MSPML : Environnements de communication et tolérance aux pannes

Abdeltouab Belbekkouche

Stage de Master

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

Avril-Septembre 2005

Remerciements

Je remercie Frédéric LOULERGUE de m'avoir accepté parmi son équipe de recherche pour effectuer ce stage. Je tiens à le remercier pour ses déplacements pour diriger nos travaux.

Je remercie les membres du Laboratoire LIFO et notamment l'équipe d'enseignement du Master de recherche de nous avoir accueillir durant cette année universitaire.

Je remercie Radia pour tout ce qu'elle me donne depuis longtemps...

Je remercie mes parents, mon frère Samir et mes oncles Rabeh et Youssef pour tout...

Résumé

Certains problèmes nécessitent des performances que seules les machines massivement parallèles peuvent offrir. Leur programmation demeure néanmoins difficile. Les travaux étudiant le mélange de la programmation fonctionnelle et du parallélisme se repartissent en deux catégories : les extensions explicitement parallèles des langages fonctionnels – mais les langages obtenus sont soit indéterministes soit brisent l'aspect fonctionnel pur – et les implantations parallèles avec sémantique fonctionnelle - mais les langages obtenus n'expriment pas directement les algorithmes parallèles et ne permettent pas la prévision des temps d'exécution. L'approche des langages à patrons, dans lesquels seulement un ensemble d'opérations sont exécutées en parallèles, est intermédiaire. Leur sémantique fonctionnelle est explicite mais leur sémantique opérationnelle parallèle est implicite. L'ensemble de patrons doit être le plus complet possible mais cet ensemble s'avère dépendant du domaine d'application.

Dans une démarche qui approfondit cette position intermédiaire on trouve l'objectif de parvenir a des langages universels dans lesquels le code source permet de déterminer le coût. Cette dernière exigence nécessite que soient explicites dans les programmes les lieux du réseau de processeurs de la machine. Dans ce cadre, la bibliothèque MSPML pour Minimally Synchronous Parallel ML est née. Elle possède de plus une sémantique d'évaluation asynchrone, une propriété qui s'avère très utile pour des programmes parallèles déséquilibrés.

Les environnements de communication sont à la base du mécanisme de communication pour MSPML. Dans une première partie, on discute l'implantation d'un nouveau mécanisme de gestion de ces environnements. Un mécanisme qui résoudra les problèmes liés à la profondeur d'asynchronisme notamment pour des programmes localement déséquilibrés.

Dans une second partie, on étudie les possibilités de doter MSPML d'un mécanisme de tolérance aux pannes. Un aspect qui est aujourd'hui indispensable pour les systèmes parallèles et distribués. Par cette démarche on cherche à donner plus de fiabilité et de disponibilité à MSPML.

Mots clés : Conception de langages de programmation parallèles ; environnements de communication ; tolérance aux pannes ; bibliothèque portable de primitives parallèles pour Objective CAML.

Abstract

Some problems require performance that only massively parallel computers offer, but their programming is still difficult. Works on functional programming and parallelism can be divided in two categories : explicit parallel extensions of functional languages – where resulting languages are either non-deterministic or non functional – and parallel implementations with functional semantics where resulting languages don't express parallel algorithms directly and don't allow the prediction of execution times. Algorithmic skeletons languages, in which only a finite set of operations (the skeletons) are run in parallel, constitutes an intermediate approach. Their functional semantics is explicit but their parallel operational semantics is implicit. The set of algorithmic skeletons has to be as complete as possible but it is often dependent on the domain of application.

In a step which looks further into this intermediate position we find the objective to have universal languages in which the source code makes it possible to determine the cost. This last requirement requires that in the programs the places of the network of processors of the machine are explicit. Within this framework, MSPML library for Minimally Synchronous Parallel ML was born. It offers moreover an asynchronous semantics of evaluation, a property which proves very useful for unbalanced parallel programs.

Communication environnements are at the base of the mechanism of communication for MSPML. In a first part, we discuss the implementation of a new mechanism of management of these environments, a mechanism which will solve the problems related to the asynchrony depth in particular for programs locally unbalanced.

In a second part, we study the possibilities of equipping MSPML with a mechanism of fault-tolerance. An aspect which is essential today for the parallel and distributed systems. By this step we seek to give more reliability and availability to MSPML.

Keywords : Design of parallel programming languages ; communication environments ; fault-tolerance ; portable library of parallel primitives for Objective CAML.

