Implantation d'une bibliothèque de programmation fonctionnelle BSP dans un environnement de méta-computing

Otmane BOUZIANI

Master recherche 2 SSI

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

;

Remerciements

Je remercie d'abord mon responsable de stage Frédéric GAVA pour sa patience et pour m’avoir encadré et guider dans mon travail.

Je tiens à remercier tout particulièrement louis GESBERT tant pour son soutien technique au moment de la mise en place des grappes que pour sa disponibilité.

Je tiens à remercier Frédéric LOULERGUE de m’avoir accepté parmi son équipe de recherche pour effectuer ce stage et pour ses conseils et son aide.

Je remercie les membres du Laboratoire LACL et notamment l’équipe d’enseignement du Master de recherche de nous avoir accueillir durant cette année universitaire, et spécialement monsieur Alexis BES pour ses efforts pour nous motiver, ainsi que monsieur Anatole SLISSENKO le directeur du laboratoire.

Résumé

De nos jours, les machines parallèles seules sont aptes à délivrer les puissances de calcul importantes. Le parallélisme de données est un paradigme de programmation parallèle dans lequel un programme décrit une séquence d'actions sur des tableaux à accès parallèle. Le modèle Bulk Synchronous Parallel ou BSP vise à maximiser la portabilité des performances en ajoutant une notion de processus explicites au parallélisme de données. Un programme BSP est écrit en fonction du nombre de processeurs de l'architecture sur laquelle il s'exécute. Le modèle d'exécution BSP sépare synchronisation et communication et oblige les deux à être des opérations collectives. Il propose un modèle de coût faible et simple permettant de prévoir les performances de façon réaliste et portable. BSML est un langage fonctionnel pour les programmes parallèles BSP. Il permet la programmation data-parallèle basée sur une structure de données parallèle et polymorphe qui est manipulée à l'aide d'opérations dédiées. Les difficultés de la programmation SPMD sont éliminées. Ainsi, les inter-blocages sont impossibles et le déterminisme est garanti. Pour des applications à grande échelle, plus d'une machine parallèle est nécessaire. On parle de métacomputing. Le projet DMML vise la conception (avec la sémantique formelle), d'une extension de ML (disponible sous la forme d'une bibliothèque pour Objective Caml) pour le métacomputing départemental. Ce travail vise à implémenter une nouvelle version de la BSMLlib et de la DMMlib en utilisant la suite logicielle PM2 et son interface de communication Madeleine, en vue d'expérimenter des algorithmes parallèles sur des grappes de PC et constater le gain de performance par rapport aux anciennes implémentations.

Abstract

Nowadays, only the parallel machines are ready to deliver the important computing powers. The parallelism of data is a paradigm of parallel programming in which a program describes a sequence of actions on vectors with parallel access. The model Bulk Synchronous Parallel (BSP) aims at maximizing the portability of the performances by adding a concept of explicit processes to parallelism of data. A BSP program is written according to the number of processors of the architecture on which it is carried out. The BSP execution model separates synchronization and communication and obliges both to be collective operations. It proposes a reliable and simple cost model making it possible to predict the performances in a realistic and portable way. BSML is a functional language for BSP parallel programs. It allows the data parallel programming based on a parallel and polymorphic data structure which is handled using dedicated operations. The difficulties of SPMD programming are eliminated. Thus, deadlocks are impossible and the determinism is guaranteed.
For applications on a large scale, more than one parallel machine is necessary. One speaks about métacomputing. Project DMML aims at the design (with formal semantics), of an extension of ML (available in the shape of a library for Objective Caml) for the métacomputing départemental. This work aims at implementing a new version of both BSMLlib and DMMlib by using the software PM2 and its interface of communication Madeleine, in order to try out parallel algorithms on bunches of PC and to note the profit of performance compared to old implementations.

Table des matières

Chapitre1

Introduction

1.1 Introduction

Certains problèmes comme la simulation de phénomènes physiques ou chimiques ou la gestion de bases de données de grande taille nécessitent des performances que seules les machines massivement parallèles peuvent offrir. Leur programmation demeure néanmoins plus difficile que celle des machines séquentielles. La conception de langages adaptés est un sujet de recherche actif.

Le parallélisme de données est un paradigme de programmation parallèle dans lequel un programme décrit une séquence d'actions sur des tableaux à accès parallèle. Le modèle BSP [6, 7] vise à maximiser la portabilité des performances en ajoutant une notion de processus explicites au parallélisme de données. Un programme BSP est écrit en fonction du nombre de processeurs de l'architecture sur laquelle il s'exécute. Le modèle d'exécution BSP sépare synchronisation et communication et oblige les deux à être des opérations collectives. Il propose un modèle de coût fiable et simple permettant de prévoir les performances de façon réaliste et portable.

Le projet BSML a deux objectifs principaux : parvenir à des langages universels et dans lesquels le programmeur peut se faire une idée du coût à partir du code source. Cette dernière exigence nécessite que soient explicites dans les programmes les lieux du réseau statique de processeurs de la machine.

BSML est étendu par des opérations parallèles BSP qui s'avère confluentes et universelles pour les algorithmes BSP. La BSMLlib [3] est une implantation partielle de ces opérations sous forme d'une bibliothèque pour le langage Objective CAML [4]. Cette bibliothèque permet d'écrire des programmes parallèles BSP portable sur une grande variété d'architectures.

1.2. Le stage

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 [24] est une extension de ML pour ce type de programmation.

Au cours du stage on a eu recours à un environnement de programmation parallèle de haut niveau et son support exécutif pour faciliter le développement et l'exécution de nos applications : **PM2** (Parallel Multithreaded Machine).

Les objectifs de ce stage sont de:

- implémentation BSML en utilisant la bibliothèque Madeleine de l'environnement d'exécution PM2.
- implémenter DMML, en utilisant PM2/Madeleine comme, une bibliothèque pour Objective Caml.
- expérimenter les implantations sur les machines parallèles du LACL[1].

Le stage s'est effectué au Laboratoire d'Algorithmique, Complexité et Logique (**LACL**) de l'université Paris 12 Créteil dans le Val-de-Marne pour une durée de 5 mois.

La grappe principale est une machine frontale et de 7 nœuds de type Pentium IV reliés par un réseau Fast Ethernet, le tout fonctionnant avec la distribution **ubuntu dapper drake 6.06**. En plus d'un serveur de type pentium II.

Les travaux décrit dans ce rapport s'inscrivent dans la suite logique des travaux effectués lors du projet Caraml [9] "CoordinAtion et Repartition des Applications Multiprocesseurs en objective camL" de l'ACI Globalisation des ressources informatiques et des données (GRID). Ce projet qui regroupait des membres de l'université d'Orléans

1 J'ai également participé avec Frédéric GAVA et Louis GESBERT à la mise en place de trois grappes de PC composées au Total de 29 nœuds à l'UFR de droit de l'université Paris 12.

(LIFO), Paris 6 & 7 (PPS), Paris 12 (LACL) et de l'INRIA. L'objectif de ce projet était le développement de bibliothèques pour le calcul haute performance et globalisé autour du langage Ocaml. 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 [4, 12, 14] et Jo-Caml [5, 8] comme infrastructures de calcul et de communication.

Primitives : des bibliothèques de primitives de haut niveau comme système de programmation des algorithmes Parallèles et globalisés.

Algorithmes : *NUM* et *PDB*, orientés 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, qui s'appuie sur le modèle de parallélisme **BSP** [1, 15, 17] dont les principes sont décris dans la section 2.1. Toutefois cette bibliothèque ne peut être utilisée que pour la programmation parallèle.

La bibliothèque DMML, pour "Departmental Metacomputing ML", a été élaborée pour permettre la programmation de plusieurs unités BSP situées dans une même institution, comme une université.

Ma contribution à ce travail, qui consiste essentiellement en une nouvelle implémentation de la biblio- thèque BSMLlib utilisant l'environnement d'exécution PM2 et l'interface de communication MADELEINE. Dans un premier temps nous présenterons le modèle de coût BSP, la BSMLlib et la DMMLlib (chapitre2). Ensuite nous donnerons les critères qui ont fait opter pour l'environnement PM2 et sa bibliothèque de communication MADELEINE (chapitre 3). Le chapitre 4 est dédié à la description de l'implémentation PM2 de BSML. Enfin le travail effectué en vue d'une implémentation PM2 de DMML est donné dans le chapitre 5. Le but étant de constater le gain de performance en utilisant la bibliothèque de communication MADELEINE de PM2.

Chapitre 2

PRELEMINAIRES

2.1 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-processeurs 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 exécute comme une séquence de **super-étapes**, chacune étant au plus divisée en trois phases successives et logiquement disjointes (cf. 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.

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

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 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).

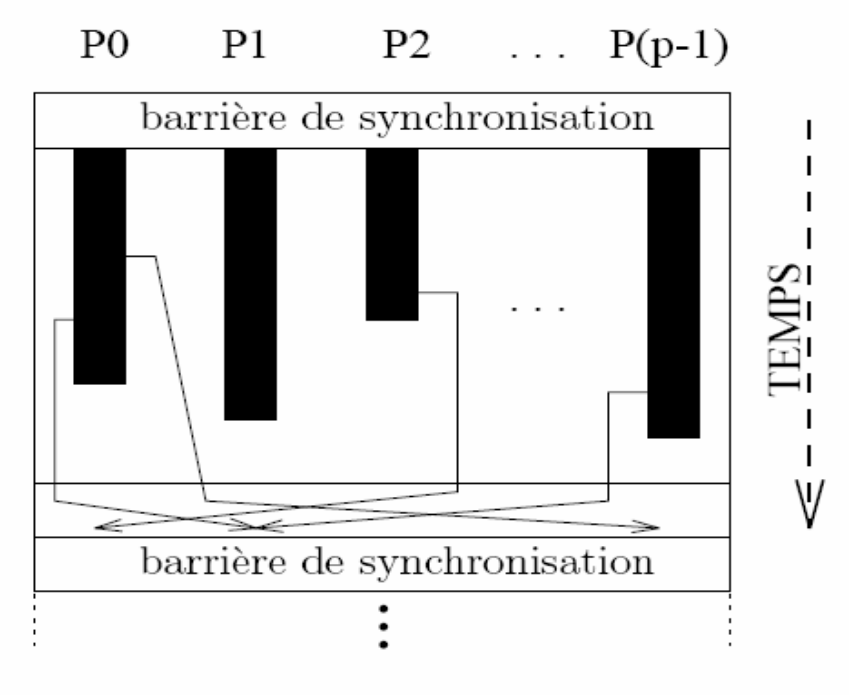

Fig. 2.1- super étape du modèle BSP

2.2 La Bibliothèque BSMLlib

Le langage Bulk Synchronous Parallel ML (BSML) est basé sur une extension parallèle de ML. Il est sans blocage et est déterministe. Il n'y a pas d'implémentation en tant que langage à part entière mais une bibliothèque, la BSMLlib, pour le langage de programmation Objective Caml. Il est possible de prouver le bon fonctionnement des programmes BSML avec l'assistant de preuve Coq [6]. Basé sur le modèle BSP, et Il est prouvable d'en utiliser le modèle de coût pour la prédiction des performances de programmes BSML.

