Bulk Synchronous Parallel ML: implémentation modulaire et prévision de performances

David Billiet

Stage de DEA

sous la direction de
Frédéric Loulergue et Frédéric Gava
Laboratoire d'Algorithmique, Complexité et Logique
61, avenue du général de Gaulle
94010 Créteil cedex – France
loulergue@univ-paris12.fr

Avril-Septembre 2004

Remerciements

Je tiens à remercier mes deux encadrants, Frédéric Loulergue et Frédéric Gava.

Je remercie particulièrement Frédéric Loulergue pour m'avoir accepté comme stagiaire au sein de son équipe. Je le remercie aussi pour sa gentillesse et d'avoir été disponible à chaque fois où j'avais besoin d'aide, n'hésitant à se déplacer juste pour m'aider.

Je remercie aussi Frédéric Gava pour son dynamisme et sa bonne humeur. Pour son aide précieuse ainsi que pour tous les petits conseils qu'il m'a donné. Je remercie également toutes les personnes travaillant au LACL, même si je ne les ai pas tous côtoyés. Je remercie en particulier Anatol Slissenko, le directeur du laboratoire.

Résumé

BSML (Bulk-Synchronous Parallel ML) et DMML (Departmental Metacomputing ML) sont deux extensions ML pour la programmation fonctionnelle d'algorithmes parallèles.

C'est autour de ces extensions que mon stage fût axé. Tout d'abord, mon travail était d'enrichir les implémentations proposées par la BSMLlib. La prévision de performance est un élément important dans l'élaboration de programmes, il était nécessaire d'avoir un programme écrit en BSML permettant d'obtenir les paramètres BSP de la machine parallèle. En plus de ces paramètres, le programme permet aussi d'obtenir la puissance de calcul de cette même machine. Enfin, une expérimentation a mis en évidence cette prévision de performance.

Le travail effectué a donné lieu à un article :

D. Billiet., F. Gava et F. Loulergue, A Modular Implementation of Bulk Synchronous Parallel ML. *Draft proceedings of the international workshop on Implementation of Functional Languages and Applications (IFL)*, Lübeck, Allemagne, 8-10 septembre 2004.

Une sélection est faite après le workshop sur des versions révisées des articles pour une publication dans un volume de la série *Lecture Notes in Computer Science*.

Table des matières

Chapitre 1

Introduction

Le stage s'est effectué au Laboratoire d'Algorithmique, Complexité et Logique (LACL) de l'université Paris 12 situé à Créteil dans le Val-de-Marne pour une durée de 4 mois.

J'ai travaillé dans le bureau des doctorants sur un poste de travail mis à ma disposition. Depuis ce poste j'avais également accès à une grappe de PC composée d'une machine frontale et de 6 noeuds de type pentium IV reliés par un réseau Gigabit Ethernet, le tout fonctionnant avec la distribution Mandrake Clic (phase 1).

L'équipe dans laquelle j'ai travaillé est composée actuellement de deux personnes, Frédéric Loulergue, le responsable d'équipe, et Frédéric Gava, un étudiant en thèse.

Les travaux décrit dans ce rapport s'inscrivent dans le projet CARAML [9], CARAML signifiant "CoordinAtion et Répartition des Applications Multiprocesseurs en objective camL". Le projet "CoordinAtion et Répartition des Applications Multiprocesseurs en objective camL" de l'ACI Globalisation des ressources informatiques et des données (GRID). Ce projet se termine. Il a regroupé des membres des universités d'Orléans (LIFO), Paris 6 & 7 (PPS), Paris 12 (LACL) et de l'INRIA, ayant en commun l'utilisation de CAML comme outil de développement et la programmation fonctionnelle comme modèle de base. Certains ont surtout contribué au langage Caml et donc à la programmation séquentielle, d'autres à la programmation parallèle ou concurrente.

L'objectif de ce projet était le développement de bibliothèques pour le calcul haute-performance et globalisé autour du langage Caml, dans son dialecte Objective Caml. Ces bibliothèques comprenaient des bibliothèques de primitives parallèles et globalisées ainsi que des bibliothèques applicatives orientées bases de données et calcul numérique.

La structure prévue des composantes logicielles du projet était faite de trois couches :

Implantation : Ocaml [12, 4, 14] et Jo-Caml [5, 8] comme infrastructures de calcul et de communication ;

Primitives : une bibliothèque BSMLlib *généralisée* comme système de programmation des algorithmes parallèles et globalisés, un algorithme pouvant être un

"serveur" s'il entrelace deux ou plusieurs algorithmes data-parallèles ;

Algorithmes : NUM et PDB, orientées respectivement vers les applications numériques et les systèmes d'information, comme bibliothèques de développement d'applications haute-performance et globalisées.

L'équipe de Paris 12 a travaillé sur les bibliothèques de primitives. La base est la bibliothèque BSMLlib pour "Bulk Synchronous Parallel ML library" qui s'appuie sur le modèle de parallélisme BSP [17, 15, 1] dont je décris les principes dans la section 2.1. Toutefois cette bibliothèque ne peut être utilisée que pour la programmation parallèle. La bibliothèque DMML, pour "Departmental Metacomputing ML", l'étend pour permettre la programmation de plusieurs grappes situées dans une même institution, comme une université. Je présente cette bibliothèque dans la section 2.2.

Ma contribution à ce travail est donnée dans les chapitres suivants, après un chapitre présentant BSML et DMML. Tout d'abord sur une nouvelle implémentation de la bibliothèque BSMLlib. Ensuite sur un programme de benchmark permettant de déterminer les paramètres BSP ainsi que des expériences de prévision de performances.

Le travail effectué a donné lieu à un article :

D. Billiet., F. Gava et F. Loulergue, A Modular Implementation of Bulk Synchronous Parallel ML. *Draft proceedings of the international workshop on Implementation of Functional Languages and Applications (IFL)*, Lübeck, Allemagne, 8-10 septembre 2004.

Une sélection est faite après le workshop sur des versions révisées des articles pour une publication dans un volume de la série *Lecture Notes in Computer Science*.