Table des matières

Chapitre 1

Introduction

1.1 Présentation générale

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 [30] 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. De nombreux travaux ont étudié la question du mélange de la programmation fonctionnelle et du parallélisme. On peut les classer en deux catégories :

1. les extensions explicitement parallèles des langages fonctionnels,
2. les implantations parallèles avec sémantique fonctionnelle.

Dans la première catégorie, Concurrent ML [23] par exemple ajoute à ML une notion de canal et a une sémantique concurrente avec les défauts qui l'accompagnent. Cette extension brise l'aspect fonctionnel pur du sous-langage fonctionnel de ML. Dans la seconde catégorie on note de nombreux travaux sur la réduction de graphes en parallèle, on trouve un exemple dans [3]. Ces systèmes n'expriment pas directement les algorithmes parallèles et ne permettent pas la prévision des temps d'exécution car les processus physiques y sont implicites et la stratégie d'évaluation parallèle n'est pas décrite par la sémantique.

Une approche intermédiaire est celle des langages à patrons ou *algorithmique skeletons* dans lesquels seulement un ensemble fixé d'opérations (les patrons) sont exécutés en parallèle. Leur sémantique fonctionnelle est explicite mais leur sémantique opérationnelle parallèle est implicite. Du point de vue du programmeur, le style de programmation associé revient à l'utilisation de combinateurs dont l'effet sur la parallélisation est défini extérieurement. Les avantages par rapport aux extensions concurrentes sont bien sûr que le déterminisme et le non-blocage sont garantis et que

l'on conserve une sémantique fonctionnelle pure. Cependant, l'implanteur de bibliothèques de patrons doit faire face à deux problèmes. D'une part, il doit proposer un ensemble le plus complet possible de patrons et cet ensemble s'avère dépendant du domaine d'application. D'autre part, il lui faut implanter efficacement pour chaque architecture chaque patron.

Le projet BSλ /BSML approfondit la position intermédiaire que le paradigme des patrons occupe avec deux objectifs : 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.

Le BSλ-calcul [19, 14] est un λ- calcul étendu par des opérations parallèles BSP qui s'avère confluent et universel pour les algorithmes BSP. La BSMLlib [10, 15] est une implantation partielle de ces opérations sous forme d'une bibliothèque pour le langage Objective CAML. Cette bibliothèque permet d'écrire des programmes parallèles BSP sur une grande variété d'architectures allant du PC à deux processeurs au systèmes massivement parallèles comprenant plusieurs centaines de processeurs, en passant par des clusters de PC.

Le mécanisme de synchronisation globale imposé par le modèle BSP soulève quelques problèmes tels que le surcoût de la barrière de synchronisation, ou par exemple la synchronisation à une étape donnée d'un processeur avec les autres alors qu'il n'est engagé dans aucune communication. De plus, dans la sémantique actuelle de BSML, l'opération de composition parallèle, telle qu'elle est définie, nécessite que deux expressions composées parallèlement s'évaluent en utilisant le même nombre de barrières de synchronisation. Cette nécessité disparaît dans une sémantique d'évaluation asynchrone. Ainsi, Minimally Synchronous Parallel ML (MSPML) est née avec une sémantique et un modèle le coût *Message Passing Machine* (MPM) qui est simple et réaliste [18].

Les environnements de communication sont à la base du mécanisme de communication pour MSPML. La gestion évoluée de ces environnements proposée dans [17] n'a pas été proprement implantée dans la distribution. Il s'agit de le faire dans une première partie.

Dans une second partie, il faudra étudier les mécanismes de tolérance aux pannes pour concevoir et implanter un tel mécanisme pour MSPML.

1.2 Organisation du manuscrit

Après ce premier chapitre dans lequel on a présenté le contexte général de ce travail et ces objectifs, la suite de ce manuscrit sera organisée comme suit.

Au chapitre 2, on présente MSPML de façon informelle en donnant son modèle de coût, sa conception et son implantation. Ce chapitre a été écrit en collaboration avec Radia Benheddi qui effectue son stage d'initiation à la recherche sur le sujet de la composition parallèle dans MSPML.

Au chapitre 3, on présente le mécanisme de gestion des environnements de communication actuel dans MSPML, le nouveau mécanisme proposé et notre façon de l'implanter.

Au chapitre 4, on donne un état de l'art des protocoles de tolérance aux pannes pour les systèmes parallèles et distribués en montrant les avantages et les inconvénients par rapport à MSPML. On présente ensuite le mécanise adopté et son implantation pour MSPML.