2.2.1 Le noyau

Le noyau de la bibliothèque BSMLlib est basé sur les primitives données dans la figure 2.2. La fonction ***bsp_p*** permet d'accéder au paramètre p donnant le nombre de processeurs de la machine BSP, p ne variant pas au cours de l'exécution du programme. Le parallélisme provient d'un type polymorphe abstrait ***α par*** qui représente le type des vecteurs parallèles de taille p. L'imbrication des types par est interdite [34]. Les vecteurs parallèles sont construits 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.

bsp_p : $unit \longrightarrow \text{int}$

mkpar : $(\text{int} \longrightarrow \alpha) \longrightarrow \alpha\ par$

applay : $(\alpha \longrightarrow \beta)\ par \longrightarrow \alpha\ par \longrightarrow \beta\ par$

type $\alpha\ option = None \mid Some\ of\ \alpha$

put : $\alpha\ par \longrightarrow \text{int} \longrightarrow \alpha\ fun\ pid \longrightarrow e$

proj : $\alpha\ par \longrightarrow \text{int} \longrightarrow \alpha\ option$

Fig. 2.2- Noyau de la bibliothèque BSMLlib

Considérons l'expression suivante :

Let $vf = mkpar(\ fun\ i \longrightarrow (+)\ i)$
and $vv = mkpar(\ fun\ i \longrightarrow 2 * i + 1)\ in$ **apply** $vf\ \ vv$

Les deux vecteurs parallèles sont respectivement équivalent à :

fun $x \longrightarrow x + 0$	**fun** $x \longrightarrow x + 1$	.....	**fun** $x \longrightarrow x + (p - 1)$

Et

0	3	.....	$2 \times (p - 1) + 1$

L'expression $vf\ \ vv$ est évaluée comme suit :

0	4	.....	$2 \times (p - 1) + 2$

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

put $(mkpar(\ fun\ i \longrightarrow fs_i\ ))$

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 f^{s_j} au processeur j doit être telle que (f^{s_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.

2.2.2 Exemples

Quelques fonctions utiles peuvent être définies avec les primitives de bases. Par exemple, la fonction ***replicate*** crée un vecteur parallèle contenant la même valeur sur tous les processeurs. La primitive ***apply*** ne peut être utilisée qu'avec des vecteurs parallèles 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 $replicate\ x = mkpar(\ fun\ pid \longrightarrow x)$
Let $apply2\ vf\ v1\ v2 = apply(apply\ vf\ v1)\ v2$

Il est aussi commun d'appliquer une même fonction séquentielle 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 $parfun3\ f\ v1\ v2\ v3 = apply(\ parfun2\ f\ v1\ v2)\ v3$

$applyat\ n\ f1\ f2\ v$ applique la fonction f1 au processeur n et la fonction f2 sur les autres processeurs :

Let $applyat\ n\ f1\ f2\ v =$
$$\textbf{apply}(mkpar(\ fun\ i \longrightarrow if\ i = n\ then\ f1\ else\ f2))\ v$$

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

$$\textbf{totex}\ \langle v_0, ..., v_{p-1} \rangle = \langle f, ..., f, ..., f \rangle$$
$$\forall i.\ (0 \le i \le p) \Rightarrow (f\ i) = v_i.$$

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

```
(* val totex : 'a par -> (int -> 'a) par *)
totex vv = parfun (compose noSome)
    (put (parfun (fun v dst -> Some v) vv))
```

2.3 DEPARTMENTAL METACOMPUTING ML

2.3.1 DMML

L'utilisation du modèle BSP sur une architecture de type méta-ordinateur n'est pas a priori envisageable. Deux problèmes majeurs empêchent cet usage. Premièrement, les différentes grappes qui constituent le méta-ordinateur n'ont pas forcément les mêmes réseaux internes de communication. 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 modèle DMM 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 en ce qui concerne les grappes (qu'on nomme unités BSP) et le modèle MPM pour leur coordination (cf. Figure 2.3). 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-étape. 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.

2.3.2 La bibliothèque standard

Plutôt 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 2.3 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 disponibles 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 ième 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, peut être une valeur usuelle d'Objective Caml ou d'une valeur parallèle BSML. La DMMLlib travaille donc sur des vecteurs départementaux, ces vecteurs sont construits 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**.

dm_p : $unit \longrightarrow$ int
dm_g : $unit \longrightarrow float$
dm_l : $unit \longrightarrow float$
dm_bsp_p : $unit \longrightarrow$ int
dm_bsp_s : $unit \longrightarrow$ int
dm_bsp_g : $unit \longrightarrow$ int
dm_bsp_l : $unit \longrightarrow$ int
mkdep : $(\text{int} \longrightarrow \alpha) \longrightarrow \alpha\ dep$
applydep : $(\alpha \longrightarrow \beta)\ dep \longrightarrow \alpha\ dep \longrightarrow \beta\ dep$
get : $(\text{int} \longrightarrow \text{int} \longrightarrow \text{int}\ option)\ par\ dep \longrightarrow (\text{int} \longrightarrow \alpha\ option)\ par\ dep$
$\longrightarrow (\text{int} \longrightarrow \text{int} \longrightarrow \alpha\ option)\ par\ dep$
projdep : $\alpha\ par\ dep \longrightarrow \text{int} \longrightarrow \text{int} \longrightarrow \alpha\ option$

Fig. 2.3-Primitive de la Librairie DMML.

Considérons l'expression suivante :

get $(mkdep(fun\ a \longrightarrow mkpar\ fun\ i \longrightarrow f_{a,i}))(mkdep(fun\ b \longrightarrow mkpar(fun\ j \longrightarrow v_{b,j})))$

Pour le processeur i de l'unité BSP a, pour recevoir la n^{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. le résultat de l'évaluation d'une primitive **get** est un vecteur départemental de vecteurs parallèles de fonctions $f'_{a,i}$, décrivant les messages reçus par chaque processeur i de chaque unité BSP a.

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 ($v^n_{b,j}$. n) est évaluée à None (le processeur j de l'unité BSP b n'a pas de n^{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 ($v^n_{b,j}$. n) est évaluée à $v^n_{b,j}$.

Il y a aussi une fonction de projection **projdep**. Elle s'utilise de la même façon que la fonction proj 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 évaluée dans un mkdep. Utiliser **projdep** permet d'avoir un comportement global dépendant d'une valeur locale.

Chapitre 3

La suite logiciels PM2

Nous introduisons dans ce chapitre les outils que nous avons utilisés pour réaliser nos nouvelles implémentations de la BSMLlib et la DMMLlib, obtenant ainsi une version dérivée répondant aux objectifs fixés. Nous allons successivement détailler les caractéristiques de ces outils pour justifier de leur choix.

3.1 Interfaces de communication et support d'exécution

Afin de mettre au point notre architecture, un environnement de programmation et d'exécution performant est indispensable. De plus, notre but étant d'aboutir à une version de la BSMLlib et la DMMLlib à la fois très efficaces et disponibles pour un nombre important de plateformes, nous avions besoin d'une bibliothèque de communication qui soit à la fois très performante et générique. Ces choix sont importants car ils conditionnent non seulement le niveau des performances obtenues mais également le travail à effectuer pour disposer d'un support pour un panel varié de technologies réseaux.

Ainsi notre choix s'est porté sur l'environnement de programmation PM2, qui nous offrait les outils nous semblant les mieux appropriés pour une telle réalisation, notamment la bibliothèque de communication MADELEINE. Avant de présenter PM2 et MADELEINE, nous allons nous attarder un peu sur les aspects importants qui ont permit de choisir notre interface de communication parmi toutes celles qui existent. Aussi nous allons voir pourquoi ce choix est-il crucial pour un réseaux hautes performances.

3.1.1 Interfaces de communication hautes performances

Vu le niveau de performance des technologies réseau utilisées au sein des grappes, le moindre surcoût logiciel ajouté par une bibliothèque de communication peut très vite devenir un facteur limitant les performances globales des applications (appel système, recopie mémoire, etc.) C'est la raison pour laquelle les interfaces et protocoles de transport « classiques » tels que TCP/IP sont inadaptés et que des travaux de recherche portant sur la définition d'interfaces de communication spécialisées pour le calcul intensif ont été menés. D'abord concentrées sur des bibliothèques capables d'exploiter efficacement le réseau interne des supercalculateurs (e.g. travaux sur **Active Messages** [19]), ces recherches se sont fortement intensifiées et diversifiées lors de la percée des grappes de PC dans le domaine du calcul intensif.

Étant donnée l'hétérogénéité des solutions matérielles disponibles pour les grappes, de nombreuses interfaces de communication sont naturellement apparues, chacune dédiée à une technologie spécifique. C'est ce que l'on appelle les interfaces « orientées performance ». On peut citer pour exemple l'interface **MX** [25] pour la technologie **MYRINET**, **ELAN** [34] pour **QUADRICS** ou encore **SISCI** [36, 18] pour le réseau **SCI**. Si ces interfaces de communication exhibent évidemment les meilleures performances possibles sur leurs technologies respectives, elles restent uniquement destinées à servir de cible pour des bibliothèques de plus haut-niveau. Développer une application directement au-dessus d'une telle interface ruinerait tout espoir de portabilité sur d'autres types de matériel réseau.

C'est précisément pour cette raison que la disponibilité d'interfaces de communication portables est cruciale pour le développement d'applications parallèles pour des grappes de PC. En fait, de telles interfaces existent depuis longtemps (e.g. **PVM** [29] puis surtout **MPI** [24, 21, 16]) et des implémentations sont disponibles sur la majorité des technologies

existantes (**MPICH-GM** [30], **SCI-MPICH** [21], etc.) Toutefois, l'extrême généricité des mécanismes proposés, sorte de dénominateur commun à toutes les technologies, ne permet pas toujours à de telles interface d'exploiter toutes les fonctionnalisés et les subtilités du matériel sous- jacent.

Récemment, de nombreux efforts de recherche se sont donc focalisés sur la recherche d'un meilleur compromis entre portabilité et performance. Ces efforts ont abouti à la définition d'interfaces de communication de niveau intermédiaire qui, moyennant l'utilisation de paradigmes de programmation un peu plus complexes que le traditionnel échange de messages, sont capables d'effectuer des optimisations spécifiques au matériel (e.g. **VIA** [26, 22], **FAST MESSAGE** [33], ou encore **NEXUS** [28]).

3.1.2 L'optimisation des transferts de données

Parmi l'ensemble des interfaces de niveau intermédiaire existantes, **MADELEINE** [20] est sans doute l'une des plus sophistiquées. Cette interface permet en effet de dissocier complètement la description de la structure des messages à transmettre et le transfert effectif des données sur le réseau. Ce fonctionnement est rendu possible en permettant en particulier à l'application de spécifier des contraintes et des tolérances quant à la façon dont les données doivent êtres rendues disponibles à la réception. Il s'agit d'un contrat passé avec la bibliothèque sous-jacente : l'application exige des propriétés mais s'engage également à s'interdire certaines opérations pendant certaines phases, et en contrepartie la bibliothèque peut ajuster sa stratégie d'optimisation en fonction du matériel réseau tout en respectant les limites fixées par l'appli- cation.

