Calcul distribué de champs de données

Jean-Louis Giavitto[1] & Olivier Michel[2]

1 : CNRS – LRI, Université de Paris-Sud,
91405 Orsay Cedex
giavitto@lri.fr

2 : LaMI, Université d'Évry Val d'Essone
Boulevard des coquibus, 91025 Evry Cedex
michel@lami.univ-evry.fr

Résumé : `FieldBroker` est une architecture logicielle client-serveur dédiée à l'évaluation de requêtes de calcul sur les champs de données. Un champ de données est une structure qui étend la notion de tableau fonctionnel parallèle et qui permet la représentation de structures irrégulières. L'application se comporte comme un client qui délègue à un serveur les calculs sur les champs. Celui-ci, après transformation et optimisation de la requête, répartit les traitements à effectuer sur un ensemble de processus esclaves. Ces esclaves utilisent des processus légers afin de recouvrir les phases de calcul et de communication. Une des originalités de `FieldBroker` est de raisonner et d'implémenter ce processus d'évaluation à travers une tour de trois langages.

Mots-clés : parallélisme et champs de données, serveur, maître-esclave distribué, `JAVA`, évaluation d'expressions fonctionnelles de tableaux.

1. Contexte de l'étude

$8_{1/2}$ est un langage déclaratif pour la simulation parallèle de systèmes dynamiques (DS). Un programme $8_{1/2}$ est un système d'équations qui définit la trajectoire d'un DS (la suite des états du système dans le temps). L'exécution d'un programme $8_{1/2}$ consiste à résoudre le système d'équations. Le processus de compilation [12, 5] transforme ce problème en celui du calcul d'un ensemble d'expressions portant sur des tableaux.

Le compilateur fournit cet ensemble d'expressions sous la forme d'un graphe à flot de données, sans cycle (les définitions récursives de tableaux sont transformées par le compilateur en calculs itératifs). Les arcs du graphes correspondent à des tableaux fonctionnels : la valeurs des éléments d'un tableau est immuable. Les sommets de ce graphe correspondent à des opérations *globales* de tableaux : extension d'opérateur scalaire (`map`), réduction (`fold`), balayage (`scan`), opération de réarrangement (`permute`), etc. Ces opérations sont bien connues [33] ; voici quelques exemples avec deux vecteurs $A = \{3, 2, 1\}$ et $B = \{4, 5, 6\}$:

$$\begin{array}{lclclcl} \texttt{map}[+]\,A\,B & \Rightarrow & \{7,7,7\} & \qquad & \texttt{fold}[+]\,A & \Rightarrow & \{6\} \\ \texttt{scan}[+]\,A & \Rightarrow & \{3,5,6\} & \qquad & \texttt{permute}\,A\,B & \Rightarrow & \{6,5,4\} \end{array}$$

L'évaluation *data-parallèle* de ces expressions consiste à répartir les éléments des tableaux sur les processeurs. Les calculs correspondant à l'élaboration de la valeur d'un élément sont effectués par le processeur qui stocke cet élément (cette règle, connue sous le nom de « *owner computes rule* » est classiquement utilisée en calcul parallèle).

L'évaluation parallèle d'un programme 81/2 revient donc à déterminer une distribution des éléments d'un ensemble de tableaux sur un ensemble de processeurs et à ordonnancer l'ensemble des calculs correspondants. Ce problème est représenté dans la figure 1.

Nous avons développé plusieurs heuristiques [25, 26] afin de résoudre efficacement le problème de la distribution et de l'ordonnancement des calculs parallèles de tableaux induits par la compilation d'un programme 81/2. Ce modèle d'exécution a été prototypé sur une `IBM SP2` [24]. Les résultats sont encourageants, mais plusieurs critiques importantes peuvent être adressé à cette approche de l'évaluation parallèle de tableaux fonctionnels :

- La taille du code généré est trop importante. Il faut générer autant de programmes distincts qu'il y a de colonne dans le diagramme de Gantt. Pour une machine de 4 processeurs, nos exemples de tests généraient facilement une dizaine de méga-octets de code.
- Les performances (temps d'exécution, nombre de messages produits, taille mémoire occupée) sont très sensibles aux paramètres des stratégies de placements et d'ordonnancement (granularité de la distribution, estimation correcte des coûts de communications, etc.). L'ajustement des paramètres rend ce modèle d'exécution difficilement portable.
- Ce modèle d'exécution est un modèle statique, adapté à l'évaluation de programmes dont on peut prévoir à la compilation tous les calculs de tableaux. Il ne permet pas l'intégration de structures de données dynamiques.
- Ce modèle d'exécution est adapté aux architectures parallèles « classiques » dont tous les nœuds sont homogènes.

La généralisation de la programmation parallèle à toute sorte d'applications (réalité virtuelle, fouille dans les bases de données, expérimenta-