Chapitre 2

Préliminaires

2.1 Bulk Synchronous Parallel ML

Nous commençons par donner quelques élément du modèle BSP, puis nous donnons les primitives du langage Bulk Synchronous Parallel ML ainsi que quelques exemple.

2.1.1 Le modèle BSP

Le modèle *Bulk-Synchronous Parallelism* (BSP) est un modèle de programmation parallèle introduit par Valiant [17] pour offrir un niveau d'abstraction comparable aux modèles PRAM tout en permettant des performances prévisibles et portables sur une large variété d'architectures. Un ordinateur BSP contient un ensemble de paires processeur-mémoire, un réseau de communication permettant l'échange de messages inter-processeur et une unité de synchronisation globale qui exécute des demandes collectives de barrières de synchronisation. Ses performances sont caractérisées par trois paramètres : le nombre p de paires processeur-mémoire, le temps l nécessaire à une barrière de synchronisation et le temps g nécessaire à une 1-relation (phase de communication où chaque processeur envoie ou reçoit au plus un mot). Pour n'importe quel h le réseau peut réaliser une h-relation, c'est-à-dire une phase de communication où chaque processeur envoie ou reçoit au plus h mots, en temps gh.

Un programme BSP est exécuté comme une séquence de *super-étapes*, chacune étant au plus divisée en trois phases successives et logiquement disjointes (Figure 2.1).

Pendant la première phase, chaque processeur utilise ses données locales pour du calcul séquentiel et pour demander des transferts de données vers ou depuis d'autres nœuds. Pendant la seconde phase, le réseau effectue les transferts de données demandées. Pendant la troisième phase, une barrière de synchronisation se produit, rendant disponibles pour la super-étape suivante les données transférées. Le temps d'exécution d'une super-étape s est ainsi la somme du maximum des temps de calculs locaux, du temps de communication des données et du temps de synchronisation

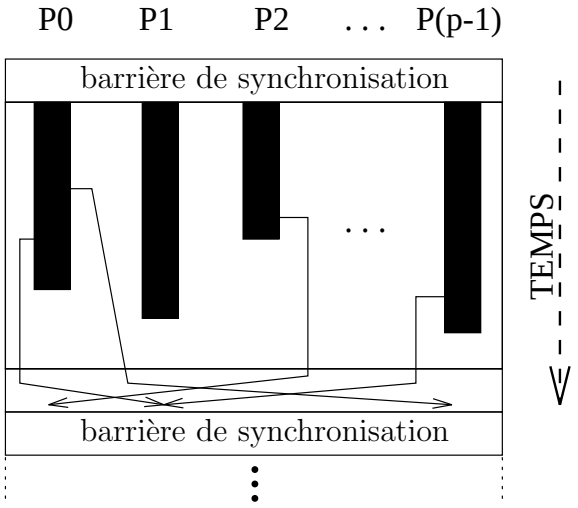

FIG. 2.1 – Super-étapes BSP

globale :

$$\mathrm{Time}(s) = \max_{i:processeur} w_i^{(s)} + \max_{i:processeur} h_i^{(s)} * g + l$$

où $w_i^{(s)}$ = temps de calcul local du processeur i durant la super-étape s et $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ où $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) est le nombre de mots transmis (resp. reçus) par le processeur i durant la super-étape s.

Le temps d'exécution $\sum_s \mathrm{Time}(s)$ d'un programme BSP composé de S super-étapes est la somme de trois termes : $W + H * g + S * l$ où $W = \sum_s \max_i w_i^{(s)}$ et $H = \sum_s \max_i h_i^{(s)}$. En général W, H et S sont fonctions de p et de la taille des données n, ou de paramètres plus complexes. Pour minimiser le temps d'exécution, un algorithme BSP doit minimiser conjointement le nombre de super-étapes, le volume total H (resp. W) et les déséquilibres $h^{(s)}$ (resp. $w^{(s)}$) de communication (resp. de calcul local).

2.1.2 La bibliothèque BSMLlib

Le langage Bulk Synchronous Parallel ML (BSML) est basé sur une extension confluente du λ-calcul [13]. Il est sans blocage et est déterministe. Il n'y a pas d'implémentation en tant que langage à part entière mais une bibliothèque, BSMLlib, de programmation fonctionnelle en Objective Caml pour la programmation selon le mode direct du modèle *Bulk Synchronous Parallelism*.

Il est possible de prouver le bon fonctionnement des programmes BSML avec l'assistant de preuve Coq [6]. Il en est de même pour la prédiction des performances de programmes BSML que nous présenterons au chapitre 4. Aujourd'hui, la bibliothèque BSMLlib en est à la version 0.3, qui est présentée au chapitre suivant.

Le noyau

Le noyau de la bibliothèque BSMLlib est basé sur les primitives suivantes :

bsp_p: unit → int
mkpar: (int → α) → α **par**
apply: (α → β) **par** → α **par** → β **par**
type α option = None | Some **of** α
put: (int → α option) **par** → (int → α option) **par**
at: α **par** → int → α

La fonction **bsp_p** () permet d'accéder au paramètre p donnant le nombre de processeurs de la machine BSP, p ne varie pas au cours de l'exécution du programme. Il y a un type polymorphe abstrait α **par** qui représente le type des vecteurs parallèles de taille p. L'imbrication des types **par** est interdite. Les vecteurs parallèles sont construit avec la fonction **mkpar** tel que (**mkpar** f) stocke (f i) sur le processeur i pour i compris entre 0 et (p-1). On écrit généralement f comme une fonction **fun** pid → e où l'expression e peut être différente sur chaque processeur. On dit que e est locale. L'expression (**mkpar** f) est un objet parallèle et est dite globale.

Un algorithme BSP est une combinaison de phases de calculs locales asynchrones suivi de phases de communications globales synchrones.

Les phases de calculs asynchrones sont programmées avec les fonctions **mkpar** et **apply**. L'expression (**apply** (**mkpar** f) (**mkpar** e)) stocke la valeur ((f i)(e i)) sur le processeur i.