La définition et l'implémentation de cette interface ont fait l'objet de la thèse d'Olivier Aumage [20]. MADELEINE est utilisée par plusieurs équipes de recherche en France et à l'étranger, et sert de fondation à des environnements logiciels complexes (**PM2** [29], **PADICOTM** [25], **HYPERION** [34]). PM2 étant le plus sophistiqué et le plus récent, il a été automatiquement choisi pour notre implémentation.

3.1.3 Techniques utilisées au sein d'interfaces de communication

On va dresser un panorama des principales techniques utilisées au sein des interfaces de communication pour obtenir des performances élevées sur réseaux rapides.

3.1.3.1 Communications depuis l'espace utilisateur

L'accès à une carte réseau est normalement réservé au système d'exploitation : une application ne peut dialoguer avec une carte sans son intermédiaire. Par conséquent, l'émission ou la réception d'un message depuis une application requiert un appel système. Cependant ces derniers sont trop coûteux pour être conservés dans une interface de communication pour réseaux rapides. Pour illustrer, avec un noyau LINUX, le coût minimal d'un appel système se mesure en milliers de cycles sur un processeur INTEL PENTIUM IV, soit environ une microseconde, ce qui est comparable à la latence du réseau QUADRICS.

La solution, déjà utilisée pour les cartes graphiques, est de procéder à un court-circuitage du système d'exploitation pour accéder directement à la carte réseau. Cette stratégie nécessite l'utilisation d'une extension du noyau fournie avec l'interface de communication [1] . Toutes les interfaces de communication contemporaines permettent des communications en espace utilisateur.

[1] À l'initialisation du programme, cette dernière établit des projections de régions mémoire qui permettent de dialoguer avec la carte dans l'espace d'adressage de l'application. Une fois ces projections établies, l'application peut alors lire et écrire dans les registres de la carte et donc la contrôler directement, c'est-à-dire sans appel système. C'est ce que l'on appelle les communications en espace utilisateur.

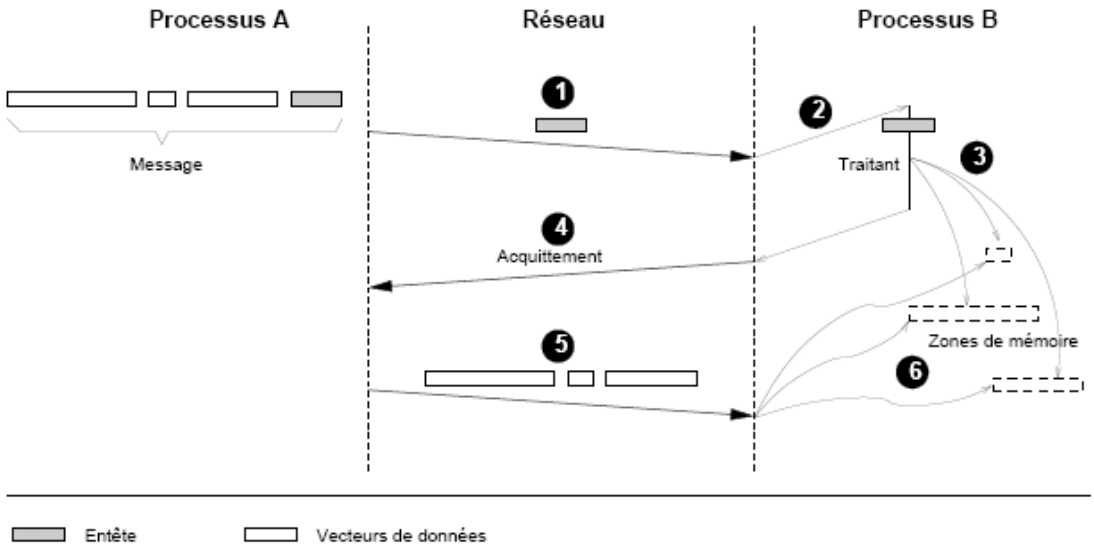

Fig. 3.1-Schéma de transmission d'un message avec Madeleine

3.1.3.2 Transmissions de données locales par PIO et DMA

Il existe deux moyens de transférer des données entre une carte et la mémoire d'un PC. La technique par entrées/sorties programmées par le processeur, appelé transfert **PIO** (Programmed Input/Output) fait transiter les données dans une zone de mémoire réservée au système d'exploitation avant qu'ils n'arrivent à la carte (ou inversement, à la mémoire). La technique par accès direct à la mémoire, appelé transfert **DMA** (Direct Memory Access), s'effectue directement sans l'aide passer par le processeur. Pour faire un transfert, le processeur commence par initialiser les registres du contrôleur DMA avec l'adresse mémoire source, la longueur à transférer, et l'adresse de destination. Le contrôleur DMA lève une interruption une fois le transfert accompli. L'opération initiée, le processeur est alors libre d'effectuer d'autres tâches.

L'emploi de l'un ou de l'autre est généralement choisi par l'interface de communication pilotant la carte, à partir de la taille de données à traiter.il est préférable d'employer le transfert PIO pour les messages courts et le transfert DMA pour les plus longs.

3.1.3.3 Transfert de données vers une carte réseau sans copie intermédiaire

La principale cause de surcoût logiciel que les bibliothèques de communication tentent d'éliminer concerne les recopies mémoire intermédiaires des données qui sont parfois effectuées en émission ou en réception. Pour éliminer ces recopies « superflues » (au moins pour les messages de grande taille), il faut faire en sorte que la bibliothèque soit capable d'assurer un transfert direct des données de l'application vers la carte réseau en émission, et que la carte réseau destinataire soit capable de déposer directement les données en mémoires à l'emplacement stipulé par l'application (cf. figure 3.1).

Il existe d'autres critères importants à satisfaire pour une interface de communication évoluée digne de ce nom tels que le transfert atomique de données non contiguës en mémoire et le multiplexage des communications… . Il serait très fastidieux de les détailler tous et surtout hors du sujet de ce stage. L'étude qui a précédé montre la place de choix qu'occupe madeleine dans le domaine des interfaces de communication ce qui justifie pleinement son choix pour implémenter notre extension de la bibliothèque BSMLlib et DMMLlib. La conception de la bibliothèque MADELEINE n'est pas une initiative de recherche isolée mais s'inscrit au contraire dans le contexte d'un projet de recherche beaucoup plus large autour de l'élaboration d'exécutifs destinés à la programmation distribuée et notamment autour du développement de l'environnement multi-thread **PM2** [29, 38]. On a évidemment opté pour sa suite logicielle pour nos implémentations.

3.2 Évolutions de PM2

PM2 est originellement un environnement de programmation basé sur le paradigme des appels de procédures à distance ou Remote Procedure Call (**RPC**). La première version de PM2 a été réalisée par Raymond NAMYST dans le cadre de sa thèse de doctorat ([29]) en 1995 et son but était de permettre l'exécution parallèle d'applications fortement irrégulières en virtualisant l'architecture sous-jacente au moyen de processus légers migrables. L'environnement était donc fondé sur deux bibliothèques : d'une part la bibliothèque **Marcel** pour la partie processus légers et d'autre part **PVM** pour la partie communications.

Cet environnement a évolué car les performances des communications étaient insuffisantes. PVM a donc été remplacée par une seconde bibliothèque de communication dédiée, MADELEINE. L'introduction de Madeleine a entraîné une mutation car il devenait essentiel que les deux aspects (multithreading et communications) soient capables de coopérer efficacement.

Progressivement, PM2 est passé du statut d'environnement de programmation destiné à des utilisateurs finaux à celui de support d'exécution (runtime system) pour des couches logicielles de plus haut niveau comme des intergiciels (middleware), en particulier MPI ou bien CORBA. Il est cependant important de comprendre que s'il est toujours possible d'utiliser indépendamment l'une de l'autre la bibliothèque Marcel et Madeleine, leurs comportements diffèrent quand elles sont employées conjointement. Ainsi, Madeleine est une bibliothèque de communication compatible avec les mécanismes du multithreading. Ce point est essentiel et fait écho à la remarque concernant la difficile intégration d'éléments logiciels distincts. Dans le cas de Marcel et Madeleine, cette intégration existe et est effective. C'est cette version de PM2 (ou plutôt de Marcel + Madeleine) que nous avons employée pour implémenter notre logiciel. Une dernière remarque concernera les RPC: si ce paradigme est toujours disponible dans l'actuel PM2, nous ne l'utilisons pas pour notre mise en œuvre.

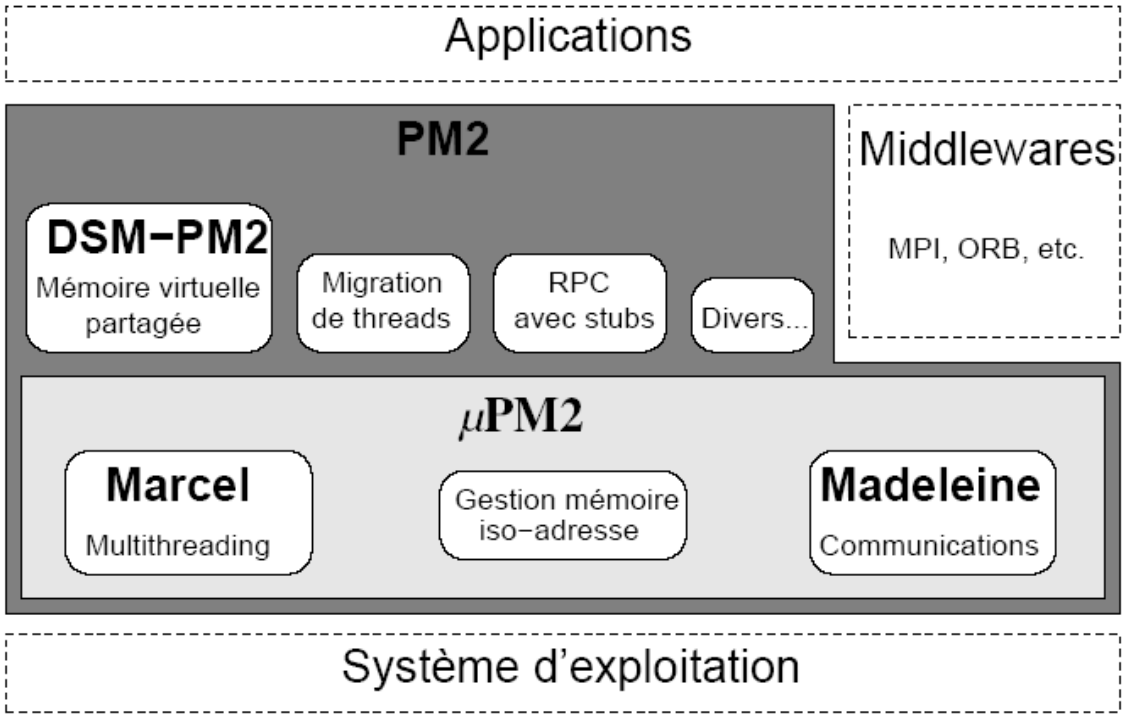

Fig. 3.2-PM2

3.3 La bibliothèque de communication Madeleine

Nous décrivons maintenant la bibliothèque de communication Madeleine. Après un tour d'horizon de ses traits généraux et des critères qui justifient son choix, nous nous attardons sur son interface avant de donner un petit exemple concret d'utilisation. Nous faisons également le point sur les capacités de Madeleine en ce qui concerne la gestion des configurations de type «grappes de grappes». Madeleine est le cœur du travail de thèse d'Olivier AUMAGE [20].