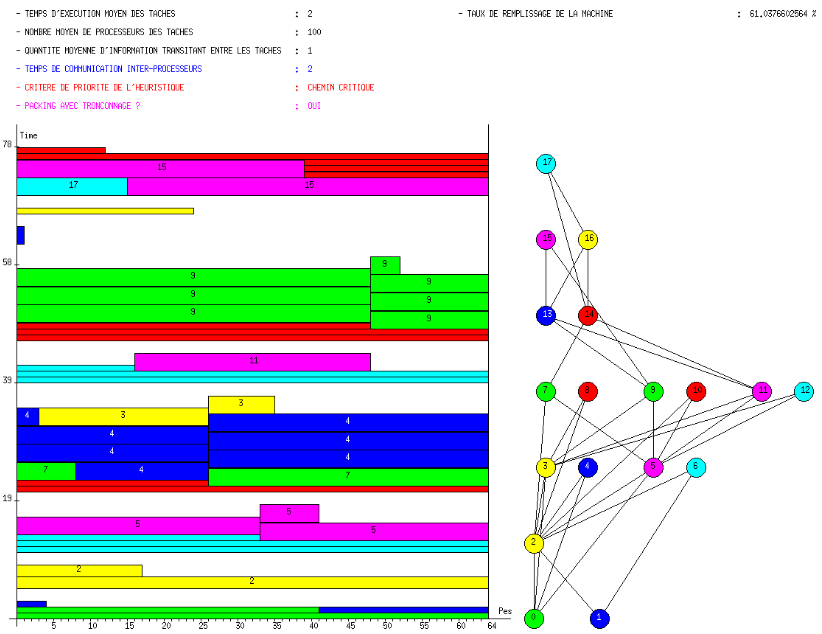

FIG. 1 – *Cette image est une capture d'écran de l'outil que nous avons développé pour la distribution et l'ordonnancement d'expressions de tableaux [25, 26]. Un ensemble d'expressions de tableaux est représenté à droite sous la forme d'un graphe à flot de données. Un tableau résultat est associé à chaque sommet S et peut se représenter sous la forme d'un rectangle nombre-de-processeurs-nécessaires×temps-d'exécution de l'opération S. La distribution et l'ordonnancement des calculs de tableaux revient alors à placer ces rectangles dans l'espace processeurs×temps du calcul. Il est possible de partitionner le rectangle correspondant à un sommet en sous-rectangles correspondant à des sous-calculs, afin d'utiliser au mieux les ressources de la machine parallèle. Le résultat, un diagramme de Gantt, est figuré à gauche.*

tions numériques...) pousse fortement à utiliser des réseaux de stations de travail hétérogènes comme des machines parallèles « à faible coût ». De plus, ces nouvelles applications et la sophistication croissante des algorithmes requièrent de plus en plus des structures de données dynamiques, ce qui nécessite aussi plus de support de la part des exécutifs. Enfin, les applications se construisent de plus en plus comme la collaboration entre différents programmes (solveur, optimiseur, ...), ce qui nécessite le développement de nouvelles architectures logicielles facilement extensibles. Ces différents points réclament tous le développement d'un nouveau modèle d'exécution pour 81/2, dynamiquement adaptable, portable et robuste, sur des plate-formes distribuées et hétérogènes.

Dans la section suivante, nous proposons un nouveau modèle d'exécution, `FieldBroker`, qui répond à ces problèmes tout en permettant de calculer avec une structure de données plus générale que les tableaux : les champs de données sur $\mathbb{Z}^n$.

`FieldBroker` est une architecture client-serveur dédiée au calcul sur les champs de données sur $\mathbb{Z}^n$ [13]. Une application (par exemple la résolution d'un programme 81/2) se présente comme un processus client qui demande par requête au serveur, des calculs sur des champs. `FieldBroker` a été développé afin de fournir une machine virtuelle portable pour l'évaluation des programmes 81/2 dynamique [27] et pour la résolution des définitions récursives de champ de données sur un groupe abélien [14].

Cependant, `FieldBroker` a aussi pour but de valider la viabilité de l'approche client-serveur dans le cadre des applications data-intensives et d'évaluer combien cette architecture permet une intégration harmonieuse de concepts et de techniques développés séparément dans le cadre des langages fonctionnels et dans le cadre du parallélisme de données.

2. Une machine virtuelle distribuée pour le calcul de champs de données

2.1. La notion de champ de données

Un champ de données est une fonction d'un ensemble d'index dans un ensemble de valeurs. La notion de champ de données généralise la notion de tableau puisqu'on peut voir un tableau de dimension d comme une fonction stricte de $\{1 \ldots n_1\} \times \cdots \times \{1 \ldots n_d\}$. Les champs de données étendent les tableaux en considérant des fonctions *partielles* sur $\mathbb{Z}^d$ et/ou en considérant des ensembles d'index plus exotiques. Nous nous restreignons dans ce travail aux champs de données sur $\mathbb{Z}^n$.

Ce point de vue permet d'exprimer des calculs sur les champs de données de manière intensionnelle en considérant des algèbres de fonctions : les opérateurs sur les champs de données sont des fonctions d'ordre supérieure et les indices des éléments n'apparaissent pas du tout.

Mais le but n'est évidement pas d'implémenter cette structure de données comme une fonction informatique (i.e. par une règle de calcul qu'il faut dérouler chaque fois qu'on veut accéder à un élément). Il faut plutôt lister en mémoire les associations indice-élément, ce qui correspond au point de vue extensionnel sur une fonction. On peut donc dire que le slogan des champs de données est : « exprimer des fonctions intensionnellement mais les implémenter extensionnellement ».

La notion de champ de données est une notion ancienne en informatique : elle apparaît déjà dans la notion d'*équation récurrente uniforme* et remonte donc au moins à [18]. Le terme de « champ de données » semble référencé la première fois dans [36] autour du projet `Crystal` [3]. En fait, la notion de champ de données est une notion familière dans le domaine du calcul systolique particulièrement dans les approches de haut-niveau comme dans le langage `Alpha` [10], et quand les techniques développées sont utilisées pour mieux compiler les langages fonctionnels [30].

Le rapprochement explicite entre la notion de champ de données et le parallélisme de données a été effectué par Lisper [21] (le tutoriel [22] est une bonne introduction à la problématique). Notons cependant que la notion de *xappings* dans [34] est loin d'être étrangère au rapprochement entre parallélisme de données et fonction. Ce rapprochement est aussi à la base du projet `PEI` [35] qui se concentre sur les propriétés géométriques du domaine des index et qui développe une approche transformationnelle de la programmation data-parallèle.

2.2. Une architecture logicielle client/serveur-maître/esclaves

L'architecture logicielle du serveur est illustrée par la figure 2. Trois niveaux sont distingués dans une tour de machines virtuelles.

1. Le **serveur** traite les requêtes correspondant à des calculs sur des *fonctions sur* $\mathbb{Z}^n$. Il est responsable de la parallélisation et de la synchronisation entre les traitements des requêtes qui proviennent d'un même client ou de clients différents.
2. Le **maître** effectue des opérations impliquant des ensembles de tableaux (fonctions définies sur une région rectangulaire de $\mathbb{Z}^n$). Ce niveau est responsable de plusieurs optimisations de haut niveau et décide aussi de la répartition des données et des calculs sur les esclaves.
3. Les **esclaves** réalisent des calculs séquentiels sur des données linéairement rangées en mémoire. Ils sont pilotés par des ordres reçus du maître. Ces ordres sont de deux sortes : soit un calcul à effectuer, soit des données à transmettre à un autre esclave. Pour masquer les latences des communications, les calculs, les réceptions et les émissions sont réalisés par des processus légers.