Considérons l'expression suivante :

let vf = **mkpar**(**fun** i → (+) i)
and vv = **mkpar**(**fun** i → 2∗i+1) **in**
apply vf vv

Les deux vecteurs parallèles sont respectivement équivalent à :

fun x → x+0	**fun** x → x+1	···	**fun** x → x+(p−1)

et

0	3	···	$2 \times (p-1)+1$

L'expression **apply** vf vv est évaluée comme suit :

0	4	···	$2 \times (p-1)+2$

Les phases de communications et de synchronisations sont éffectuées par la fonction **put**. Considérons l'expression :

$$\textbf{put}(\textbf{mkpar}(\textbf{fun}\ \mathrm{i} \rightarrow \mathrm{fs}_i)) \tag{2.1}$$

Le type α option permet de définir des valeurs optionnelles. Ce type permet de savoir quelles valeurs sont à envoyer. Une valeur de type α option est soit None soit

Some v avec v de type α . Pour envoyer une valeur v du processeur j au processeur i, la fonction fs_j au processeur j doit être telle que (fs_j `i`) est évaluée à Some v. Dans le cas où le processeur j n'envoie pas de valeur au processeur i, cette même expression doit être évaluée à None.

Exemples

Quelques fonctions utiles peuvent être définies avec les primitives de bases. Par exemple, la fonction replicate créé un vecteur parallèle contenant la même valeur sur tous les processeurs. La primitive apply ne peut être utilisée qu'avec des vecteurs parallèle de fonctions ne prenant qu'un seul argument. Pour pouvoir les utiliser avec des fonctions à deux arguments, il est nécessaire de définir la fonction apply2.

let replicate x = **mkpar**(**fun** pid → x)
let apply2 vf v1 v2 = **apply** (**apply** vf v1) v2

Il est aussi commun d'appliquer une même fonction sequentielle sur tous les processeurs. Cela peut être réalisé en utilisant les fonctions parfun : elles se distinguent uniquement par le nombre d'arguments de la fonction à appliquer.

let parfun f v = **apply** (replicate f) v
let parfun2 f v1 v2 = **apply** (parfun f v1) v2
let parfun3 f v1 v2 v3 = **apply** (parfun2 f v1 v2) v3

applyat n f_1 f_2 v applique la fonction f_1 au processeur n et la fonction f_2 sur les autres processeurs :

let applyat n f1 f2 v =
  apply (**mkpar**(**fun** i → **if** i=n **then** f1 **else** f2)) v

Un exemple de fonction de communication, la fonction d'échange totale (totex). La sémantique de la fonction totex est donné par :

$$\text{totex} \langle\ v_0\ , \ldots ,\ v_{p-1}\ \rangle\ =\ \langle\ f\ , \ldots ,\ f\ , \ldots ,\ f\ \rangle$$

où $\forall i.(0 \leq i < p) \Rightarrow (f\ i) = v_i$.

Dans le code qui suit, la fonction noSome enlève le constructeur Some et la fonction compose est une fonction usuelle de compisition.

(∗ val totex: α par → (int → α) par ∗)
let totex vv = parfun (compose noSome)
  (**put**(parfun (**fun** v dst → Some v) vv))

2.2 Departmental metacomputing ML

Les besoins de calcul sont tels qu'il est désormais nécessaire d'utiliser des réseaux de grappes ou de machines parallèles plutôt qu'une seule machine parallèle. Lorsque

ces machines sont dans plusieurs services d'une même institution (en général elles sont connectées par un même réseau mais le réseau interne de chaque grappe peut varier d'une grappe à l'autre) ont parle de Departmental metacomputing. DMML ou Departmental metacomputing ML est une extension de ML pour ce type de programmation.

2.2.1 DMML

L'application du modèle BSP sur ce type d'architecture n'est pas envisageable. Deux problèmes majeurs empêche cette application. Premièrement, les différentes grappes qui constituent le département n'ont pas forcément les mêmes réseaux internes. Une barrière de synchronisation sur le département entier serait bien trop coûteuse en temps pour espérer des performances globales satisfaisantes. Deuxièmement, ce modèle ne prend pas en compte les différentes capacités des machines parallèles ainsi que de leurs réseaux. Il n'est donc pas concevable d'utiliser le modèle BSP pour ce type d'architecture. Pour y remédier, le DMML est basé sur une architecture à deux couches. Il y a la couche globale, celle qui concerne le département entier et la couche locale au niveau de chaque unité BSP. Deux modèles sont utilisés, le modèle BSP qui concerne la grappe (qu'on nomme unité BSP) et le modèle MPM pour le département. Le modèle MPM est directement inspiré du modèle BSP. Dans ce modèle on ne parle plus de super-étape mais de m-étape. En une m-étape, chaque processeur exécute une phase de calcul suivie d'une phase de communication. Durant cette phase de communication, les processeurs s'échangent des données pour la prochaine m-step. Cependant, à la différence du modèle BSP, le modèle MPM ne comporte pas de barrière de synchronisation. Une fois les données nécessaires à un processeur reçues, celui-ci passe à la prochaine m-étape. Ceci permet de s'affranchir du problème de la barrière de synchronisation au niveau départemental.

2.2.2 La bibliothèque standard

Plutot qu'un langage complet, la DMML est disponible sous la forme d'une bibliothèque Objective Caml. La DMMLlib étend la BSMLlib en lui ajoutant des fonctions au niveau de la couche départementale. la figure 2.2 présente le noyau de la DMMLlib. Il propose des fonctions d'accès aux paramètres du département, en particulier, la fonction **dm_p**:unit → int (resp. **dm_g** et **dm_l**) tel que la valeur de **dm_p**() est P, le nombre statique d'unités BSP (resp. G et L, le temps de communication et le temps de latence du département). Les paramètres des unités BSP sont aussi disponible au travers des fonctions **dm_bsp_p**, **dm_bsp_s**, **dm_bsp_g** et **dm_bsp_l**. Par exemple, (**dm_bsp_p** i) donne le nombre de processeurs de la $i^{\text{ème}}$ unité BSP.