3.3.1 Caractéristiques de Madeleine

Madeleine est une interface de communication portable destinée à l'exploitation des grappes de PC inter connectés par des réseaux hauts-débits. Comme il s'agit en fait du sous-système de communication de PM2, elle est donc optimisée pour une utilisation conjointe avec Marcel.

Madeleine possède des propriétés nous intéressant directement :

- Madeleine est **multithread**, aussi bien dans sa conception que dans son utilisation, ce qui signifie que non seulement elle utilise les processus légers pour mettre en place certaines fonctionnalités (par exemple le service de retransmission des messages) mais qu'elle est aussi réentrante, permettant à plusieurs processus légers d'effectuer de façon concurrente des opérations de communication;

- Madeleine est **multi-paradigme** c'est-à-dire qu'elle permet l'emploi, pour chaque protocole supporté, de différentes méthodes de transfert (envoi de messages, écriture dans une zone de mémoire distante, DMA, etc.);

- Madeleine est **multi-protocole** car elle permet aux applications de gérer simultanément plusieurs protocoles réseaux différents.

Madeleine a été portée au-dessus de nombreux protocoles et interfaces de communication : TCP, UDP, VRP, BIP, GM, SBP, VIA, SISCI et même. . . MPI [30]. Les performances affichées par Madeleine sont de tout premier plan, en particulier dans le domaine des réseaux rapides.

Enfin, Madeleine peut être considérée comme une suite logicielle plus que comme une bibliothèque de programmation, car elle utilise des outils annexes très pratiques, comme un analyseur de fichiers de configuration et un lanceur d'application (le logiciel **Léonie**, cf. figure 3.3).

3.3.2 Interface de Madeleine

L'interface de Madeleine est plutôt orientée vers le passage de messages, avec un nombre restreint de fonctions. Nous allons les détailler ainsi que les différents modes de construction des messages.

3.3.2.1 Fonctions d'émission et de réception

L'interface de MADELEINE se restreint à 6 fonctions (cf. figure 3.4) : Début de construction d'un nouveau message, Acceptation d'un message entrant; Finalisation d'une construction de message, Finalisation de la réception d'un message, Empaquetage d'un bloc de données et Dépaquetage d'un bloc de données. Toutes ces fonctions correspondent à des appels **bloquants**, ce qui peut sembler un problème à première vue car par exemple MPI possède tout un ensemble d'opérations de communication non bloquantes. Cependant, comme nous utilisons également des processus légers, cette difficulté peut être résolue.

3.3.2.2 Modes d'empaquetage et de dépaquetage

MADELEINE permet à l'application de spécifier des contraintes quant à l'émission et la réception des données transmises. Par exemple, lors d'une opération d'empaquetage **mad_pack**, il est possible d'imposer que les données soient immédiatement disponibles du côté récepteur lors de l'opération **mad_unpack** correspondante. Au contraire, on peut relâcher complètement cette contrainte de disponibilité pour permettre à MADELEINE d'optimiser au maximum le mode de transmission en fonction du réseau sous-jacent. L'expression de telles contraintes par l'application constitue véritablement le point clé permettant l'obtention de bonnes performances tout en utilisant une interface complètement générique. La liste des différentes contraintes supportées par MADELEINE en matière d'empaquetage/dépaquetage des données est la suivante du côté émetteur :

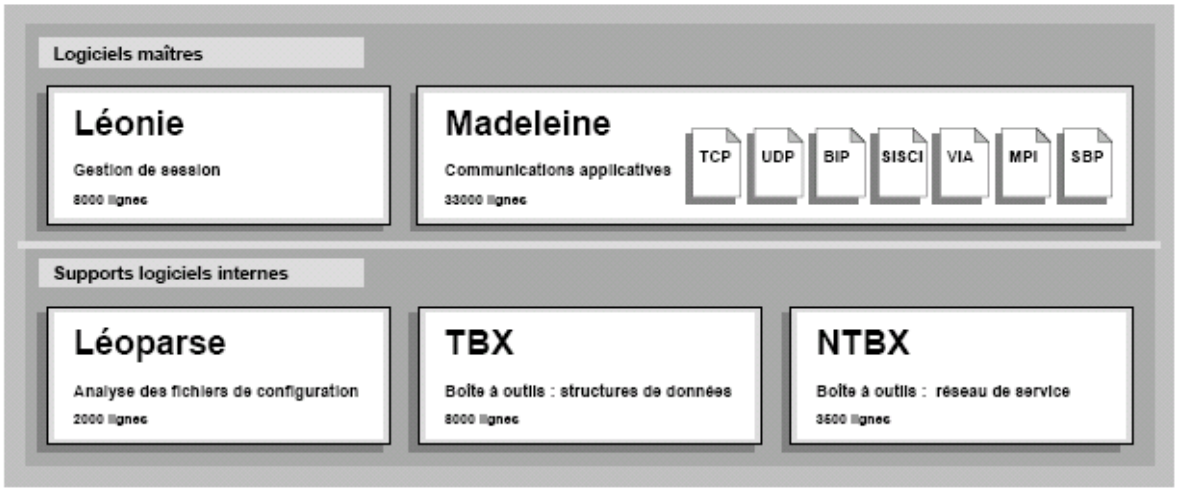

Fig. 3.3-Madeleine

send_safer :

Indique que MADELEINE doit empaqueter les données de telle manière que des modifications ultérieures de la zone mémoire correspondante ne puissent pas modifier le message. C'est en particulier nécessaire si la zone des données à émettre est réutilisée avant que le message ne soit envoyé ;

send_later :

Indique que MADELEINE ne peut pas accéder aux données avant que la fonction ***mad_end_packing*** ne soit appelée. Ceci signifie que toute modification de ces données effectuée entre leur empaquetage et leur envoi entraîne une mise à jour du contenu du message;

send_cheaper:

C'est le mode par défaut. Il autorise MADELEINE à faire de son mieux pour traiter les données le plus efficacement possible. La contrepartie est qu'aucune hypothèse ne peut être faite sur le moment où MADELEINE accède aux données. Par conséquent, celles-ci doivent rester inchangées jusqu'à la fin de l'opération d'envoi.

```
p_mad_connection_t
mad_begin_packing(p_mad_channel_t channel,mad_process_lrank_t remote_rank);
```

```
Void
mad_pack(p_mad_connection_t connection,

          void *buffer,
          size_t buffer_length,
          mad_send_mode_t send_mode,
          mad_receive_mode_t receive_mode);
```

```
Void
mad_end_packing(p_mad_connection_t connection);
```

```
p_mad_connection_t
mad_begin_unpacking(p_mad_channel_t channel);
```

```
Void
mad_unpack(p_mad_connection_t connection,

           void *buffer,
           size_t buffer_length,
           mad_send_mode_t send_mode,
           mad_receive_mode_t receive_mode);
```

```
Void
mad_end_unpacking(p_mad_connection_t connection);
```

Fig. 3.4-primitives Madeleine

Du côté récepteur on a les modes suivants :

receive_express :

Force MADELEINE à garantir que les données correspondantes sont immédiatement disponibles après l'opération de dépaquetage ***mad_unpack***. Typiquement, ce mode est obligatoire si les données sont nécessaires pour procéder aux opérations de dépaquetage suivantes. Avec certains protocoles réseau cette fonctionnalité n'est pas très coûteuse alors qu'avec d'autres, cela peut avoir des répercussions sensibles, tant sur le plan de la latence que du débit.

receive_cheaper :

Autorise MADELEINE à différer l'extraction des données correspondantes jusqu'à l'exécution de la fonction ***mad_end_unpacking***. Par conséquent, aucune hypothèse ne peut être émise sur le moment exact où les données seront extraites. MADELEINE optimise automatiquement le temps de transmission des messages (cela dépend du réseau sous-jacent).

3.3.3 Exemple pratique d'utilisation de Madeleine

La figure 3.5 donne un exemple d'utilisation de l'interface de MADELEINE. Nous souhaitons envoyer un tampon d'octets dont la taille est inconnue du processus récepteur. Celui-ci doit donc extraire la taille du tampon (un entier) avant d'extraire le tampon lui-même, parce que la zone de destination doit être allouée dynamiquement. Dans cet exemple, la contrainte est que l'entier indiquant la taille doit être extrait en mode EXPRESS avant de pouvoir dépaqueter les données correspondantes. Ces dernières peuvent quant à elles être extraites en mode CHEAPER, pour plus d'efficacité.

3.3.4 Support des configurations hétérogènes et des grappes de grappes

3.3.4.1 Idées et concepts principaux

Madeleine possède également des fonctionnalités destinées à l'exploitation des grappes de grappes [20, 21], en plus des caractéristiques multi-protocoles de Madeleine ce qui justifie pleinement son choix pour l'implémentation de DMMLlib qui un des objectifs de ce stage. Pour ce faire, Madeleine introduit la notion de réseau virtuel hétérogène. Lorsque des grappes distinctes sont inter- connectées, Madeleine utilise les réseaux locaux d'interconnexion pour en construire un plus large, englobant l'ensemble de ces grappes. Grace à ce support muti-protocole, Madeleine autorise l'utilisation des réseaux rapides pour les communications intra-grappes.

Du côté des communications inter-grappes, Madeleine est très souple : les schémas de communication peuvent être soit directs, soit avec des retransmissions, car un service de passerelle est disponible. Ces passerelles logicielles permettent à un nœud équipé de plusieurs technologies réseaux distinctes de procéder à une retransmission des messages émanant d'un processus appartenant à une première grappe et destinés à un processus membre d'une autre grappe. Ce mécanisme permet donc de ne pas recourir uniquement à TCP dans le cas des communications inter-grappes et les liens rapides sont alors pleinement exploitables. Ces passerelles retransmettent non seulement les messages mais jouent également le rôle de routeurs. On peut donc cascader de tels routeurs. Ces routeurs sont implémentés à l'aide de processus légers, ce qui permet de mettre en place des pipe-lines logiciels.

Ces mécanismes sont totalement transparents pour l'application, lui donnant l'impression de communiquer sur un unique réseau global, qu'il est alors possible de qualifier de réseau virtuel hétérogène. La création d'un tel réseau virtuel ainsi que la manipulation des différents protocoles se fait par l'intermédiaire d'objets appelés des canaux. Madeleine distingue au niveau applicatif deux types de canaux : les canaux physiques et les canaux virtuels (cf. figure 3.6) :

- Les Canaux physiques instancient un protocole de communication utilisé pour l'exploitation d'un réseau physique. Par exemple, si une machine dispose d'une carte **Myrinet**, il sera possible de créer plusieurs canaux physiques correspondants à cette technologie. Ce mécanisme peut être considéré comme une forme de multiplexage;
- Les Canaux virtuels sont eux construits à partir de canaux physiques uniquement (on ne peut donc pas mettre en place un canal virtuel au-dessus d'un autre canal virtuel). Cette construction se base sur des fichiers de configuration écrits par l'utilisateur, et s'effectue au démarrage de l'application. C'est lors de cette phase que sont déterminées les passerelles entre les différents canaux physiques (i.e. les différentes instances des protocoles réseaux) et ce choix n'est pas remis en cause ultérieurement (le routage est donc statique). Les canaux physiques qui servent de base à un canal virtuel sont «absorbés» par ce dernier et deviennent par conséquent invisibles pour l'application.