Enfin, au chapitre 5, on donne notre conclusion des objectifs réalisés et les perspective à atteindre pour des travaux futurs.

Chapitre 2

Minimally Synchronous Parallel ML

Le modèle de programmation Bulk Synchronous Parallel (BSP) a été introduit par Valiant [30] pour offrir une haut niveau d'abstraction. Il a été utilisé par une large variété d'applications : calcul scientifique, algorithmes génétiques, réseaux de neurones, bases de données parallèles, etc. Ce succès est dû aux avantages qu'offre ce modèle :

- absence du blocage et l'indéterminisme peut être soit évité, soit limité à des cas très particuliers ;
- la portabilité et la prévision des performances.

Ainsi, on trouve aujourd'hui une nouvelle orientation de la recherche dans le domaine des algorithmes parallèles dont la plupart conçus récemment suivent le modèle BSP ou le modèle CGM (Coarse-Grained Multicomputer) qui peut être vu comme étant un cas particulier du modèle BSP [18]. Dans cette orientation, on trouve Bulk Synchronous Parallel ML (BSML) qui est un langage fonctionnel pour la programmation parallèle BSP [16, 10].

Cependant, BSP présente deux inconvénients :

- d'abord la barrière de synchronisation qui peut être coûteuse ;
- et, les restrictions imposées par ce modèle (on ne peut écrire tout les algorithmes sous BSP).

Par conséquent, un nouveau langage parallèle fonctionnel, sans barrières de synchronisation, est proposé dans [18]. Il est appelé Minimally Synchronous Parallel ML (MSPML). MSPML possède le même langage source et la même sémantique haut niveau que BSML mais avec une sémantique bas niveau (plus efficace pour les programmes déséquilibrés) et une implantation différente. Il n'existe pas à présent une implantation complète du langage MSPML mais plutôt une implantation comme étant une bibliothèque pour le langage de programmation fonctionnel Objective Caml. Dans ce chapitre, on présente d'abord le modèle de coûts et d'exécution pour MSPML, puis on présente son noyau et son architecture. Enfin, on donne une comparaison entre MSPML et BSML.

2.1 Modèle de coûts et d'exécution

Plutôt que de passer directement à la conception d'un modèle et d'un langage à deux niveaux pour l'utilisation de grappes de machines parallèles, les créateurs de MSPML on d'abord considéré la conception d'un langage proche de BSML mais sans les barrières de synchronisation.

Il est souvent admis que les barrières de synchronisation ne sont pas un handicap pour les performances (dans le cas *d'une seule* machine parallèle), notamment parce qu'une vue globale du calcul permet des optimisations qui ne sont pas possibles dans le cas d'un parallélisme moins structuré (voir par exemple [12]) ou que l'éventualité de performances un peu moins bonnes n'est pas un désavantage suffisant par rapport à la plus grande facilité de conception et de correction/vérification des algorithmes et programmes.

Toutefois il y a de nombreux programmes parallèles implémentés, en particulier en MPI, qui ne suivent pas le modèle BSP mais pour lesquels on souhaite pouvoir raisonner sur le coût. C'est ce qui a conduit au modèle *BSP without barrier* [27] (BSPWB) puis au modèle MPM [26, 4].

BSPWB est un modèle directement inspiré du modèle BSP. Il propose de remplacer la notion de super-étape par la notion de m-étape définie comme suit. À chaque m-étape, chaque processeur effectue une phase de calcul suivie par une phase de communication. Durant la phase de communication, les processeurs échangent les données dont ils ont besoin pour la m-étape suivante.

La machine parallèle est caractérisée par les trois paramètres suivants (les deux derniers sont exprimés comme multiples de la puissance de calcul des processeurs) :

- le nombre de processeurs p,
- la latence L du réseau,
- le temps g pour échanger un mot entre deux processeurs.

Le temps nécessaire à un processeur i pour exécuter une m-étape s est $t_{s,i}$ borné par T_s le temps nécessaire à l'exécution de la m-étape s par la machine parallèle.