En outre, il y a un nouveau type polymorphique α **dep** qui représente un vecteur départemental d'objets de type α , un par unité BSP. L'imbrication d'objets de type **dep** dans **dep** ou de type **par** dans **par** est interdite. Mais le α d'un type **dep**

dm_p: unit → int
dm_g: unit → float
dm_l: unit → float
dm_bsp_p : int → int
dm_bsp_s : int → int
dm_bsp_g : int → int
dm_bsp_l : int → int
mkdep : (int → α) → α **dep**
applydep : (α → β) **dep** → α **dep** → β **dep**
get : (int → int → int option) **par dep** → (int → α option) **par dep**
→ (int → int → α option) **par dep**
atdep : α **par dep** → int → int → α

FIG. 2.2 – librairie DMML

peut être une valeur usuelle d'Objective Caml ou d'une valeur parallèle BSML.

La DMMLlib travaille donc sur des vecteurs départemental, ces vecteurs sont construit avec la fonction **mkdep**. (**mkdep** f) crée une valeur (f i) sur l'unité BSP i pour i compris entre 0 et $(P-1)$. Les valeurs parallèles BSML (de type **par**) ne doivent pas être évaluées en dehors d'un **mkdep**. Ceci sera contraint par un système de typage bien qu'actuellement seul le programmeur est responsable de respecter cette règle.

Les phases de communications d'une m-étape sont exécutées par la fonction **get**.

$$\textbf{get}(\textbf{mkdep}(\textbf{fun}\ \mathrm{a} \rightarrow \textbf{mkpar}(\ \textbf{fun}\ \mathrm{i} \rightarrow \mathrm{f}_{a,i})))(\textbf{mkdep}(\textbf{fun}\ \mathrm{b} \rightarrow \textbf{mkpar}(\ \textbf{fun}\ \mathrm{j} \rightarrow \mathrm{v}_{b,j}))) \tag{2.2}$$

Pour le processeur i de l'unité BSP a, pour recevoir la $n^{\text{ième}}$ valeur du processeur j de l'unité BSP b, la fonction $f_{a,i}$ au processeur i de l'unité BSP a doit être telle que ($f_{a,i}$ b j) est évaluée à Some n. Pour ne recevoir aucune valeur ($f_{a,i}$ b j) doit être évaluée à None.

Sur un vecteur départemental contenant des vecteurs parallèles de fonctions $f'_{a,i}$, notre expression évalue toutes les fonctions de messages délivrés sur tous les processeurs de toutes les unités BSP.

Pour le processeur i de l'unité BSP a, ($f'_{a,i}$ b j) est évaluée à None si le processeur i de l'unité BSP a ne reçoit pas de message du processeur j de l'unité BSP b ou si ($v_{b,j}$ n)est évaluée à None (le processeur j de l'unité BSP b n'a pas la $n^{\text{ième}}$ valeur). Par contre, elle est évaluée à Some $v^n_{b,j}$ si il a reçu une valeur du processeur j de l'unité BSP b et si ($v_{b,j}$ n) est évaluée à $v^n_{b,j}$.

Il y a aussi une fonction de projection **atdep**. elle s'utilise de la même façon que la fonction **at** mais prend en arguments un vecteur départemental de vecteurs parallèles et deux entiers qui sont l'identifiant du cluster et l'identifiant du processeur considéré. Cette fonction ne doit pas être évalué dans un **mkdep**. Utiliser **atdep** permet d'avoir un comportement global dépendant d'une valeur locale.

Chapitre 3

La bibliothèque BSMLlib modulaire

La version 0.25 de la bibliothèque BSMLlib n'était pas modulaire, au sens des modules de Objective Caml. Cette nouvelle implémentation était un objectif pour la version 0.3. De passer à une version modulaire permet d'harmoniser les différentes implémentations. Cela facilite aussi la maintenance et les évolutions futurs de la bibliothèque.

L'implémentation des primitives de cette bibliothèque repose sur deux éléments. Une implémentation générique prenant en argument un module de communication. Ce second élément, le module Comm, permet la communication entre les processeurs. Plusieurs implémentations de ce module sont disponibles. Ces implémentations sont décrites dans la section 3.2. Nous commençons par donner une aperçu du module générique.

3.1 Un aperçu de l'implémentation

Dans l'implémentation de la bibliothèque BSMLlib version 0.3, le module qui contient les primitives présentées dans la section 2.1.2, est implémenté en SPMD utilisant une bibliothèque de communication de plus bas niveau. Ce module, appelé Comm, est basé sur les principaux éléments donnés dans la figure 3.1.

val pid : unit $\rightarrow$ int
val nprocs : unit $\rightarrow$ int
val send : α option array $\rightarrow$ α option array

FIG. 3.1 – Le module Comm

Il y a plusieurs implémentations du module Comm basé sur MPI [16], PVM [7], BSPlib [10], PUB [3], TCP/IP, etc. L'implémentation de tous les autres modules

de la bibliothèque BSMLlib, notamment le module Bsmllib, est indépendante de l'implémentation du module Comm, mais dépend seulement de son interface.

La signification de pid et de nprocs est évidente, ils donnent respectivement l'identifiant du processeur et le nombre de processeurs de la machine parallèle. La fonction send prend sur chaque processeur un tableau de taille nprocs(). Ces tableaux contiennent des valeurs optionnels. Ces valeurs d'entrées sont obtenues en appliquant sur chaque processeur i la fonction f_i (argument de la primitive **put**) sur les entiers allant de 0 à (bsp()−1).

Si au processeur j la valeur contenue à l'index i est (Some v) alors la valeur v sera envoyée du processeur j au processeur i. Si la valeur est None, aucune valeur ne sera envoyée. Dans le résultat, qui est aussi un tableau, None à l'index j sur le processeur i signifie que le processeur j n'a pas envoyé de valeur au processeur i et la valeur (Some v) signifie que le processeur j a envoyé la valeur v à i. Une synchronisation globale s'effectue à l'intérieur de cette fonction de communication. **put** and **at** sont implémentés en utilisant send.