Les bénéfices attendus de l'architecture proposée sont les suivants.

- **Accessibilité et indépendance vis à vis des clients :** le découplage des applications et des calculs sur les champs permet

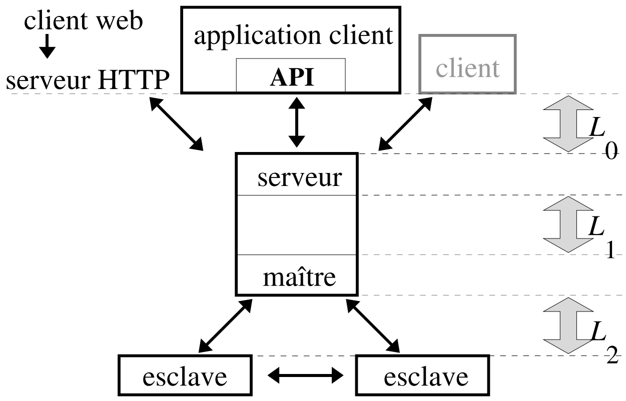

FIG. 2 – *Une architecture client/serveur-maître/esclaves-multithread pour l'évaluation data-parallèle des calculs sur les champs. Chaque « boîte » est un processus autonome. Les esclaves sont distribués sur des processeurs différents. L'architecture est modélisée par les trois algèbres $\mathcal{L}_0$, $\mathcal{L}_1$ et $\mathcal{L}_2$ de la figure 3.*

au serveur d'évoluer indépendamment. De plus, le serveur est écrit en `JAVA`, ce qui permet d'envoyer des requêtes, soit à travers une API, soit interactivement à travers des applets (client WEB). Dans ce dernier cas, le serveur apparaît comme « une calculette sur les champs de données ».

- **Services autonomes :** la durée de vie du serveur n'est pas liée à la durée de vie du client. Il est donc beaucoup plus facile d'implémenter des services comme la rémanence des champs, les points de reprise et le partage de données entre applications, que dans le cas d'un programme SPMD ad hoc et monolithique.
- **Interactions inter-clients :** cette architecture permet la construction d'applications par composition d'applications existantes en utilisant le partage de données ou le pipelinage.

Le serveur-maître et les esclaves sont des programmes `JAVA` afin de supporter l'hétérogénéité des architectures cibles, la portabilité et l'extensibilité dynamique. Cet environnement logiciel doit *a priori* faciliter l'interaction entre les applications clientes, les ressources de calcul et les données. L'objectif est double : il s'agit de développer un environnement adapté aux calculs dynamiques et collaboratifs, et de fournir une plate-forme permettant de tester des constructions data-parallèles (en particulier celles impliquées par l'approche fonctionnelle du parallélisme de données), des mécanismes d'évaluation, des optimisations au-vol, et des stratégies de distribution des données et de répartition de charge.

2.3. Une tour de 3 langages

L'architecture logicielle que nous venons de décrire peut se modéliser par une tour de 3 langages. La communication entre le niveau i et $i+1$ de l'architecture logicielle correspond à la spécification d'un langage $\mathcal{L}_i$ qui fixe les opérations offertes et la représentation des champs par le niveaux $i+1$.

$$\begin{tikzcd}
\mathcal{L}_0 \arrow[r, "{}_0\mathrm{T}_1"] \arrow[dr, "\llbracket\;\rrbracket_0"'] & \mathcal{L}_1 \arrow[r, "{}_1\mathrm{T}_2"] \arrow[d, "\llbracket\;\rrbracket_1"] & \mathcal{L}_2 \arrow[dl, "\llbracket\;\rrbracket_2"] \arrow[loop right, "\mathcal{E}"] \\
& (\mathbb{Z}^n \to \mathit{Valeur}) &
\end{tikzcd}$$

FIG. 3 – *Relations entre les trois algèbres de champs de données $\mathcal{L}_0, \mathcal{L}_1$ et $\mathcal{L}_2$. Les fonctions ${}_i\mathrm{T}_{i+1}$ correspondent à des phases de l'évaluation. La relation $\mathcal{E}$ réduit un terme de $\mathcal{L}_2$ en une* constante *de $\mathcal{L}_2$ et correspond à l'interprète réalisé par un esclave (Cf. figure 2). Les fonctions $\llbracket\;\rrbracket_i$ sont les fonctions sémantiques qui associent à chaque expression de $\mathcal{L}_i$ un champ, i.e. un élément de $\mathbb{Z}^n \to$ Valeur. Les fonctions ${}_i\mathrm{T}_{i+1}$ sont telles que $\llbracket e_i \rrbracket_i = \llbracket {}_i\mathrm{T}_{i+1}(e_i) \rrbracket_{i+1}$ pour $i \in \{0,1\}$ et $e_i \in \mathcal{L}_i$. De plus, $\llbracket e_2 \rrbracket_2 = \llbracket \mathcal{E}(e_2) \rrbracket_2$, ce qui assure la correction du processus d'évaluation.*

Les requêtes émises par le client sont de deux types : des requêtes de calcul ou bien des commandes qui modifient le comportement opérationnel du serveur (par exemple, contrôle explicite de la répartition des données ou contrainte de synchronisation pour la récupération mémoire). Nous ne considérons ici que des demandes de calcul sur les champs ; elles sont représentées par un terme du langage $\mathcal{L}_0$.

L'évaluation d'un terme de $\mathcal{L}_0$ commence par son optimisation en un terme équivalent, puis se poursuit par sa traduction par ${}_0\mathrm{T}_1$ en un terme de $\mathcal{L}_1$. Le même traitement est appliqué à ce terme : après optimisation, un terme de $\mathcal{L}_1$ est traduit par ${}_1\mathrm{T}_2$ en un *ensemble* de termes de $\mathcal{L}_2$. Ces derniers sont envoyés pour évaluation finale aux esclaves qui assurent une exécution data-parallèle. Les processus esclaves sont principalement des $\mathcal{L}_2$-interprètes distribués dans un réseau.

Le processus d'évaluation est illustré à la figure 3. Les détails techniques sont développés dans [6]. Dans la suite nous allons présenter les langages $\mathcal{L}_0$, $\mathcal{L}_1$ et $\mathcal{L}_2$.