T_s est défini inductivement par :

$$\begin{cases} T_1 = max\{w_{1,i}\} + \max\{g \times h_{1,i} + L\} \\ T_s = T_{s-1} + max\{w_{s,i}\} + max\{g \times h_{s,i} + L\} \end{cases}$$

où $i \in \{0, \ldots, p-1\}$ et $s \in \{2, \ldots, R\}$ où R est le nombre de m-étapes du programme et $w_{s,i}$ et $h_{s,i}$ sont respectivement le temps de calcul local au processeur i durant la m-étape s et $h_{s,i} = max\{h^+_{s,i}, h^-_{s,i}\}$ où $h^+_{s,i}$ (resp. $h^-_{s,i}$) est le nombre de mots reçus (resp. envoyés) par le processeur i durant la m-étape s.

Dans ce modèle il y a toutefois une barrière implicite à chaque étape, le coût de la barrière elle-même étant nul. De ce fait ce modèle est une approximation trop grossière. Une meilleur borne $\Phi_{s,i}$ est donnée par le modèle *Message Passing Machine* [26]. Les paramètres de ce modèle sont identiques à ceux du modèle BSPWB.

On utilise l'ensemble $\Omega_{s,i}$ pour un processeur i et une m-étape s (Figure 2.1)

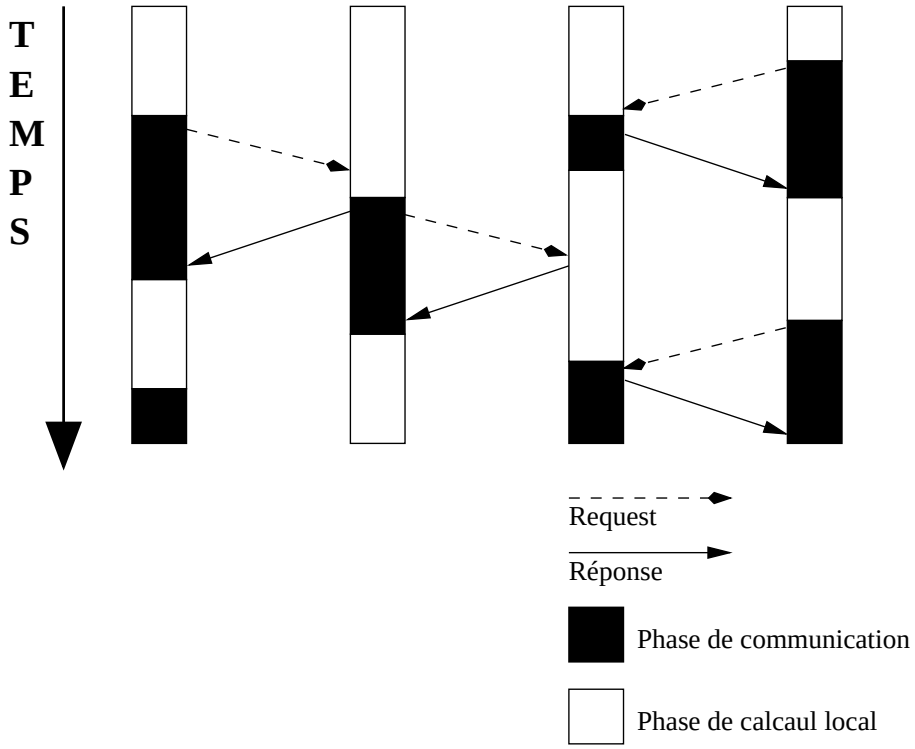

FIG. 2.1 – Le modèle MPM.

défini par :

$$\Omega_{s,i} = \{j|\text{processeur } j \text{ envoie un message au processeur } i \text{ à la m-étape } s\} \bigcup \{i\}$$

Les processeurs de l'ensemble $\Omega_{s,i}$ sont appelés "partenaires entrants" du processeur i à la m-étape s. La borne $\Phi_{s,i}$ est définie inductivement par :

$$\begin{cases} \Phi_{1,i} = \max\{w_{1,j}|j \in \Omega_{1,i}\} + (g \times h_{1,i} + L) \\ \Phi_{s,i} = \max\{\Phi_{s-1,j} + w_{s-1,j}|j \in \Omega_{s,i}\} + (g \times h_{s,i} + L) \end{cases}$$

où $h_{s,i} = \max\{h^+_{s,i}, h^-_{s,i}\}$ pour $i \in \{0, \ldots, p-1\}$ et $s \in \{2, \ldots, R\}$.

Le temps d'exécution pour un programme est donc borné par :

$$\Psi = \max\{\Phi_{R,j}|j \in \{0, 1, \ldots, p-1\}\}$$

Le modèle MPM prend en compte le fait qu'un processeur ne se synchronise qu'avec chacun de ses partenaires entrants et est donc plus précis que BSPWB. Les expériences menées montrent que ce modèle s'applique bien à MSPML [20].