L'implémentation des types abstraits, **mkpar** et **apply** est la suivante :

type α **par** = α
let mkpar f = f (pid())
let apply f v = f v

Pour la version MPI, la fonction pid n'appelle pas la fonction MPI_Comm_rank à chaque fois. La fonction MPI est appelé une seule fois à l'initialisation et la valeur est stockée dans une référence. La fonction pid retourne seulement la valeur de cette référence.

Regardons comment la fonction **put** travaille. Pour cela, on considère que l'on travaille sur une machine parallèle à 4 processeurs. Soit la fonction f_i de type int $\rightarrow$ α **par** telle que $(f_i\ (i+1))$ = Some v_i pour $i = 0, 1, 2$ et $(f_i\ j)$ = None sinon. L'expression **mkpar**(**fun** i $\rightarrow$ f_i) sera évaluée comme suit :

1. d'abord, sur chaque processeur la fonction est appliquée à tous les identifiants des processeurs. On produit ainsi p valeurs, les messages à envoyer pour chaque processeur. Dans la figure qui suit, chaque colonne représente les valeurs produites par processeurs et chaque ligne correspond à une destination (la première ligne représente les messages envoyés au processeur 0, etc.) :

None	None	None	None
Some v_0	None	None	None
None	Some v_1	None	None
None	None	Some v_2	None

2. Ensuite l'échange s'effectue et on obtient une nouvelle matrice, qui est en fait la transposée de la matrice précédente.

None	Some v_0	None	None
None	None	Some v_1	None
None	None	None	Some v_2
None	None	None	None

3. Finalement, le vecteur parallèle de fonctions est produit. Chaque processeur i a un tableau a_i de taille p (une colonne de la matrice précédente) et la fonction est **fun** x $\rightarrow a_i$.(x). Dans notre exemple, au processeur 3 $(f_3\ 0)$ = None, ce qui signifie que le processeur 3 n'a pas reçu de message du processeur 0 et $(f_3\ 2) = \text{Some}\, v_2$ qui veut dire que le processeur 2 a envoyé un message au processeur 3.

3.2 Les différents modules Comm

3.2.1 Implémentation MPI

Pour la version 0.3, deux implémentations MPI sont disponibles. La première implémentation utilise une fonction de communication collective de la bibliothèque MPI pour l'envoi des messages contrairement à la seconde implémentation où les messages sont envoyés un par un. Le principe de la fonction send est le même pour les deux implémentations. L'échange se fait en deux étapes.

La première étape consiste en l'envoi de la taille des messages qui seront envoyer entre les processeurs. Si le processeur i doit envoyer un message de taille n au processeur j alors il lui envoie un message contenant la taille de la donnée qui lui est destinée. Au total,pour une machine à p processeurs, il y a $p(p-1)$ messages envoyés. Cette première étape est indispensable pour que chaque processeur puisse allouer la mémoire nécessaire à la réception des messages de la deuxième étape. Et dans la deuxième étape, les processeurs s'envoient et réceptionnent les messages.

Dans la première implémentation, les envoies de messages des deux étapes se font avec la fonction collective MPI_Alltoall de la bibliothèque MPI.

Néanmoins, nous voulions avoir une seconde implémentation dans laquelle l'envoie des messages ne se fait pas par des fonctions collectives de la bibliothèque MPI. La première étape est identique dans les deux implémentations, seul la seconde change.

Ne pas utiliser de fonctions collective nécessite une attention particulière afin d'éviter tout problème de blocage. Pour ce faire, l'utilisation d'un carré latin pour ordonnancer l'ordre d'envoi et de réception des messages est un moyen sûr pour y parvenir. Le principe est le suivant :

Chaque processeur i envoie un message au processeur $(i+1) \bmod p$ (p est le nombre de processeurs) puis reçoit un message du processeur $(i-1) \bmod p$. Ensuite il envoie un message au processeur $(i+2) \bmod p$ puis reçoit un message du processeur $(i-2) \bmod p$. Ainsi de suite jusqu'à envoyer au processeur $(i+p-1)$ et recevoir du processeur $(i-p+1)$.

Ce principe permet d'éviter les cas de blocage où, par exemple, le processeur i est en attente d'un message du processeur j et le processeur j attend lui aussi un message du processeur i.

3.2.2 Implémentation PVM

PVM fonctionnant par passage de messages, tout comme MPI, on a pu s'appuyer sur les bases de l'implémentation MPI pour développer en PVM. Cependant, le nombre de fonctions collectives de la bibliothèque PVM est très réduite, contrairement à MPI. Le principe de l'implémentation PVM est le principe du carré latin expliqué dans l'implémentation MPI.

Tout comme dans l'implémentation MPI, l'échange se fait en deux étapes. Durant la première étape, tous les processeurs envoient aux autres processeurs une valeurs correspondant à la taille des messages qu'ils s'envoient. La deuxième étape consiste en l'envoi et la réception des messages.

Une des particularité de PVM sont les entrées-sorties standard des programmes. Par exemple, avec MPI la sortie d'un printf se fait sur la machine sur laquelle on a lancé le programme. Avec PVM, la sortie standard des noeuds n'est pas redirigé vers la machine à partir de laquelle on contrôle le programme. Hors, ce qu'on souhaitait, c'est que le comportement de la BSMLlib soit identique quelque soit l'implémentation du module Comm. Le problème a été contourné en utilisant une fonction PVM permettant de stocker dans un fichier les sorties de chaque noeud. Ensuite, le contenu du fichier s'affiche sur la console de la machine principale, après avoir subit un prétraitement.