Côté émetteur	Côté récepteur
conn = mad_begin_packing(arguments); **mad_pack(conn,&size,sizeof(int),** **send_CHEAPER,receive_EXPRESS);** **mad_pack(conn, array, size,** **send_CHEAPER,receive_CHEAPER);** **mad_end_packing(conn);**	**conn = mad_begin_unpacking(arguments);** **mad_unpack(conn,&size,sizeof(int),** **send_CHEAPER,receive_EXPRESS);** **array = malloc(size);** **mad_unpack(conn, array, size,** **send_CHEAPER,receive_CHEAPER);** **mad_end_unpacking(conn);**

Fig. 3.5-Envoi de message avec MADELEINE.

3.3.4.1 Un exemple concret d'utilisation des canaux

Afin de mieux appréhender comment s'utilisent concrètement ces fonctionnalités, nous allons donner un petit exemple. Supposons que l'on veuille inter connecter deux grappes de six nœuds chacune. La grappe A (avec les nœuds G1, G2,… et G6) possède un réseau rapide de type Myrinet (avec le protocole GM). Quant à la grappe B, H1, H2,… et H7 possèdent un réseau rapide de type SCI (avec le protocole SISCI). Nous allons également supposer qu'un nœud passerelle dispose aussi d'une carte SCI et que TCP/Ethernet est disponible sur l'ensemble des douze nœuds (cf. figure 3.7).

L'utilisateur commence par écrire un fichier décrivant les technologies disponibles sur les différents nœuds puis il fournit ensuite un fichier de description des canaux (cf. figure 3.8). Ce sont ces deux fichiers qui vont être utilisés par l'application afin de construire ses canaux de communication. On voit que dans notre exemple, nous construisons trois canaux physiques correspondant aux réseaux sous-jacents. Ensuite nous créons un canal virtuel au dessus des deux canaux physiques des réseaux rapides. Au niveau applicatif, nous allons donc pouvoir utiliser deux canaux : soit le canal physique correspondant à TCP, soit le canal virtuel correspondant au réseau virtuel hétérogène Myrinet/SCI. Dans le cas d'une communication entre un nœud de la grappe A est un nœud de la grappe B, si nous choisissons d'opter pour le canal virtuel, alors tous les messages vont d'abord transiter via le réseau Myrinet pour être acheminés vers le nœud qui joue le rôle de passerelle. Ce nœud retransmet ensuite le message vers son véritable destinataire en utilisant cette fois-ci le réseau SCI. Dans un tel cas de figure, il est donc possible de se passer totalement de TCP.

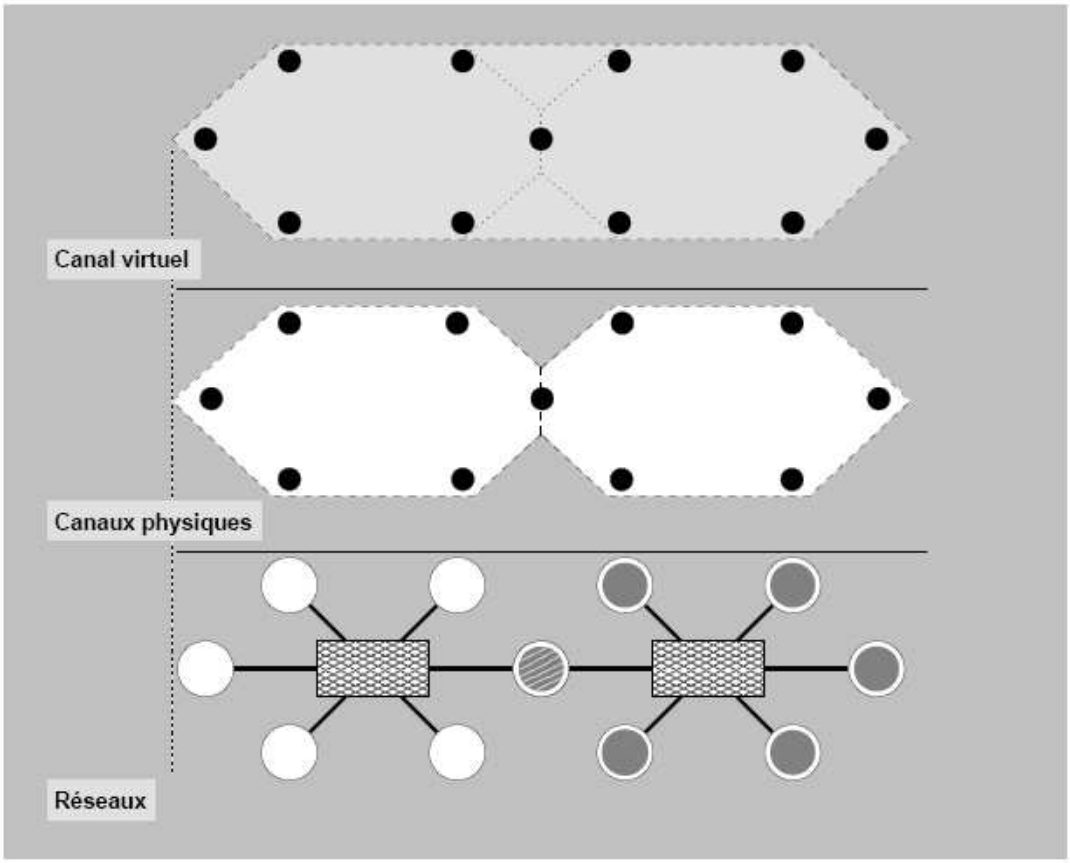

Fig. 3.6-Exemple de canaux physiques et virtuels dans Madeleine

3.4 La bibliothèque de processus légers Marcel

Après cette rapide description de la bibliothèque de communication Madeleine, vient le tour de l'autre pilier logiciel de PM2 : la bibliothèque de processus légers Marcel. La thèse de Vincent DANJEAN est consacrée à Marcel [27].

Marcel est une bibliothèque de processus légers de niveau utilisateur : la conséquence est que les temps de création, de destruction et de changement de contexte entre les processus sont très courts en comparaison des processus légers de niveau noyau. Ceci est aussi vrai pour les opérations de synchronisation car elles ne font pas appel au noyau du système d'exploitation. Marcel présente une interface orientée POSIX et son utilisation conjointe avec Madeleine est optimisée. Marcel possède également des fonctionnalités particulières, car son Ordonnanceur est adaptable selon l'architecture : dans le cas d'une machine uniprocesseur, il s'agira d'un ordonnanceur en espace utilisateur uniquement tandis que pour une meilleure exploitation des machines multi-processeurs, Marcel utilisera un ordonnanceur hybride à deux niveaux. Un tel ordonnanceur couple les processus légers de niveau utilisateur à des processus légers de niveau noyau, reconnus par le système et susceptibles d'être affectés à des processeurs physiques différents. L'ordonnancement de Marcel est préemptif, les processus légers ne sont donc pas dans l'obligation de rendre explicitement la main.

Marcel implémente également un modèle des Scheduler Activations [35], ce qui permet aux processus légers d'effectuer des appels-système bloquants sans pour autant immobiliser tout le processus UNIX auquel ils appartiennent. Enfin l'ordonnanceur de Marcel permet d'effectuer des traitements intéressants dans le cas des communications.

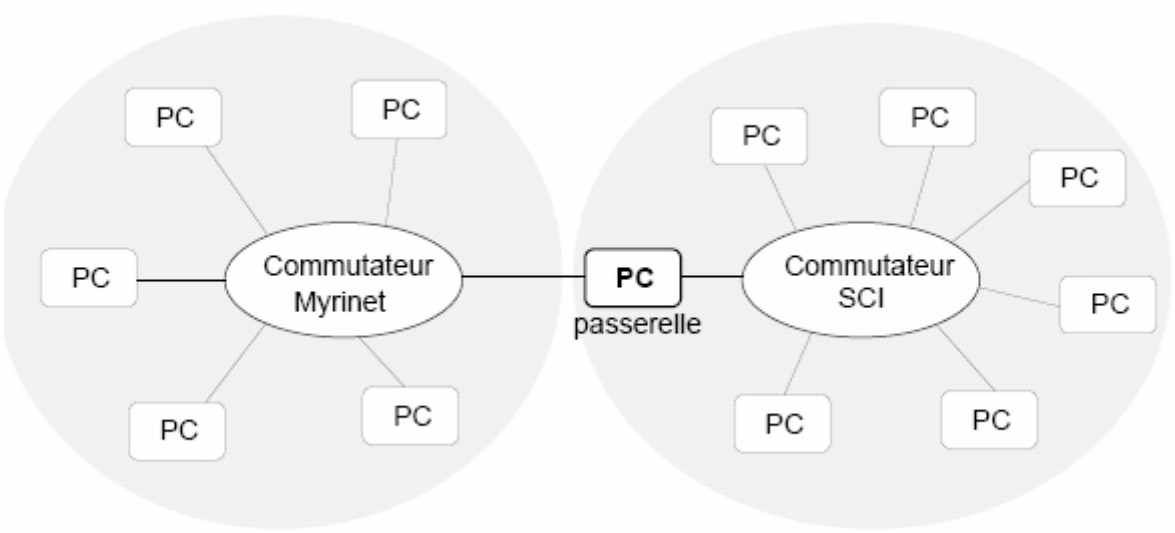

Fig. 3.7-Exemple de grappe de grappes hétérogène

```
# networks.cfg
networks : ({
        name   : bip_net;
        hosts  : (popc0, popc1, popc2, popc3, popc4, popc5,
                  popc6, popc7, popc8, popc9, popc10, popc11,
                  myri0, myri1, myri2, myri3, sci5);
        dev    : bip;
        mandatory_loader : bipload;
}, {
        name   : sisci_net;
        hosts  : (sci0, sci1, sci2, sci3,
                  sci4, sci5, sci6, sci7);
        dev    : sisci;
});
```

Fig. 3.8-Exemple de fichier de configuration réseaux

Chapitre 4

Implémentation de BSMLlib avec PM2

4.1 La bibliothèque BSMLlib modulaire

La version 0.25 de la bibliothèque BSMLlib n'était pas modulaire, au sens des modules d'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 futures 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 4.3. Nous commençons par donner un aperçu du module générique.

4.2 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 2.2, est implémenté en **SPMD** (Single Program Multiple Data) utilisant une bibliothèque de communication de plus bas niveau. Ce module, appelé **Comm**, est basé sur les principaux éléments donnés dans la figure 4.1.

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

Fig. 4.1-Primitives du 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 optionnelles.