3. $\mathcal{L}_0$: calculer avec des fonctions de $\mathbb{Z}^n$

$\mathcal{L}_0$ ne permet pas d'exprimer n'importe quelle fonction sur $\mathbb{Z}^n$. En effet, nous voulons préserver une propriété opérationnelle de la structure de données « tableau idéal » : l'accès à un élément doit se faire en temps constant. En termes de champ de données, cela se traduit par le fait que l'évaluation de l'application d'un champ à un index doit s'évaluer en temps constant. En conséquence, nous acceptons deux types de fonctions comme champ de données : les fonctions dont le domaine de définition est fini, car elles peuvent être *tabulées* afin de satisfaire à la propriété précédente, et certaines familles de fonctions décrites par des règles d'évaluation en temps constant.

Le premier type d'éléments constitue les *constantes extensionnelles* du langage : elles seront gérées comme des ensembles de tableaux (Cf. le paragraphe 4). Le second type d'éléments correspond à des *constantes symboliques* ou *intensionnelles*. L'intérêt des constantes symboliques s'illustre à travers un exemple.

Utilisation des constantes intensionelles. L'évaluation habituelle de l'expression $\texttt{map}[+]\ \ A\ \ 1$, où A est un champ de données, doit assigner à la constante 1 un champ dont le domaine de définition est le même que celui de A, ce qui conduit à surcharger la constante avec plusieurs géométries [11, 17]. Notre interprétation consiste à considérer que 1 est un champ défini sur tout $\mathbb{Z}^n$ et que $\texttt{map}[+]$ est un opérateur strict. L'avantage est qu'il n'y a plus besoin de surcharger la constante 1, ni même de construire à l'exécution un tableau « rempli de 1 » : l'expression $\texttt{map}[f]\ \ A\ \ c_s$ où c_s est une constante symbolique peut être automatiquement reconnue et spécialisée en $\texttt{map}[\lambda z.f\ z\ c_s]\ \ A$ (l'opérateur $\texttt{map}$ est surchargé et s'applique à des fonctions d'arité quelconque).

Champs à domaine non borné et opérateur de restriction. Nous avons considéré deux familles de constantes intensionelles : $\texttt{L}[\alpha;\beta]$ et $\mathrm{R}[\alpha;\beta]$. Les paramètres α et β sont des éléments de $\mathbb{Z}^n$ et la sémantique de ces constantes est :

$$\begin{aligned} [\![\, \texttt{L}[\alpha;\beta] \,]\!]_0 &= \lambda z.(\alpha \,\centerdot\, z + \beta) \\ [\![\, \mathrm{R}[\alpha;\beta] \,]\!]_0 &= \lambda z.(\alpha \leq z) \wedge (z \leq \beta) \end{aligned}$$

(l'opérateur $\centerdot$ est le produit scalaire et $(u_1,\ldots,u_n) \leq (v_1,\ldots,v_n)$ si et seulement si pour tout i, $u_i \leq v_i$). La constante c est en fait l'abréviation de $\texttt{L}[(0,\ldots,0);(c,\ldots,c)]$. Les constantes $\texttt{R}[\ldots]$ permettent de construire des fonctions qui prennent la valeur vrai dans un hyper-rectangle (spécifié par deux points extrémaux de coordonnées α et β) et faux à l'extérieur de cette région. Il est possible de spécifier des « points à l'infini » afin de décrire des régions hyper-rectangulaires non bornées (par exemple $\mathtt{R}[0;\infty]$ représente la fonction indicatrice du segment positif de $\mathbb{Z}$).

Les constantes intensionelles permettent de construire des expressions de champs dont la valeur a un domaine infini et qui ne peuvent pas s'exprimer comme une constante intensionnelle (par exemple $\mathtt{map}[*]\ \mathtt{L}[1;0]\ \mathtt{L}[2;0]$ ne peut pas s'écrire comme un $\mathtt{L}[\alpha;\beta]$). Une solution est de considérer de telles expressions mais de ne les calculer que sur des domaines finis (leur valeur sera alors une constante extensionnelle). Cela amène l'introduction d'une nouvelle opération fondamentale qui est la *restriction explicite* d'un champ :

$$\mathtt{restrict}\ F\ B$$

avec F un champ quelconque et B un champ booléen. Le résultat est un champ qui coïncide avec F sur les points de l'index où le champ B prend la valeur vrai et qui est indéfini ailleurs.

Les expressions de $\mathcal{L}_0$. Outre les opérateurs classiques sur les tableaux, qui sont étendus aux champs (Cf. la figure 4 pour l'exemple de la généralisation du `map` aux champs de données), et l'opérateur de restriction que nous venons d'introduire, le langage $\mathcal{L}_0$ offre des *opérateurs géométriques* (translation, transposition, dilatation, injection...) et un opérateur d'*union asymétrique* `merge`. Une opération géométrique est une fonction qui n'intervient que sur l'indexation d'un champ:

$$[\![gop\ A]\!]_0 \;=\; [\![A]\!]_0 \circ gop$$

L'opération d'union asymétrique est une opération non stricte qui permet de fusionner deux champs :

$$[\![\mathtt{merge}\ A\ B]\!]_0 \;=\; \lambda z.\text{si } A(z) \neq \star \text{ alors } A(z) \text{ sinon } B(z)$$

$\star$ représente une valeur particulière : la valeur « indéfini ». Cette valeur est soigneusement distinguée de $\perp$ qui représente un calcul qui ne se termine pas, Cf. [22].

Optimisations dans $\mathcal{L}_0$. Les optimisations faites dans $\mathcal{L}_0$ consistent à simplifier les expressions comportant des restrictions et à transformer toute séquence d'opérations géométriques en une séquence normalisée comportant au plus cinq opérateurs [4]. Ces optimisations se font par des règles de réécriture, ce qui permet de les étendre facilement en incluant des règles supplémentaires.