3.2.3 Implémentation TCP/IP

Contrairement aux précédentes implémentations, l'implémentation TCP/IP est écrite intégralement en Objective Caml. Dans cette implémentation il n'y a pas d'utilisation de bibliothèque par passage de message. Tout est géré explicitement, les sockets comme les threads ainsi que les synchronisations. Les sockets proviennent de la bibliothèque Unix de Caml. En conséquence, le code écrit pour cette implémentation est beaucoup plus important. Contrairement aux deux précédentes implémentations, les deux étapes sont imbriqués et on utilise aussi le carré latin.

Chaque processeur i qui veut envoyer un message au processeur $(i + 1) \bmod p$ lui envoie d'abord la taille du message puis le message lui-même. Ensuite, il se met en attente de deux messages du processeur $(i - 1) \bmod p$, la taille du message qu'il va recevoir et le message lui-même. Ainsi de suite jusqu'au processeur $(i + p - 1)$. Ensuite les processeurs se synchronisent.

Chapitre 4

Prévision de performance

Un des principaux avantages du modèle BSP est son modèle de coût : il est simple mais précis. Plusieurs papiers (par exemple [11]) l'ont démontré en utilisant la bibliothèque BSPlib [10] ou la bibliothèque Paderborn University BSP (PUB) [2]. Dans ce livre [1], Bisseling présente, dans le premier chapitre, le modèle BSP, la programmation en utilisant la bibliothèque BSPlib et des exemples. Dans ces exemples, le programme "probe" est un benchmark utilisé pour déterminer les paramètres des machines BSP. J'ai adapté ce programme pour BSML (section 4.1). Ensuite un petit exemple de prévision a été effectué (section 4.2).

4.1 Le programme `bsmlprobe`

4.1.1 Pourquoi ?

Pour prédire le temps d'exécution d'un programme BSML, il est nécessaire de connaître les paramètres BSP ainsi que la puissance de calcul de la machine qui exécutera le programme. Il existe de nombreux programmes dans différents langages permettant de les calculer mais aucun n'étaient écrit en BSML. On a donc écrit un programme déterminant les paramètres BSP dans ce langage.

4.1.2 Comment

Il y a quatre paramètres BSP. Bien sur, le nombre de processeurs n'a pas besoin d'être benchmarké. Les paramètres g et L sont exprimés comme multiples de la vitesse des processeurs r. Ce paramètre est le premier à être déterminé. C'est aussi le plus difficile à déterminer avec précision.

Pour pouvoir le développer, nous nous sommes aidés du livre de R. Bisseling [1] expliquant comment déterminer les paramètres BSP. La détermination de la puissance de calcul de la machine parallèle est le premier. Pour cela, on mesure le temps mit par la machine pour calculer les opérations y :=a*x+y et y :=a*x+y un certain nombre de fois (x et y sont des vecteurs). Cette combinaison d'opérations

est représentative pour la majorité des calculs scientifiques. Il est important de bien choisir la taille des vecteurs. Prendre des vecteurs trop petit ne permet pas d'obtenir une bonne précision, et il ne faut pas travailler sur des vecteurs trop grand afin d'éviter les défauts de cache.

Le paramètre de communication g et celui de synchronisation l sont obtenues en mesurant le temps total d'une h-relation. Ces paramètres se mesurent en flops et sont dépendant du paramètre r qui se mesure en Mflops/s. Une h-relation est une étape de communication où chaque processeur envoie au plus h mots et reçoit au plus h mots et où au moins un processeur a reçu ou envoyé h mots, un mot étant un réel ou un entier. Dans bsmlprobe, les mots sont des flottants. La technique utilisée est appelée méthode des plus faibles carrés.

4.1.3 Paramètres BSP de notre grappe

La figure 4.1 présente les résultats obtenus en exécutant le probe (C+MPI) et bsmlprobe (BSMLlib 0.3 avec les diverses implémentations du module Comm) 10 fois sur une grappe constituée de 6 noeuds Pentium IV interconnectés par un réseau Gigabit Ethernet.

4.2 Expérience

Nous avons faits quelques expérimentations sur un programme qui calcule le produit intérieur de deux vecteurs. Il y a deux versions, une utilisant les tableaux pour stocker les vecteurs et une autre utilisant les listes. Le code est donné dans la figure 4.2. Il utilise la fonction fold_direct de la bibliothèque standard BSMLlib. La formule de coût BSP de inprod est :

$$n + 2 \times p + (p - 1) \times g + L$$

Ces programmes ont été exécutés 15 fois en incrémentant à chaque fois la taille des vecteurs (de 5000 à 100000) et on prend la moyenne. La figure 4.3 résume les temps d'exécution. Les prédictions de performances en utilisant les paramètres obtenus dans la section précédente sont aussi donnés (version MPI).

C+MPI				BSMLlib (MPI)			
	r	g	L		r	g	L
1	477	44.9	449167	1	468	29.3	227258
2	477	44.9	449167	2	462	40.2	218733
3	478	19.4	673227	3	469	44.5	217905
4	480	23.5	680942	4	469	31.3	223746
5	479	28.5	652335	5	454	23.0	220677
6	478	20.0	685116	6	479	16.0	239412
7	476	15.7	672573	7	471	28.5	227467
8	477	20.1	660288	8	473	23.7	233947
9	476	18.2	643489	9	458	16.1	229192
10	478	16.8	665109	10	484	27.7	236784
Avg	**478**	**25.2**	**623141**	**Avg**	**469**	**28.0**	**227512**
r en Mflops/s, g et L en flops							

BSMLlib (PVM)				BSMLlib (TCP/IP)			
	r	g	L		r	g	L
1	273	4.9	55856	1	273	3.5	36208
2	273	11.1	55761	2	273	5.8	36108
3	307	16.9	61443	3	307	7.2	39544
4	273	17.0	55004	4	239	8.5	30771
5	273	8.0	56557	5	239	9.8	30687
6	273	16.8	55077	6	307	10.2	39228
7	307	16.5	62156	7	273	6.9	35406
8	273	14.9	56030	8	273	10.2	34597
9	341	19.1	68581	9	273	8.5	34702
10	239	8.8	48558	10	307	3.1	40580
Avg	**283**	**13.4**	**57502**	**Avg**	**276**	**7.4**	**35783**
r en Mflops/s, g et L en flops							