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 (nprocs-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 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*** et ***proj*** 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 appelée 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 $\longrightarrow \alpha\ par$ telle que $(f_i(i+1)) = Some\ v_i$ pour i = 0, 1,2 et $(f_i\ j)$ = *None* sinon. L'expression $mkpar(fun\ i \longrightarrow f_i)$ sera évaluée comme suit :

- 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*

- 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*

- 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) = *Some* v2 qui veut dire que le processeur 2 a envoyé un message au processeur 3.

4.3 Les différents modules Comm

4.3.1. Implémentation MPI

Pour la version 0.3, deux implémentations MPI étaient disponibles. La première implémentation utilise une fonction de communication collective de la bibliothèque MPI : **Alltoall**, 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 deuxième implémentation fut abandonnée, car jugée inefficace.

La première étape consiste en l'envoi de la taille des messages qui seront envoyés 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. Dans la deuxième implémentation 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, seule la seconde change.

4.3.2 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 principe du carré latin [6], puisqu'il n y a pas de fonctions collectives.

Ne pas utiliser de fonctions collectives 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) mod p (p est le nombre de processeurs) puis reçoit un message du processeur (i-1) mod p. Ensuite il envoie un message au processeur (i +2) mod p puis reçoit un message du processeur (i -2) mod 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. Chaque processeur i qui veut envoyer un message au processeur (i + 1) mod 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) mod 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.

4.3.3 Implémentation PUB

Il existe deux librairies qui permettent d'implémenter les algorithmes **BSP** : **PUB** [3] de l'université de **Paderborn** et **BSPLib** le l'université d'Oxford. Il existe une implémentation du module Comm en PUB.

La PUB (BSP-Bibliothèque de l'université de Paderborn) est une bibliothèque de routines de communication écrite en C. Ces routines permettent d'implémenter les algorithmes qui sont conçus pour le modèle de BSP. PUB offre des fonctions pour le *message passing* et l'accès mémoire à distance. En outre, certaines opérations collectives de communication comme le *broadcast* et le parallel prefix sont également fournies. Pour devenir plus flexible, PUB laisse créer des objets BSP indépendants qui représentent chacun un ordinateur virtuel BSP.

A l'instar de MPI, l'implémentation est facilitée par les fonctions de synchronisation de la PUB comme **bsp_oblsync** qui est utilisée à la place de **bsp_sync**[1]. Chaque processeur effectue ses envoies aux autres processeurs en une seule fois, par le biais da la fonction **`bsp_hpsend`**[2] `et` d'une boucle, avant d'appeler les fonctions de synchronisation.

4.3.4 Implémentation MADELEINE

Le principe de l'implémentation est le même que celui de MPI. Le travail effectué consistait à produire une partie du code en Ocaml qui implémente les fonctions du module Comm, notamment la fonction **send**, qui fait appel à son tour à des fonctions de la bibliothèque Madeleine écrite dans la partie C.

4.3.4.1 Partie OCAML du code

La communication entre nœuds se fait en deux étapes : dans un premier temps les processeurs s'envoient les tailles des messages. Vue l'absence de fonctions collectives de communication on peut utiliser dans ce cas afin d'ordonnancer l'ordre des envois le principe du carré latin qui a été utilisé précédemment. Puis dans une seconde étape les messages sont envoyés selon le même ordonnancement.

Le mode opératoire est le même que celui utilisé pour MPI, sauf qu'avec Madeleine on peut se passer de la première phase, et envoyer directement les messages. Dans ce cas c'est au niveau du code C et précisément de l'empaquetage des messages que l'on va empaqueter dans un premier temps la taille du message en mode EXPRESS, pour que cette donnée soit disponible immédiatement après, pour l'allocation de la mémoire nécessaire, puis dans un deuxième temps le message sera lui empaqueté en mode CHEAPER, pour une optimisation maximale. L'envoi et la réception des longueurs des messages a été gardé pour être utilisé lors du contrôle sur la taille des données avant leur envoi, afin

[1] Puisqu'on sait exactement combien de message chaque processeur reçoit dans une super-étape

[2] Comme la fonction **bsp_send**, envoie un message à partir d'un buffer mais le protège contre toute modification.

d'éviter l'envoi de chaines vides[1].

La version 0.3 de la BSMLlib, sur laquelle on a travaillé est la plus récente, elle est caractérisée par une conception modulaire qui utilise la notion de **foncteurs**, ce qui rend la modification et la maintenance du code relativement aisées. Dans le cas échéant l'implantation de madeleine pour gérer les communications inter-processeur concerne essentiellement le module Comm.

L'implémentation commence par la définition des paramètres globaux et des constantes utilisées notamment les fonctions **pid** et **nprocs** qui se résument à un appel aux fonctions externes en C **Madeleine_pid** et **Madeleine_nprocs**, par les déclarations :

```
(* ############################ Parameters ################################### *)
 external pid : unit -> int = "Madeleine_pid"
 external nprocs : unit -> int = "Madeleine_nprocs"
(* ########################################################################### *)

(* ############################ Constants ################################### *)
 let noSome (Some x) = x
 let args = ref [||]
(* ########################################################################### *)
```

Aussi, l'implémentation du module Comm comporte le corps de plusieurs fonctions tels que **wtime** qui donne l'heure system, et fait appel pour cela à la fonction **gettimeofday** du module Unix. La fonction **abort** quant à elle permet de quitter proprement la session.

```
let wtime  = Unix.gettimeofday

external abort : int ⟶ unit = "Madeleine_abort"
```

Les fonctions **Initialize** et **Finalize** permettent respectivement l'initialisation et la terminaison du programme en appelant des fonctions appropriées écrites en C, celles-ci font appel à leur tour à des fonctions de PM2 pour l'initialisation des bibliothèques et des structures de données, elles seront décrites dans la section suivantes.

```
external finalize : unit ⟶ unit = "Madeleine_finalize"
external initialize : unit ⟶ unit = "Madeleine_init"
```

Les communications inter-processeurs sont implémentées par la fonction **send** qui requière un peu plus d'attention. Pour des raisons évidente d'interfaçage entre le code écrit en C et celui écrit en Ocaml, on commence par la déclaration des fonctions externes C utilisées dans la fonction comme suit :

```
external bsmlMadeleine_send_int : int -> int -> unit = "Madeleine_send_int"
external bsmlMadeleine_receive_int : int ->int -> int = "Madeleine_receive_int"
external bsmlMadeleine_send_str : string -> int-> unit = "Madeleine_send_str"
external bsmlMadeleine_receive_str : string -> int -> int -> string = "Madeleine_receive_str"
```

La fonction **send** prend, pour un processeur donné, en argument un tableau de valeurs optionnelles qui représentent les valeurs à envoyer aux autres processeurs, et retourne un tableau représentant les valeurs reçues. On commence d'abord par contrôler si l'argument est de dimension valide (i.e. égale à **nprocs** ()).

[1] L'envoi de chaines vide en Madeleine représente un coût non négligeable.

Ensuite on procède au « ***Marshaling »*** des données à envoyer qui peuvent être de natures différentes, on obtient ainsi un tableau de **string** : ***buffers*** grâce à une fonction de **filtrage** comme suit :

```
Let buffers = Array.map (fun d -> match d with
                                    None     -> ""
                                  | Some x -> Marshal.to_string  x [Marshal.Closures]) data
```

La déclaration **« let sendlengths = Array.map String.length buffers »** permet d'obtenir un tableau des longueurs à envoyer, on fait de même pour les longueurs des valeurs à recevoir. Cela permet de faire des contrôles avant appeler les fonctions de Madeleine d'envoie et de réception, parce que l'envoi de chaines vides lance comme même les routines de communication de madeleine.

Afin d'éviter les inters blocages on utilise deux fois le principe du carré latin évoqué précédemment pour l'envoi et la réception des messages et de leurs longueurs. Ainsi pour obtenir le tableau qui représente les longueurs des valeurs en réception, on utilise la boucle suivante :

```
/*carre latin pour l'envoi et la réception des longueurs de messages*/

    for i=0 to (nprocs()) - 2 do
     let destination =  (pid() + i +1) mod (nprocs()) in
       Madeleine_send_int sendlengths.(destination) destination;
       recvlengths.((nprocs()+pid() -i -1) mod (nprocs())) <- Madeleine_receive_int
                                                              ((nprocs()+pid() -i -1) mod (nprocs())) (pid());
     done;
```

On procède après à la création d'un tableau de retour vide de longueur nprocs () avec la fonction **Array.make.** Lors de l'envoi et la réception des messages, pour chaque entrée du tableau **sendlengths** qui n'est pas nulle on procède à un envoi par l'appel de la fonction **Madeleine_send_int**, et pour chaque entrée non nulle du tableau **recvlengths** on appelle la fonction de réception qui retourne la chaine reçue et la met dans une variable temporaire, avant qu'elle soit « **démarshalisé** » et repositionnée dans le tableau **retour**, qui est finalement renvoyé par la fonction **send** (cf. figure 4.2).

4.3.4.2 Partie C du code

La partie la plus fastidieuse du travail fut l'installation et la configuration de PM2 sur les nœuds, et les paquetages nécessaires à son fonctionnement. Celui-ci nécessite de renseigner plusieurs variables d'environnement et de construire une base de données de ***flavors***[1]. Aussi la compilation d'un fichier sous PM2 n'est pas trivial, heureusement PM2 Offre un utilitaire de réparation par la commande **make sos** et un mode débogage [29]. L'exécution d'un fichier source se fait par l'appel de la fonction **pm2load**, après avoir spécifié les nœuds sur lesquels le code doit s'exécuter par la commande **pm2conf hostname.**

Le point d'entrée d'un programme PM2 sur chaque nœud est la fonction **pm2_main**, qui s'utilise en lieu et place d'un main classique. Dans le cas où on utilise le mode madeleine, le fichier source doit inclure le fichier **madeleine.h** et un main classique ; le pm2_main étant fourni par la bibliothèque au moment de la compilation. Un des problèmes rencontré lors de l'implémentation des fonctions externes en C, est la notion de numéro de processus local et global utilisé par Madeleine.

---

[1] Sorte de modes de configuration en PM2 qui permettent de définir des options à la compilation et à l'exécution et utiliser soit madeleine : flavor mad3, marcel : flavor marcel ou les deux à la fois : flavor pm2.

```
    let temp = retour.(pid()) <- data.(pid()) in

  /*carre latin pour l'envoi et la réception des messages*/
  /* à chaque fois on teste la longueur du message avant envoi ou réception*/

      for i=0 to (nprocs()) - 2 do
       let destination =  (pid() + i +1) mod (nprocs()) in
        if sendlengths.( destination  ) <>0 then
         Madeleine_send_str buffers.(destination)
              destination sendlengths.(destination);

        if recvlengths.( (nprocs()+pid() -i -1) mod (nprocs()))<>0 then
         let index = (nprocs()+pid() -i -1) mod (nprocs()) in
         let receivebuffer = String.create recvlengths.(index) in
         let res = bsmlMadeleine_receive_str
                                  receivebuffer index
                                  recvlengths.(index)  in
          retour.(index) <- Marshal.from_string receivebuffer 0;
      done;
    retour
```

Fig. 4.2-Envoi et réception des messages par carré latin

Madeleine utilise une numérotation globale des processus qui est unique au cours de la session. Après l'initialisation de l'objet madeleine chaque processus se voit attribué un numéro unique **g_rank**. En outre Madeleine procède à une numérotation des processus communicant dans un canal, à l'image d'un canal classique en télécommunication, Ce procédé permet un cloisonnement des communications et évite qu'elles n'interfèrent. Pour que deux nœuds puissent communiquer il faut qu'ils appartiennent au même canal, et qu'ils connaissent leurs numéros locaux **l_rank**. Lors de chaque fonction de communication entre deux nœuds on a procédé à la conversion des numéros globaux en numéros locaux par la fonction **ntbx_pc_ global_to_local**. La bibliothèque Madeleine maintient une table de correspondance entre les **g_rank** et les **l_rank** comme une table de routage et offre tout un panel d'outils et de fonctions de conversion.