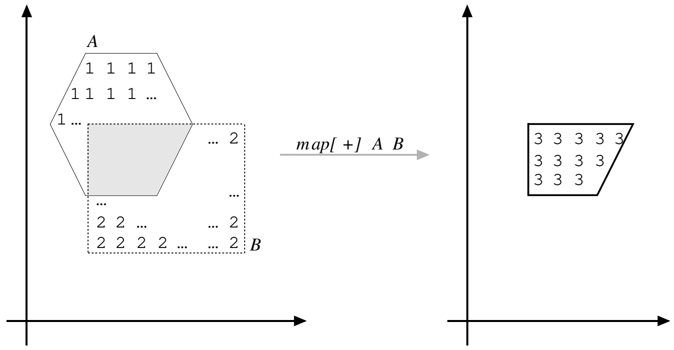

FIG. 4 – *Un exemple d'opération* **map** *dans* $\mathbb{Z}^2$.

La simplification des opérations de restrictions a été étudiée dans [16]. L'objectif est de propager les opérations de restrictions aux sous-expressions d'un terme. En effet, le coût d'une opération est, grossièrement, une fonction croissante du cardinal du domaine de définition de chacun de ses arguments. Voici un exemple :

$$\texttt{restrict}\,(\texttt{map}[f]\,A)\,B \quad\rightarrow\quad \texttt{map}[f]\,(\texttt{restrict}\,A\,B)$$

Et voici un exemple de règle de transformation de séquence géométrique :

$$\texttt{shift}[\alpha]\circ\texttt{shift}[\beta] \quad\rightarrow\quad \texttt{shift}[\alpha+\beta]$$

(l'opération $\texttt{shift}[\alpha]$ dénote une translation de vecteur α).

4. $\mathcal{L}_1$: calculer avec des ensembles de tableaux

L'objectif de $\mathcal{L}_1$ est double. Il s'agit d'abord d'annoter explicitement tous les expressions et toutes les constantes de $\mathcal{L}_0$ par leur *domaine d'itération*. En première approximation, le domaine d'itération correspond à la spécification du domaine d'une boucle `forall` réalisant le calcul dénoté. Dans le langage $\mathcal{L}_1$, ce domaine est spécifié par un champ booléen de $\mathcal{L}_0$, ce champ correspondant à l'indicatrice d'un sur-ensemble du domaine de définition de l'expression. L'expression de ce champ booléen ne prend pas une forme quelconque (Cf. plus bas). De plus, s'il correspond à une région infinie de $\mathbb{Z}^n$, une erreur est signalée.

Le deuxième objectif de $\mathcal{L}_1$ est d'éviter l'évaluation de certaines expressions en généralisant une astuce technique bien connue pour la transposition des matrices : on évite le calcul effectif de M^t en se rappelant d'utiliser $M(j,i)$ à la place de $M^t(i,j)$ dans la suite des calculs.

L'idée générale est de stocker, avec la représentation d'un champ, des informations sur l'accès aux éléments de ce champ, et de modifier ces informations d'accès plutôt que de toucher à la représentation du champ. Cette stratégie d'évaluation est intéressante si le surcoût induit par la gestion des accès est faible devant le calcul d'une représentation. Trois opérations de $\mathcal{L}_0$ bénéficient de cette stratégie « symbolique » d'évaluation : la *translation*, la *restriction* et l'*union asymétrique*. Ces trois opérateurs sont importants et apparaissent par exemple en algorithmique numérique dans les algorithmes de relaxation « red-black » et dans les méthodes explicites sur les grilles régulières.

Notons que la mémorisation de la translation à effectuer, en place de son évaluation effective, peut se voir comme un analogue de la *directive d'alignement* qui existe en `HPF` [31], mais où l'alignement est associé à chaque valeur plutôt qu'à une variable.

Les constantes de $\mathcal{L}_1$. Une constante de $\mathcal{L}_1$ prend l'allure suivante

$$\langle (s_1, b_1, s'_1, f_1) \; ; \; \ldots \; ; \; (s_p, b_p, s'_p, f_p) \rangle$$

où les s_i, s'_i sont des translations, les b_i sont des constantes booléennes de $\mathcal{L}_0$ et les f_i sont des constantes de $\mathcal{L}_0$. L'idée est que s_i est la translation associée à la garde booléenne b_i alors que s'_i est la translation attachée au champ de valeurs f_i. La valeur en un point z de $\langle \ldots \rangle$ est la valeur du premier quadruplet ayant une valeur définie en ce point c'est-à-dire :

$$\begin{array}{l} \langle\,\rangle(z) = \star \\ ((s, b, s', f) :: l)\,(z) = \texttt{si}\ (b \circ s)(z)\ \texttt{alors}\ (f \circ s')(z)\ \texttt{sinon}\ l(z) \end{array}$$

où l est une liste de quadruplets, « :: » dénote l'opération de concaténation de liste et $\star$ représente la valeur indéfinie.

Les expressions de $\mathcal{L}_1$. Les expressions de $\mathcal{L}_1$ sont constituées des opérateurs de $\mathcal{L}_0$ explicitement annotés par un *domaine d'itération*. Le domaine d'itération consiste simplement en une expression booléenne b de $\mathcal{L}_0$ qui est l'indicatrice d'un sur-ensemble du domaine de définition du champ. L'idée est que ce prédicat booléen agit comme un opérateur `restrict` sur chaque expression. En effet, la définition de la fonction sémantique $[\![\ ]\!]_1$ vérifie la propriété suivante : $[\![b]\!]_0 \sqsubseteq [\![b']\!]_0 \;\Rightarrow\; [\![e^b]\!]_1 \sqsubseteq [\![e^{b'}]\!]_1$ où $\sqsubseteq$ est l'ordre de Scott sur $(\mathbb{Z}^n \rightarrow \mathit{Value})$.

Nous n'admettons pas n'importe quelle expression booléenne pour b. En effet, le domaine d'itération correspond aussi à la description de la géométrie et de la localisation (sur les esclaves) du domaine de définition. Toute garde b se décrit comme une disjonction de prédicats élémentaires qui correspondent chacun à une région hyper-rectangulaire (disjointe des autres). Un hyper-rectangle élémentaire sera localisé sur un seul esclave.

