition for Distributed Memory Machines. *In : 28th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, janvier 1995.

[91] P. Pierce. – The NX/2 Operating System. *In : 3rd Conference on Hypercube Concurrent Computers and Applications*, Pasadena, Californie, janvier 1988, pp. 384–390.

[92] D. Reed, R. Aydr, T. Madhyasta, R. Noe, K. Shields et B. Schwartz. – *An Overview of the Pablo Performance Analysis Environnement*. – Rapport technique, Department of Computer Science, University of Illinois, novembre 1992.

[93] J. Saltz, K. Crowley, R. Mirchandaney et H. Berryman. – Run-Time Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, vol. 8, avril 1990.

[94] N. Tawbi. – *Parallélisation automatique : estimation des durées d'exécution et allocation statique de processeurs*. – Thèse de doctorat, Université de Paris VI, septembre 1991.

[95] H. Thomas. – *Une approche de la compilation de programmes séquentiels pour machine à mémoire distribuée*. – Thèse de doctorat, IFSIC / Université de Rennes I, juin 1990.

[96] H. Thomas, H. Sips et E. Paalvast. – A Taxonomy of User-Annotated Programs for Distributed Memory Computers. *In : International Conference on Parallel Processing*, University Park, Pennsylvanie, août 1992.

[97] C.-W. Tseng. – *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. – Thèse de doctorat, Rice University, Houston, Texas, janvier 1993.

[98] M. van Riek, B. Tourancheau et X.-F. Vigouroux. – *Monitoring of Distributed Memory Multicomputer Programs*. – Rapport technique TR93441, Center for Research on Parallel Computation, Rice University, Houston, Texas, 1993.

[99] M. Wolfe. – *Optimizing Supercompilers for Supercomputers*. – Pitman Publishing, 1989, *Research Monographs in Parallel and Distributed Computing, MIT Press*.

[100] P. H. Worley. – *A New PICL Trace File Format*. – Rapport technique ORNL/TM-12125, Oak Ridge National Laboratory, Tennessee, octobre 1992.

[101] H. Zima, H.-J. Bast et M. Gerndt. – SUPERB : A Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, n° 6, 1988, pp. 1–18.

[102] H. Zima et B. Chapman. – *Supercompilers for Parallel and Vector Computers.* – ACM Press, 1990.

[103] H. Zima et B. Chapman. – *Compiling for Distributed-Memory Systems.* – Rapport technique ACPC/TR 92-17, Austrian Center for Parallel Computation, Université de Vienne, novembre 1992.

Résumé

La difficulté de programmation des architectures parallèles à mémoire distribuée est un obstacle à l'exploitation de leur puissance de calcul potentielle. Parmi les différentes approches proposées pour pallier cette difficulté, celle de la compilation dirigée par les données semble prometteuse, notamment dans le domaine du calcul scientifique. Le programme source, exprimé par exemple en HPF, est un programme séquentiel impératif dans lequel il est précisé comment sont réparties les données sur les processeurs ; le compilateur dérive un code parallèle en distribuant le contrôle d'après la distribution des données.

La mise en œuvre de cette approche nécessite le développement d'environnements complets. Cette thèse présente le travail réalisé dans le cadre d'un environnement de ce type : l'environnement PANDORE. Nous nous sommes intéressés à la conception et la réalisation d'un exécutif portable et efficace qui doit être associé au compilateur ainsi qu'à l'évaluation des performances des programmes générés.

Après avoir situé l'approche de la compilation par distribution de données dans le contexte plus large de la programmation des machines parallèles à mémoire distribuée, nous définissons des opérations de haut niveau qui permettent la description des schémas de compilation et la prise en compte des optimisations. Deux types de machines cibles sont considérés, d'une part des machines à messages et d'autre part des machines disposant d'un mécanisme de mémoire virtuelle partagée. Les points clés de la mise en œuvre des opérations dans le compilateur et l'exécutif sont abordés. Nous insistons plus particulièrement sur la gestion des données distribuées et sur les optimisations des communications à l'exécution. Une mise en œuvre réalisée dans l'environnement PANDORE est ensuite détaillée. L'évaluation des performances des programmes est également étudiée, dans un premier temps par une série d'expérimentations sur plusieurs applications et dans un deuxième temps par la définition d'outils de mesure et de visualisation adaptés à la compilation par distribution de données.