FIG. 4.1 – Benchmark des paramètres BSP (p=6)

```
let inprod_array v1 v2 =
  let s = ref 0. in
    for i = 0 to (Array.length v1)−1 do
      s:=!s+.(v1.(i)∗.v2.(i));
    done;
    !s

let inprod_list v1 v2 =
  List.fold_left2 (fun s x y → s+.x∗.y) 0. v1 v2

let inprod seqinprod v1 v2 =
  let local_inprod = parfun2 seqinprod v1 v2 in
    Bsmlcomm.fold_direct (+.) local_inprod
```

FIG. 4.2 – Produit intérieur en BSML

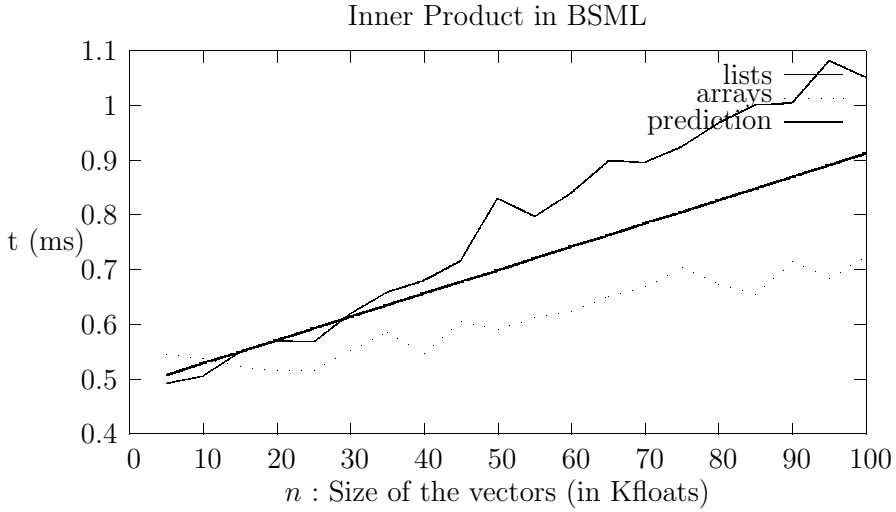

FIG. 4.3 – Temps d'exécution

Chapitre 5

Conclusions

Le metacomputing est un domaine très intéressant que je ne connaissait pas beaucoup. Ce stage m'a permit de mieux connaître ce domaine de l'informatique.

J'ai rencontré certaines difficultés tout au long du stage :

- Il a fallu que je révise et approfondisse mes connaissance sur le modèle BSP, en particulier le modèle de coûts ainsi que la bibliothèque de développement BSPlib (pour comprendre les programmes de benchmark existants)
- Connaissant très peu la programmation fonctionnelle, j'ai dû me mettre à niveau pour pouvoir commencer les implémentations.
- L'implémentation PVM m'a posé beaucoup de difficultés. Je ne connaissais pas du tout cette bibliothèque et mes encadrants n'avaient pas de connaissances pratiques non plus. Ce qui fait que j'ai dû avancer pas à pas. Au final, l'implémentation fonctionne.
- L'implémentation TCP/IP aussi a été difficile. Je connaissais ce protocole mais je n'avais pas eu l'occasion de mettre en pratique. Ce stage m'a donc permis d'y palier.
- La compréhension du fonctionnement du BSML a été assez laborieuse, notamment dans les communications entre processeurs. La logique est toute autre de la programmation parallèle classique. Mais au final, la programmation en BSML apporte un plus indéniable dans l'écriture de programme parallèle, étant peu éloigné de l'écriture séquentielle.

L'expérience a été très satisfaisante, je suis très content du stage même si je n'ai pas été aussi productif que je l'aurai souhaité. Par exemple je n'ai pas eu le temps de concevoir le module Comm utilisant la bibliothèque PUB. De plus, le fait de travailler en laboratoire et de côtoyer des chercheurs et des doctorants m'a permit de mieux comprendre le fonctionnement de la recherche scientifique.

Au niveau des perspectives de ce travail, il reste à uniformiser les scripts de lancement des programmes BSML et à tester intensivement la bibliothèque avant que la version 0.3 soit distribuée. Ceci devrait être fait d'ici la fin de l'année. Des expériences complémentaires sur la prévisions de performances seront menées, notamment avec des programmes plus conséquents.

Annexe A

Les modules Comm

Voici les fonctions send des différentes implémentations du module Comm présentées dans la section 3.2.

A.1 Première implémentation MPI

```
let send data =
if Array.length data <> (nprocs())
then failwith "Mpi.alltoall: wrong array size";
let buffers =
  Array.map (fun d  → Marshal.to_string d [Marshal.Closures]) data in
(* Determine lengths of strings *)
let sendlengths = Array.map String.length buffers in
let total_len = Array.fold_left (+) 0 sendlengths in
let send_buffer = String.create total_len in
let pos = ref 0 in
for i = 0 to (nprocs()) − 1 do
  String.blit buffers.(i) 0 send_buffer !pos sendlengths.(i);
  pos := !pos + sendlengths.(i)
done;
let recvlengths = Array.create (nprocs()) 0 in
(* Alltoall those lengths *)
alltoall_int_array sendlengths recvlengths;

let total_len = Array.fold_left (+) 0 recvlengths in
(* Allocate receive buffer *)
let recv_buffer = String.create total_len in
(* Do the alltoall *)
alltoall_string send_buffer sendlengths recv_buffer recvlengths;
(* Build array of results *)
let res0= Marshal.from_string recv_buffer 0 in
let res = Array.make (nprocs()) res0 in
```

```
  let pos = ref 0 in
  for i = 1 to (nprocs()) - 1 do
    pos := !pos + recvlengths.(i - 1);
    res.(i) <- Marshal.from_string recv_buffer !pos
  done;
  res
```