Ainsi, le domaine d'itération permet tout à la fois d'expliciter un sur-ensemble du domaine de définition, de prévoir l'allocation des données, et d'expliciter aussi la distribution des données. La représentation du domaine d'itération par une union d'hyper-rectangles n'est sans doute pas la meilleure en terme de complexité et d'autres sont envisageables (par exemple, la représentation d'une région triangulaire dans $\mathbb{Z}^2$ utilisera n rectangles élémentaires, n étant la longueur d'un des cotés, alors que cette région sera décrite par un seul polyèdre). Cependant, les calculs de régions impliqués par l'évaluation des champs de données (intersection, union, translation...) sont simples avec cette représentation qui simplifie aussi la synthèse des ordres de calcul à envoyer aux esclaves.

La traduction de $\mathcal{L}_0$ en $\mathcal{L}_1$. La traduction ${}_0\mathrm{T}_1$ des termes de $\mathcal{L}_0$ en $\mathcal{L}_1$ est détaillée dans [6]. Voici quelques exemples qui illustrent le mécanisme d'évaluation « symbolique ». Si on suppose que ${}_0\mathrm{T}_1(F_i) = \langle s_i, b_i, s'_i, f_i \rangle$, alors

$$ {}_0\mathrm{T}_1(\texttt{merge}\ F_1\ F_2) = \langle (s_1, b_1, s'_1, f_1)\ ;\ (s_2, b_2, s'_2, f_2) \rangle $$

Pour une translation $t = \texttt{shift}[\alpha]$,

$$ {}_0\mathrm{T}_1(t\ F_1) = \langle s_1 \circ t, b_1, s'_1 \circ t, f_1 \rangle $$

Finalement, pour une restriction :

$$ {}_0\mathrm{T}_1(\texttt{restrict}\ F_1\ F_2) = \langle id, (b_1 \circ s_1) \wedge (b_2 \circ s_2) \wedge (f_2 \circ s'_2), s'_1, f_1 \rangle $$

où id dénote la fonction identité. Remarquez que cette dernière expression n'est pas une constante de $\mathcal{L}_1$ mais une expression de $\mathcal{L}_1$ si le prédicat $(b_1 \circ s_1) \wedge (b_2 \circ s_2) \wedge (f_2 \circ s'_2)$ ne peut pas se simplifier en une constante de $\mathcal{L}_0$.

Les optimisations dans $\mathcal{L}_1$. Les optimisations de $\mathcal{L}_1$ consistent à minimiser les gardes booléennes correspondant aux domaines d'itérations. La minimisation des gardes consiste à remplacer une garde par un prédicat plus précis sans changer la sémantique de l'expression. Formellement, on va remplacer e^b par $e^{b'}$ de telle sorte que $[\![e^b]\!]_1 = [\![e^{b'}]\!]_1$ mais que $[\![b']\!]_0 \Rightarrow [\![b]\!]_0$ (i.e. b' est moins défini ou bien prend moins souvent la valeur vraie que b).

Ce processus d'optimisation se fait en deux passes sur le DAG des expressions. La première passe, montante, propage les domaines d'itération des arguments au résultat d'une expression. La deuxième passe, descendante, propage le domaine du résultat aux arguments d'une expression.

Voici un exemple sur une expression vectorielle. L'expression

$$\texttt{map}[+] \quad 1 \quad (\texttt{restrict}\ 2\ \texttt{R}[0;10])$$

est une expression de $\mathcal{L}_0$ qui dénote un champ de données sur $\mathbb{Z}$. Elle correspond à additionner un vecteur infiniment long ne contenant que des 1 et un vecteur fini ne contenant que des 2 sur le domaine $\{0,1,\ldots,10\}$ ($\texttt{R}[0;10]$ représente l'indicatrice de l'ensemble $\{0,1,\ldots,10\}$). Cette expression est traduite en un terme de $\mathcal{L}_1$:

$$\begin{array}{ll} \texttt{map}^{true}[+] & \langle id, true, id, 1\rangle \\ & \qquad \texttt{restrict}^{true}\ \langle id, true, id, 2\rangle\ \langle id, \texttt{R}[0;10], id, true\rangle \end{array}$$

qui est optimisée en :

$$\texttt{map}^{\texttt{R}[0;10]}[+] \quad \langle id, \texttt{R}[0;10], id, 1\rangle \quad \langle id, \texttt{R}[0;10], id, 2\rangle$$

Notons qu'avec une seule passe montante, on n'aurait pas pu restreindre le domaine de la constante intensionnelle 1. Remarquons aussi qu'après la propagation des gardes booléennes, il n'y a plus de champ avec une extension infinie dans cet exemple. Cependant, les constantes symboliques 1 et 2 (abréviation de $\texttt{L}[0;1]$ et $\texttt{L}[0;2]$) restent symboliques et ne sont pas traduite en constante extensionnelle.

5. $\mathcal{L}_2$: calculer avec des vecteurs simples

Un terme de $\mathcal{L}_1$ est traduit en un ensemble de termes de $\mathcal{L}_2$. Ce langage manipule des vecteurs correspondant à la linéarisation d'une région hyper-rectangulaire. À chaque vecteur est associé un descripteur (s, b) décrivant le processus de linéarisation : si le vecteur `v` correspond au tableau `a`, alors $\texttt{a}[z] = \texttt{v}[s.z + b]$ avec « **.** » le produit scalaire. Les opérations de $\mathcal{L}_2$ correspondent aux opérations de $\mathcal{L}_1$ et aux formes curryfiées de celles-ci.

Pour ce qui est de la sémantique d'une expression de $\mathcal{L}_2$, nous avons développé une approche hybride originale. La fonction sémantique $[\![\ ]\!]_2$ n'est a priori définie que sur les constantes de $\mathcal{L}_2$ fournissant ainsi une interprétation minimale des objets représentés. Une *relation de réduction* $\mathcal{E}$ est définie sur $\mathcal{L}_2$ qui transforme toute expression en une constante (sa forme normale). Cette relation de réduction correspond exactement à l'interprète réalisé par les esclaves. La sémantique dénotationnelle d'une expression $\mathcal{L}_2$ est alors définie comme la dénotation de sa forme normale.

Ainsi, on rend compte du calcul effectué par les esclaves et on assure sa cohérence avec les langages $\mathcal{L}_0$ et $\mathcal{L}_1$.

Les processus esclaves sont principalement des $\mathcal{L}_2$-interprètes distribués dans un réseau. L'attribution d'un terme de $\mathcal{L}_2$ à un esclave se fait par le maître et correspond à une stratégie de répartition simple des données ayant pour but d'uniformiser l'utilisation mémoire des esclaves. D'autres stratégies, qui portent l'accent sur la minimisation des communications, sont à l'étude.