Les fonctions d'envoi et réception d'entiers ne présentent aucune difficulté et repose sur l'utilisation du mode *Express* pour l'empaquetage et le dépaquetage des données. Tandis que les fonctions d'envoi (resp de réception) des chaines de caractères ont nécessité deux appels à la fonction **pack** (resp **unpack**), un pour la taille de la chaine et le deuxième pour le message lui-même (cf. figure 4.3).

Dans l'annexe on décortique la partie C de notre implémentation avec les quatre fonctions d'envoi et de réception mentionnées précédemment.

Chapitre 5

Implémentation de DMMLlib avec PM2

5.1 La bibliothèque DMMLlib modulaire

Une version de test de l'implantation des primitives DMML a été faite par Frédéric GAVA dans le cadre de sa thèse ([34]). Celle-ci repose sur l'utilisation de MPI au niveau des unités BSP et TCP/IP au niveau du réseau départemental. L'utilisation de TCP/IP oblige la création d'un réseau privé virtuel **VPN** (*virtual private network en anglais*) pour avoir des adresses IP compatibles à tous les nœuds du méta-ordinateur. L'implantation TCP/IP est très proche de celle de **MSPML** et le code de la BSMLlib a été directement utilisé pour les communications BSP.

Cette première version de la DMMLlib, à l'instar de la première BSMLlib, est modulaire est utilise les foncteurs du langage Ocaml. Un module générique implémentant les primitives décrites dans la section 2.3 et prend en argument deux modules de communication : Un module **BSP_Comm** pour les communications au sein d'une unité BSP, et le module **MPM_Comm** pour les communications inter-grappes. L'implémentation des modules génériques est indépendante de celle des deux modules de communication mais dépend de leurs interfaces qui sont décrites dans la figure 4.3. L'interface du module BSP_Comm reprend en tous point celle du module Comm de la BSMLlib, tandis que celle du MPM_Comm est plus complexe.

Le nombre de clusters de la machine est donné par la fonction **nclus**, le rang de la m-étape est donné par la fonction **dstep**, et l'identifiant de l'unité BSP par **cluspid**. L'implémentation des types abstraits, ***mkpar*** et ***apply*** est la suivante :

```
type α par = α
let mkpar f = f (BSP_Comm.pid ())
let apply f v = f v
```

Les fonctions ***put*** et ***proj*** dans une unité BSP sont implémentées avec la fonction ***send***, comme dans la BSMLlib. Au niveau départemental, les types abstraits **mkdep** et **applydep** sont implémentés comme suit :

```
type α dep = α
let mkdep f = f (MPM_Comm.cluspid ())
let applydep f v = f v
```

L'implémentation de la fonction **get** fait appel à plusieurs nouvelles fonctions et à l'utilisation des librairies pré chargées d'Ocaml pour la programmation concurrente, en particulier **THREAD** pour la création de thread et **MUTEX** pour créer des verrous entre processus en relation d'exclusion mutuelle sur une zone de mémoire. L'exemple qui suit montre une implémentation possible des fonctions ***get*** et ***put***.

Exp:

```
let get fi fv =
   Lib_Comm.print_verbose "Here a GET\n";
   let requested = Array.init (dm_p())
                   (fun clus -> Array.init (dm_bsp_p clus) (fi clus)) in
   let result = get_array fv requested in
    fun clus proc ->
      if ((0<=clus) && (clus<dm_p()) && (0<=proc) && (proc<(dm_bsp_p clus)))
       then (result.(clus)).(proc)
      else None
     end
```

```
let put f =
  let data = Array.init (bsp_p()) (fun i->f i) in
  let res = Lib_Comm.send data in
   fun i -> if ((0<=i) && (i<bsp_p()))
            then res.(i)
              else None
```

L'implémentation de la DMMLlib en utilisant PM2 et son interface de communication Madeleine se résume à l'implémentation de ses deux modules de communication BSP_Comm et MPM_Comm à l'aide de Madeleine.le module BSP_Comm étant la reprise du module Comm de la BSMLlib, il ne reste pratiquement rien à faire : il suffit de positionner les fichiers déjà utilisés pour la BSMLlib qu'on a vu dans le chapitre précèdent. Pour en ce qui concerne le deuxième module de communication MPM_Comm on a préféré, dans un premier temps garder la même version, l'implémentation de ce module en utilisant la bibliothèque de threads MARCEL fera l'objet de prochains travaux.

5.2 Perspectives pour la DMMLlib

Afin de fournir une nouvelle implémentation de la DMMLlib efficaces en profitant au mieux des avantages fourni par l'environnement de PM2, Madeleine et la bibliothèque Marcel en ce qui concerne la gestion des threads et aussi pour les communications inter-grappes avec la notion de canaux virtuels. il faudrait réécrire le code du module MPM_Comm en utilisant les fonctions offerts par PM2 et sa bibliothèque MARCEL pour la gestion des threads et les opérations de synchronisation comme positionnement de verrous pour les relations d'exclusion mutuelle sue les zones de mémoire. Une partie qui a été jusqu'à présent implémentée à l'aide des fonctions des modules THREAD et MUTEX pour la programmation concurrente en Ocaml.

Il est aussi dans nos projets d'instaurer un système de fenêtre glissante afin de réduire l'asymétrie de calcul entre les nœuds de façon dynamique.

```
(** Module of BSP Communication *)
module type BSP_Comm =
 sig
  val pid : unit -> int
  val nprocs : unit -> int
  val initialize : string array -> unit
  val finalize : unit -> unit
  val send : 'a option array -> 'a option array
  val wtime : unit -> float
  val abort : int -> unit
 end

(** Module of MPM Communication *)
module type MPM_Comm =
 sig
  type message = Start | Stop | Dstep of int * int
  val cluspid : unit -> int
  val nclus : unit -> int
  val dstep : unit -> int
  val print_verbose : string -> unit
  val print_verbose_err: string -> unit
  val init: unit -> unit
  val run_communication : unit -> unit
  val start_computation : unit -> unit
  val end_computation : Thread.t -> unit
  val store : 'a -> unit
  val request : int -> int -> message -> 'a
  val wtime : unit -> float
  val abort : int -> unit
 end
```

Fig. 4.3-Primitives des modules Comm de la DMMLlib

Conclusions

Le parallélisme et méta-computing sont des thèmes de recherche très actifs auxquels, lors de ma scolarité, je n'avais pas vraiment été initié (excepté quelques heures en M2). Ce stage m'a donc 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 connaissances sur le modèle BSP, en particulier le modèle de coûts ainsi que la bibliothèque de développement BSPIib (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 de l'environnement PM2 et son interface de communication Madeleine m'a posé beaucoup de difficultés. Je ne connaissais pas du tout cette bibliothèque et je n'avais jamais utilisé l'environnement PM2. Ce qui fait que j'ai dû avancer pas à pas. Au final, l'implémentation fonctionne.
- La compréhension du fonctionnement de BSML et DMML 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'aurais souhaité. Par exemple je n'ai pas eu le temps de finir la conception du deuxième module de communication de la DMMLlib, utilisant la bibliothèque Madeleine. De plus, l'installation de PM2 sur les grappes de PC était plus difficile que prévue, et à cause de problèmes techniques les grappes n'étaient pas disponibles qu'à partir du mois de septembre, ce qui ne nous a pas laisser le temps d'expérimenter nos implémentations et mettre en œuvre des algorithmes usuels pour la DMMLlib.

Au niveau des perspectives de ce travail, il reste à finir l'implémentation du module MPM_Comm et à tester intensivement la bibliothèque avant que la version 0.2 soit distribuée. Ceci devrait être fait d'ici la fin de l'année. Des expériences complémentaires sur les prévisions de performances seront menées, notamment avec des algorithmes et des programmes conçus spécialement à cet effet.

Bibliography

[1] R. Bisseling. Parallel Scientifc 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 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. Scienti¯c 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.
[18] ANSUIEEE, « Standard for Scalable Coherent Interface (SCI) », août 1993, Standard 1596-1992.
[19] AUMAGE O., BOUGÉ L., DENIS A., MÉHAUT J.-F., MERCIER G., NAMYST R., PRYLLIL., « A Portable and Effi cient Communication Library for High-Performance Cluster Computing », in: IEEE Intl Conff on Cluster Computing (Cluster 2000), p. 78-87, Chemnitz, Saxony, Germant', novembre 2000.
[20] AUMAGE O., BOUGÉ L., NAMYST R., « A Portable and Adaptative Multi-Protocol Communication Library for Multithreaded Runtime Systems », in : Proc. 4th Workshop on Runtime Systems forParallel Programming (RTSPP '00), vol. 1800 de Lect. Notes in Camp. Science, p. 1136-1143, Springer-Verlag, Cancun, Mexico, mai 2000.
[21] AUMAGE O., MERCIER G., NAMYST R., « MPICH/Madeleine: a truc multiprotocol MPI for high-performance networks », in: Proc. ISth International Parallel and DistributedProcessingSymposium (IPDPS1001), p. 51, San Francisco, avril 2001.
[22] Compaq, Intel, and Microsoft Corporations, « Virtual Interface Architecture. Version 1.0 », décembre 1997,