A.2 Seconde implémentation MPI

```
let send data =
  if Array.length data <> (nprocs())
  then failwith "Mpi.alltoall: wrong array size";
  let buffers =
  Array.map (fun d  → match d with
        None  → ""
        | _  → Marshal.to_string d [Marshal.Closures]) data in
  let sendlengths = Array.map String.length buffers in
  let recvlengths = Array.create (nprocs()) 0 in
  let envoi = alltoall_int_array sendlengths recvlengths in
  let retour = Array.make (nprocs()) data.(pid()) in
  let temp=retour.(pid()) <- data.(pid()) in
  for i=0 to (nprocs()) - 2 do
     let destination = (pid() + i +1) mod (nprocs()) in
     if sendlengths.( destination ) <>0
     then
     begin
     mpi_send buffers.( destination ) (pid()) destination sendlengths.( destination );
     end;
     if recvlengths.( (nprocs()+pid() −i −1) mod (nprocs()) ) <>0
     then
        let receivebuffer =
            String.create recvlengths.( (nprocs()+pid() −i −1) mod (nprocs()) ) in
        let res= mpi_receive receivebuffer ((nprocs()+pid()−i−1) mod(nprocs()))
                 (pid()) recvlengths.((nprocs()+pid()−i−1) mod (nprocs())) in
        begin
        let res0= Marshal.from_string receivebuffer 0 in
        retour.((nprocs()+pid() −i −1) mod (nprocs())) <- res0;
        end;
done;
retour
```

A.3 Implémentation PVM

```
let send data =
  if Array.length data <> (nprocs())
```

```
  then failwith "PVM_send_data:_wrong_array_size";
  let buffers =
  Array.map (fun d  → match d with
        None  → ""
        | Some x  → Marshal.to_string (Some x) [Marshal.Closures]) data in
  let sendlengths = Array.map String.length buffers in
  let recvlengths = Array.create (nprocs()) 0 in
  for i=0 to (nprocs()) − 2 do
     let destination = (pid() + i +1) mod (nprocs()) in
     pvm_send_int sendlengths.( destination ) (pid()) destination;
     recvlengths.((nprocs()+pid() −i −1) mod (nprocs()))<−
     pvm_receive_int ( (nprocs()+pid() −i −1) mod (nprocs())) (pid());
  done;
let retour = Array.make (nprocs()) None in
let temp=retour.(pid()) <− data.(pid()) in
for i=0 to (nprocs()) − 2 do
   let destination = (pid() + i +1) mod (nprocs()) in
   if sendlengths.( destination ) <>0
   then
      pvm_send_str buffers.( destination ) (pid()) destination sendlengths.(destination);
      if recvlengths.( (nprocs()+pid() −i −1) mod (nprocs()) ) <>0
      then
         let receivebuffer =
                  String.create recvlengths.((nprocs()+pid()−i−1) mod (nprocs())) in
         let receivebuffer0=
                  pvm_receive_str receivebuffer ((nprocs()+pid()−i−1) mod (nprocs()))
                  (pid()) recvlengths.((nprocs()+pid()−i−1) mod (nprocs())) in
         let res0= Marshal.from_string receivebuffer 0 in
         retour.((nprocs()+pid() −i −1) mod (nprocs())) <− res0;
done;
retour
```

A.4 Implémentation TCP/IP

```
  let send values_to_send =
   (* create an array of the received values *)
   let values_received = Array.make (!p_ref) None in
   (* for all other processes *)
   for i=0 to (!p_ref−2) do
    (* take the pid to send from the latin−square *)
    (let pid_for=tab_pid_to_send.(i) in
      (* send a value to another process *)
      (* print_pid ("send to "^(string_of_int pid_for)); *)
       ignore(Thread.create
                (fun ()  → send_a_value values_to_send.(pid_for) pid_for) ());
```

```
      (* print_pid ("has send to "^(string_of_int pid_for)) *) );
    (* received a value *)
    (* print_pid ("accept a value "); *)
    accept values_received;
    (* synchronize with the sended thread *)
    wait_sended ()
  done;
  values_received.(!pid_ref)<-values_to_send.(!pid_ref);
  memo_values:=[];
  (* print_pid ("debut barrier"); *)
  (* barrier of synchronization *)
  barrier_of_sync ();
  (* print_pid ("fin barrier"); *)
  nb_sync:=((!p_ref)-1);
  values_received
```

Bibliography

[1] R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.

[2] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library - Design, Implementation and Performance. In *Proc. of 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, San-Juan, Puerto-Rico, April 1999.

[3] O. Bonorden, B. Juurlink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2) :187–207, 2003.

[4] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml.* O'Reilly France, 2000. freely available in english at http ://caml.inria.fr/oreilly-book.

[5] S. Conchon and F. Le Fessant. Jocaml : Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, pages pages 22–29. IEEE Press, 1999.

[6] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3) :365–376, 2003.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation Series. MIT Press, 1994.

[8] G. Gonthier and C. Fournier. The Join Calculus : a Language for Distributed Mobile Programming. In G. Barthe, P. Dyjber, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, number 2395 in LNCS, pages 268–332. Springer, 2002.

[9] G. Hains and al. Coordination et répartition des applications multiprocesseurs en Objective Caml (CARAML). ACI Grid, 2002-2004. `http ://www.caraml.org`.

[10] J.M.D. Hill, W.F. McColl, and al. BSPlib : The BSP Programming Library. *Parallel Computing*, 24 :1947–1980, 1998.

[11] S.A. Jarvis, J.M.D Hill, C.J. Siniolakis, and V.P. Vasilev. Portable and architecture independent parallel performance tuning using BSP. *Parallel Computing*, 28 :1587–1609, 2002.

[12] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.07. Web pages at www.ocaml.org, 2003.

[13] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3) :253–277, 2000.

[14] D. Rémy. Using, Understanding, and Unravellling the OCaml Language. In G. Barthe, P. Dyjber, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, number 2395 in LNCS, pages 413–536. Springer, 2002.

[15] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3) :249–274, 1997.

[16] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.

[17] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103–111, August 1990.