Typiquement, un esclave est en attente d'une requête du maître et crée de nouveaux processus légers pour exécuter les calculs correspondants. Un *scoreboard* est utilisé afin de synchroniser les calculs avec la réception éventuelle des arguments nécessaires. La sémantique exprimée en termes de relation de réduction rend bien compte du non-déterminisme du flot du contrôle dû à l'utilisation des processus légers. Enfin, une tâche de calcul correspond à une opération élémentaire implémentée par une seule boucle de calcul, cela indépendamment de la dimension de la région à traiter.

6. Implémentation

Une première implémentation est en cours de réalisation. L'objectif est de permettre une évaluation des fonctionnalités offertes par une telle architecture. La maquette actuelle[1] représente 15000 lignes de code `JAVA` qui implémentent un sous-ensemble des langages $\mathcal{L}_i$. Les seules opérations supportées sont `map` et `shift`. La représentation symbolique n'est pas implémentée. Le code du serveur-maître et des esclaves sont structurés par des « patrons de conception » (*design patterns*) ce qui rend l'implémentation très facilement extensible. La couche de communication utilisée est `RMI`.

Le choix de `JAVA` comme langage d'implémentation se justifie par des raisons de portabilité, de support de l'hétérogénéité des architectures cibles et de rapidité de développement (en particulier, grâce à la gestion implicite de l'allocation mémoire). D'autres choix auraient été possibles comme par exemple `OCAML`.

Notre effort n'a pas porté sur les performances, et celles-ci sont décevantes ; en effet, le choix de `JAVA` est coûteux, probablement pour trois raisons : le modèle de communication, la gestion dynamique de la mémoire et l'approche interprétée.

Plusieurs études ont montré que le choix d'un modèle de communication de type `RMI` n'est pas beaucoup plus coûteux qu'un modèle à

1. `http://www.lri.fr/~giavitto/HomePageOtto/`

passage de messages comme `PVM` ou `MPI` [7, 20]. De plus, l'utilisation de multiprotocoles de communication devrait permettre d'intégrer plus efficacement les deux mondes [8].

La gestion dynamique de la mémoire est souhaitable pour les applications irrégulières qui sont notre cible.

La lenteur de la machine virtuelle `JAVA` est donc le problème le plus important, sans doute en relation avec un traitement trop lourd des vecteurs de `JAVA`. Une voie pour l'amélioration des performances consisterait à utiliser un compilateur « juste à temps » et à profiter des nombreuses optimisations rendues possibles par notre architecture.

7. Travaux apparentés

Plusieurs des concepts et des techniques utilisés pour `FieldBroker` ont été développés dans le cadre d'autres projets.

La relation entre champs de données et fonctions a été étudiée dans [21].

La simplification des opérations géométriques de $\mathcal{L}_0$ est inspirée de [32].

L'analyse et la minimisation des gardes de $\mathcal{L}_1$ est similaire au problème étudié dans [23].

L'implémentation des régions de $\mathbb{Z}^n$ dans $\mathcal{L}_2$ n'est pas étrangère aux motivations et à l'approche étudiées dans le projet `LPARX` [19] qui développe un langage et une bibliothèque dédiée aux algorithmes numériques sur les structures non-uniformes de blocs. Cependant, nous nous sommes concentrés sur un traitement uniforme des régions, au lieu de considérer de multiples représentations.

La linéarisation des tableaux dans $\mathcal{L}_2$ est utilisée aussi dans A++/P++ [29] (lui-même utilisant un algorithme initialement décrit dans [28]).

L'utilisation d'architectures clients-serveurs pour les applications haute-performances (HPC) a été proposée récemment : [20, 15] et de nombreux efforts de recherches sont à présent menés pour essayer d'intégrer les « technologies WEB » dans le cadre HPC [9, 2, 1].

L'intégration de tous ces concepts, issus des domaines différents de la programmation fonctionnelle, de la programmation distribuée et de la programmation data-parallèle, est à notre connaissance unique.

La notion de constante symbolique, et la systématisation de la représentation symbolique de certaines opérations sont aussi des nouveautés. Des études supplémentaires sont nécessaires pour formaliser et généraliser cette approche, mais le cadre algébrique dans lequel les langages $\mathcal{L}_i$ sont définis, se révèle approprié.

La spécification des fonctions sémantiques $[\![\,]\!]_i$ et de la fonction d'évaluation $\mathcal{E}$ ne présentent pas de difficultés : les expressions ne sont pas récursives et les fonctions calculées sont simples. Le problème que nous avons rencontré réside plutôt dans la conception de $\mathcal{L}_1$ et dans la spécification des fonctions ${}_i\mathrm{T}_{i+1}$ qui imposent une représentation des champs et doivent rendre compte de la distribution des données.

Nous avons essayé de refléter dans une sémantique abstraite les problèmes fondamentaux de conception. L'objectif est de rendre compte et de pouvoir raisonner avec les outils fournis par la sémantique formelle, sur la conception d'une machine virtuelle parallèle (qu'est ce qui doit être communiqué, quelles sont les synchronisations, comment gérer la répartition, etc.) tout en restant indépendant d'une implémentation particulière. Un exemple est donné par le rôle que nous faisons jouer aux gardes de $\mathcal{L}_1$ qui servent à spécifier la localisation des sous-régions implémentant le domaine d'un champ de données. En imposant une forme close donnée à ces gardes, nous spécifions en même temps la représentation et la répartition des champs de données sur les processeurs (Cf. [6]).

Remerciements. Les auteurs remercient vivement D. De Vito qui a travaillé pendant une partie de sa thèse sur la plate-forme `FieldBroker`, ainsi que le groupe de travail LaMI-LRI sur le calcul distribué haute-performance pour de fructueuses discussions.

Références

[1] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. HPJava: data parallel extensions to Java. In *Proceedings of The ACM Workshop on Java for High-Performance Network Computing*, Stanford University, Palo Alto, California, February/March 1998.

[2] H. Casanova and J. Dongarra. The use of Java in the NetSolve project. In *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematic*, Berlin, 1997.