Disponible à l' http ://www. viarch. org/.
[23] DENIS A., PÉREZ C., PRIOL T., « Portable parallel CORBA objects : an approach to combine parallel and distributed programming for Grid Computing », in : Euro-Par 2001: Parallel Processing, p. 835-844, Springer-Verlag, Manchester, UK, août 2001.
[24] DENIS A., PÉREZ C., PRIOL T« Towards High Performance CORBA and MPI Middlewares for Grid Computing », in : Proc 2nd Intl. Workshop on Grid Computing, Denver, Colorado, novembre 2001, To appear.
[25] DONGARRA J., HUSS-LEDERMAN S., OTTO S., SNIR M., WALKER D., WI: The Complète Reference, MIT Press, 1996.
[26] DUNNING D., REGNIER G., MCALPINE G., CAMERON D., SHUBERT B., BERRY F., MERRIT A. M., GRONKE E., DODD C., « The Virtual Interface Architecture », IEEE Micro, p. 66-75, mars 1998.
[27] Vincent Danjean, De la réactivité des threads, Thèse de doctorat, spécialité informatique, École normale supérieure de Lyon, 46, allée d'Italie, 69364 Lyon cedex 07,
[28] FOSTER I., KESSELMAN C., TUECKE S., « The Nexus approach t0 integrating multithreading and communication », Journal on Parallel and Distributed Computing, vol. 37, n° 1, p. 70-82, 1996.
[29] Raymond Namyst et Jean-François Méhaut, PM2 : Parallel multithreaded machine. A computing environment for distributed architectures, Parallel Computing (ParCo '95), Elsevier Science Publishers, Septembre 1995, pp. 279–285.
[30] LUSK E., GROPP W., « MPICH Working Note : The Second-Generation ADI for the MPICH Implementation of MPI », Rapport technique, Argonne National Laboratory, 1996.
[31] « The Comment Object Request Broker : Architecture and Specifi cation (Revision 2.3) », décembre 1998, Disponible à l'URL www. opengroup. org/branding/ prodstds/x98ob.htm.
[32]« OmniORB Home Page », Disponible à l'URL www. omniorb. org.
[33] PAKIN S., KARAMCHETI V., CHIEN A., « Fast Messages : Effi cient, Portable Communication for Workstation Clusters and MPPs », IEEE Concurrency, vol. 5, n° 2, p. 60-73, avril 1997.
[34] PRYLLI L., TOURANCHEAU B., « BIP : a new protocol designed for high performance networking on Myrinet », in : I st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98), vol. 1388 de Lect. Notes in Comp. Science, p. 472-485, Springer-Verlag, avril 1998.
[35] Vincent Danjean, LinuxActivations : un support système performant pour les applications de calcul multithreads, Actes des Rencontres francophones du parallélisme (Ren-Par 12) (LIB, Univ. Besançon), 2000, pp. 87–92.
[36] Dolphin Interconnect, « SISCI Documentation and Library », Disponible à l'URL www. dolphinics.no.France, Décembre 2004.

ANNEXE : partie C du code Madeleine

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pm2_common.h>
#include <caml/mlvalues.h>    /* necessaire pour reconnoitre les types Ocaml*/
#include <caml/alloc.h>
#include <caml/memory.h>
#include "bsmlimpl.h"      /*interface des fonctions*/
#include <time.h>
#include <madeleine.h>        /* ce fichier est indispensable dans un programme qui utilise la*/
                              /*  bibliothèque Madeleine*/

#define CHANNEL_NAME = "tcp"       /* on utilise qu'un seul canal tcp pour notre notre*/
                                    /*grappes*/

        /* variables globales nombre de  processeurs,  pid des  processeurs */
        /*. elles sont initialisées au début du programme Par la fonction Initialize */

  int nprocs, pid, argc;


value Madeleine_send_int( value sendbuffer, value destination)
{
        /* les macros CAMLparamn sont indisponsable pour que le GC d'ocaml ne modifie pas
        /*Les parametres utilisés par les fonctions C*/

        CAMLparam2(sendbuffer, destination);

        /*donne une référence a la structure du canal correspondant*/
        /*on a un seul canal tcp dans notre grappe*/

        channel = tbx_htable_get(madeleine->madeleine_htable, CHANNEL_NAME);

        /* au cas où le nœud n'appartient pas au canal, ici inutile puisqu'on en a qu'un seul*/

        If ( !channel ) {
        DISP ("je n'appartiens pas à ce cannal");
        return;


      /* Récupération du numéro global du processus (unique dans la session)*/
      /* on peut utiliser aussi la fonction madeleine->session->process_rank à la place*/

      int my_global_rank = pm2_self();

      /* la function int_val sert à convertir la valeur ocaml destination en un entier C*/

      int remote_global_rank = int_val(destination);

      /*
       * Conversion du numéro global du processus en numéro local au
```

```
 * niveau du canal.
 *
 * Opération en deux temps:
 *
 * 1) on récupère le "process container" du canal qui est un tableau
 * à double entrées rang local/rang global
 *
 * 2) on convertit le numéro global du processus en un numéro local
 * dans le contexte du canal. Le contexte est fourni par le process_ container
 *.
 */

pc = channel->pc;
k = ntbx_pc_global_to_local(pc, my_global_rank);
rk = ntbx_pc_global_to_local(pc, remote_global_rank);

/*
 * On ne s'occupe que des processus de numéro local k ou rk du canal.
 * Les autres n'interviennent pas dans ce programme.
 */

      /*
       * Le processus de numéro local 'k' va jouer le rôle de l'émetteur.
       */

      /*
       * L'objet "connection" pour émettre.
       */
      p_mad_connection_t  out = NULL;

      int length = Int_val(sendbuffer);

      /*
       * On demande l'émission d'un message sur le canal
       * de "k", vers le processus de numéro local "rk".
       */
      out  = mad_begin_packing(channel, rk);



      /*
       * On envoie l'entier en Cheaper.
       */

      mad_pack(out, &length, sizeof(length), mad_send_CHEAPER,  mad_receive_ CHEAPER);

      /*
       * On indique que le message est entièrement construit
       * et que toutes les données non encore envoyées doivent partir.
       *
       */
      mad_end_packing(out);

       /* on ne retourne rien
```

```
  CAMLreturn (Val_unit);

}

/* la function d'envoi de chaine de caractères*/
/* prend en argument un pointeur sur la chaine à envoyer et le numéro du nœud destination */

value Madeleine_send_str(value sendbuffer, value destination)


{
  CAMLparam2(sendbuffer, destination);

        /*donne une réference a la structure du canal correspondant*/
        /*on a un seul canal tcp pour l'instant*/

        channel = tbx_htable_get(madeleine->madeleine_htable, CHANNEL_NAME);

        /* au cas ou le noeud n'appartient pas au canal*/

        if(!channel) {
        DISP ("je n'appartiens pas à ce cannal");
        return;

        /*
       * Récupération du numéro global du processus (unique dans la session).
      */

    int my_global_rank = Int_val(pm2_self);
    int remote_global_rank = int_val(destination);


    /*
     * Conversion du numéro global du processus en numéro local au
     * niveau du canal.
     *
     * Opération en deux temps:
     *
     * 1) on récupère le "process container" du canal qui est un tableau
     * à double entrées rang local/rang global
     *
     * 2) on convertit le numéro global du processus en un numéro local
     * dans le contexte du canal. Le contexte est fourni par le process container.
     */

    pc = channel->pc;
    k = ntbx_pc_global_to_local(pc, my_global_rank);
    rk = ntbx_pc_global_to_local(pc, remote_global_rank);

    /*
     * On ne s'occupe que des processus de numéro local k ou rk du canal.
     * Les autres n'interviennent pas dans ce programme.
     */
```

```
        /*
         * Le processus de numéro local 'k' va jouer le rôle de l'émetteur.
         */

        /*
         * L'objet "connection" pour émettre.
         */
        p_mad_connection_t  out = NULL;

        int length = strlen(String_val(sendbuffer));
        char *string = String_val(sendbuffer);

        /*
         * On demande l'émission d'un message sur le canal
         * de "k", vers le processus de numéro local "rk".
         */

        out  = mad_begin_packing(channel, rk);

        /*
         * On envoie la longueur de la chaîne en Express.
         */

        mad_pack(out, &length, sizeof(length), mad_send_CHEAPER, mad_receive_EXPRESS);

        /*
         * On envoie la chaîne proprement dite, en Cheaper.
         */
        mad_pack(out, string, length, mad_send_CHEAPER, mad_receive_CHEAPER);

        /*
         * On indique que le message est entièrement construit
         * et que toutes les données non encore envoyées
         * doivent partir.
         */
        mad_end_packing(out);

  /* on ne retourne rien*/

  CAMLreturn (Val_unit);

}

/* fonction pour la reception d'un entier */

value Madeleine_receive_int(, value destination)
{
CAMLparam1(destination);

      /* le même code que pour l'envoi*/

        channel = tbx_htable_get(madeleine->madeleine_htable, CHANNEL_NAME);
        if(!channel) {
```

```
  DISP ("je n'appartiens pas à ce cannal");
  return;
  int my_global_rank = Int_val(pm2_self);
  int remote_global_rank = int_val(destination);
pc = channel->pc;
k = ntbx_pc_global_to_local(pc, my_global_rank);
rk = ntbx_pc_global_to_local(pc, remote_global_rank);
p_mad_connection_t  in  = NULL;

   int received_int =    0;

   /*
    * On demande la réception du premier "message"
    * disponible. Note: il n'est pas possible de
    * spécifier "de qui" on veut recevoir un message.
    */
   in = mad_begin_unpacking(channel);

   /*
    * On reçoit l'entier en cheaper pour optimiser
    */
   mad_unpack(in, &received_int, sizeof(received_int), mad_send_CHEAPER, mad_receive_ CHEAPER);


   /*
    * On indique la fin de la réception. C'est seulement
    * après l'appel à end_unpacking que la disponibilité
    * des données marquées "receive_CHEAPER" sont
    * garanties.
    */
   mad_end_unpacking(in);

     /* on retourne l'entier reçu */

  CAMLreturn (Val_int(received_int));
}

/* la function de reception d'une chaine de caractères */

value Madeleine_receive_str( value destination)


{
    CAMLparam1(destination);

   /*même début de code que les fonctions précédentes */

   channel = tbx_htable_get(madeleine->madeleine_htable, CHANNEL_NAME);

   if(!channel) {
   DISP ("je n'appartiens pas à ce cannal");
   return;

   int my_global_rank = Int_val(pm2_self);
```

```
    int remote_global_rank = int_val(destination);


pc = channel->pc;
k = ntbx_pc_global_to_local(pc, my_global_rank);
rk = ntbx_pc_global_to_local(pc, remote_global_rank);


    p_mad_connection_t  in  = NULL;

    int receivedlength =  0;
    char *msg = NULL;

    /*
     * On demande la réception du premier "message"
     * disponible. Note: il n'est pas possible de
     * spécifier "de qui" on veut recevoir un message.
     */
    in = mad_begin_unpacking(channel);

    /*
     * On reçoit la longueur de la chaîne en Express car
     * on a immédiatement besoin de cette information pour
     * allouer la mémoire nécessaire à la réception de la
     * chaîne proprement dite.
     */
    mad_unpack(in,&receivedlength,sizeof(receivedlength),mad_send_CHEAPER, mad_receive_EXPRESS);

    /*
     * On alloue de la mémoire pour recevoir la chaîne.
     */
    msg = TBX_MALLOC(receivedlength);

    /*
     * On reçoit la chaîne de caractères en Cheaper.
     */
    mad_unpack(in, msg,  receivedlength, mad_send_CHEAPER,mad_receive_CHEAPER);


    /*
     * On indique la fin de la réception. C'est seulement
     * après l'appel à end_unpacking que la disponibilité
     * des données marquées "receive_CHEAPER" sont
     * garanties.
     */
    mad_end_unpacking(in);
    /*
     * On libère la mémoire allouée pour recevoir la chaîne.
     */
    TBX_FREE(msg);
}

/*
 * On termine l'exécution des bibliothèques.
```

```
 */
common_exit(NULL);

/* on retourne la chaine reçue après l'avoir convertie en chaine ocaml*/

CAMLreturn (make_str(msg));

}
```

Fig. 4.3-code Madeleine