[3] Marina Chen, Young il Choo, and Jingke Li. Crystal: Theory and Pragmatics of Generating Efficient Parallel Code. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 7, pages 255–308. ACM Press, New York, 1991.

[4] D. De Vito. Simplification of sequence expressions of shift, inject, project and transpose applications on domains or grids of $\mathbb{Z}^n$. Technical Report 1043, Laboratoire de Recherche en Informatique, May 1996. 23 pages.

[5] D. De Vito. Compilation efficace du langage déclaratif 81/2. In M. Gengler and C. Queinnec, editors, *JFLA'97, Journées francophones des langages applicatifs*, Collection didactique, pages 85–104, Dolomieu, Isère, January 1997. INRIA.

[6] D. De Vito. *Conception et implémentation d'un modèle d'exécution pour un langage déclaratif data-parallèle.* Thèse de doctorat, Université de Paris-Sud, centre d'Orsay, Juin 1998.

[7] R. Fatoohi. Performance evaluation of communication software systems for distributed computing. Technical Report NAS-96-006, NASA Ames Research Center, June 1996.

[8] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

[9] I. Foster and S. Tuecke. Enabling technologies for web-based ubiquitous supercomputing. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, New York, August 1996. IEEE Computer Society Press.

[10] Pierrick Gachet, Christophe Mauras, Patrice Quinton, and Yannick Saouter. Alpha du Centaur: A prototype environment for the design of parallel regular algorithms. In *Conference Proceedings, 1989 International Conference on Supercomputing*, pages 235–243, Crete, Greece, June 5–9, 1989. ACM SIGARCH.

[11] J.-L. Giavitto. Typing geometries of homogeneous collection. In *2nd Int. workshop on array manipulation, (ATABLE)*, Montréal, 1992.

[12] J.-L. Giavitto, D. De Vito, and O. Michel. Semantics and compilation of recursive sequential streams in 81/2. In H. Glaser and H. Kuchen, editors, *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, volume 1292 of *Lecture Notes in Computer Science*, pages 207–223, Southampton, 3-5 September 1997. Springer-Verlag.

[13] J.-L. Giavitto, D. De Vito, and J.-P. Sansonnet. A data parallel Java client-server architecture for data field computations over $\mathbb{Z}^n$. In *EuroPar'98 Parallel Processing*, Lecture Notes in Computer Science, September 1998.

[14] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. Jr. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *Lecture Notes in Computer Science*, pages 209–215, Beaune (France), 2-4 October 1995. Springer-Verlag.

[15] Paul A. Gray and Vaidy S. Sunderam. IceT: Distributed computing and java. In *ACM Workshop on Java for Science and Engineering Computation*. ACM, 1997.

[16] Joacim Halén, Per Hammarlund, and Björn Lisper. An experimental implementation of a higly abstract model of data parallel programming. Technical Report TRITA-IT 9702, Royal Institute of Technology, Sweden, March 1997.

[17] C.B. Jay, D.G. Clarke, and J.J. Edwards. Exploiting shape in parallel programming. In *1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing: Proceedings*, pages 295–302. IEEE, 1996.

[18] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[19] S. R. Kohn and S. B. Baden. A robust parallel programming model for dynamic non-uniform scientific computation. Technical Report TR-CS-94-354, U. of California at San-Diego, March 1994.

[20] A. T. Krantz and V. S. Sunderam. Client server computing on message passing systems: Experiences with PVM-RPC. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing, Third International Euro-Par Conference*, number 1300 in LNCS, pages 110–117, Passau, Germany, August 1997. Springer-Verlag.

[21] B. Lisper. On the relation between functional and data-parallel programming languages. In *Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures.* ACM, ACM Press, June 1993.

[22] B. Lisper. Data parallelism and functional programming. In *Proc. ParaDigme Spring School on Data Parallelism.* Springer-Verlag, March 1996. Les Ménuires, France.

[23] B. Lisper and J.-F. Collard. Extent analysis of data fields. Technical Report TRITA-IT R 94:03, Royal Institute of Technology, Sweden, January 1994.

[24] A. Mahiout. Integrating the automatic mapping and scheduling for data-parallel dataflow applications on MIMD parallel architectures. In *Parallel Computing: Trends and Applications*, 19-22 September, Gent, Belgium, 1995. Elsevier. poster session.

[25] A. Mahiout, J.-L. Giavitto, and J.-P. Sansonnet. Placement et ordonnancement de graphes dataflow data-parallèles. In *5ièmes rencontres sur le parallélisme, Brest.* Univ. Brest & CNRS, 1993.

[26] A. Mahiout, J.-L. Giavitto, and J.-P. Sansonnet. Distribution and scheduling data-parallel dataflow programs on massively parallel architectures. In *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, pages 380–389, Moscow, Septembre 1994. Office of Naval Research USA & Russian Basic Research Foundation.

[27] O. Michel. Introducing dynamicity in the data-parallel language 81/2. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 678–686. Springer-Verlag, August 1996.

[28] I. Oliver. *Programming Classics: Implementing the World's Best Algorithms.* Prentice Hall, December 1994.

[29] R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Conference on Object-Oriented Numerics*, 1994.

[30] Patrice Quinton, Sanjay Rajopadhye, and Doran Wilde. Deriving imperative code from functional programs. In *Proceedings of the Seventh*

International Conference on Functional Programming Languages and Computer Architecture (FPCA'95), pages 36–44, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[31] Rice University, Houston, Texas. *High Performance Fortran Language Specification*, 1.1 edition, May 93.

[32] Luigi Semenzato. The infidel virtual machine. Technical Report UCB/CSD 93-761, Computer Science Division, University of California, Berkeley, 25 July 1993.

[33] Jay M. Sipelstein and Guy Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[34] G. L. Steele and W. D. Hillis. Connection machine LISP: Fine grained parallel symbolic programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, Cambridge, MA*, pages 279–297, New York, NY, 1986. ACM.

[35] Eric Violard and Guy-Rene Perrin. PEI: a single unifying model to design parallel programs. In *PARLE'93*. LNCS 694, Springler Verlag, 1993.

[36] J. Allan Yang and Young-il Choo. Data fields as parallel programs. In *Proceedings of the Second International Workshop on Array Structure*, Montreal, Canada, June/July 1992.