2.2 Le noyau de la bibliothèque

La bibliothèque MSPML est basée sur les primitives donnés dans la figure 2.2. Elles donnent l'accès aux paramètres du modèle Message Passing Machine (MPM)

utilisé. En particulier, la fonction **p()** retourne le nombre statique de processeurs de la machine parallèle. Cette valeur ne change pas durant l'exécution. Il y a aussi un type abstrait polymorphe α **par** qui représente le type des vecteurs parallèles de longueur p d'objets de type α, un seul objet par processeur. La non-imbrication des types **par** est assurée par le système de typage [9].

p	:	$unit- > int$
g	:	$unit- > float$
l	:	$unit- > float$
mkpar	:	$(int- >' a)- >' a\ par$
apply	:	$('a- >' b)\ par- >' a\ par- >' b\ par$
get	:	$'apar- > int\ par- >' a\ par$
mget	:	$(int- >' a)\ par- > (int- > bool)\ par- > (int- >' aoption)\ par$
at	:	$'apar- > int- >' a$
juxta	:	$int- > (unit- >' a\ par)- > (unit- >' a\ par)- >' a\ par$

FIG. 2.2 – Le noyau de la bibliothèque MSPML.

Les constructeurs parallèles opèrent sur les vecteurs parallèles. Ces vecteurs parallèles sont crées par la primitive **mkpar**, ainsi, (**mkpar** f) stock (fi) dans le processeur i pour i entre 0 et $(p-1)$. On écrit habituellement **fun** $pid\ \rightarrow\ e$ pour f afin de montrer que l'expression e peut être différente sur chaque processeur. Cette expression est appelée locale : elle est à l'intérieur de la fonction **mkpar** et sa valeur dépend du processeur local sur lequel elle se trouve. L'expression (**mkpar** f) est un objet parallèle global. Par exemple l'expression **mkpar**(**fun** $pid\ \rightarrow\ pid$) sera évaluée en un vecteur parallèle $\langle\ 0\ ,\ldots,\ p-1\ \rangle$.

Dans le modèle MPM, un algorithme est écrit comme étant une combinaison entre des calculs locaux asynchrones et des phases de communication. Les phases asynchrones sont programmée avec **mkpar** et **apply**. Par exemple, l'expression **apply** (**mkpar** f) (**mkpar** e) stock (fi) (ei) dans le processeur i.

Les phases de communication sont réalisées par **get** et **mget**. La sémantique du **get** est donnée par :

$$\textbf{get}\langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle\langle\ i_0\ ,\ldots,\ i_{p-1}\ \rangle = \langle\ v_{i0\%p}\ ,\ldots,\ v_{i0\%p-1}\ \rangle$$

où % est le modulo.

La fonction **mget** est une généralisation qui permet d'avoir les données à partir de différents processeurs durant la même m-étape et de délivrer différents messages à de différents processeurs.

Dans le type de **mget** dans la figure 2.2, α option est définie comme suit :

$$\textbf{type}\ \alpha\ \text{option} = \text{None} \mid \text{Some}\ \textbf{of}\ \alpha.$$

La sémantique de la fonction **mget** est :

$$\textbf{mget}\langle\ f_0\ ,\ldots,\ f_{p-1}\ \rangle\langle\ b_0\ ,\ldots,\ b_{p-1}\ \rangle = \langle\ g_0\ ,\ldots,\ g_{p-1}\ \rangle$$

où $g_i =$ **fun** $j \ \rightarrow$ **if** $b_i \ j$ **then** Some $(f_j \ i)$ **else** None

Le langage complet contient aussi une conditionnelle synchrone :

$$\textbf{if}\, e\, \textbf{at}\, n\, \textbf{then}\, e_1\, \textbf{else}\, e_2$$

Selon la valeur du vecteur parallèle au processeur donné par la valeur n l'expression est évaluée en v_1 (la valeur obtenue par l'évaluation de e_1) ou en v_2 (la valeur obtenue par l'évaluation de e_2). Mais, Objective Caml est un langage strict, ainsi, cette opération conditionnelle synchrone ne peut être définie comme une fonction. Pour cela, la bibliothèque MSPML contient la fonction **at** pour être utilisée dans les constructions suivantes :

- **if at** $e \ n$ **then** ... **else** ...
- **match**(**at** $e \ n$) **with**...

at exprime la phase de communication. La conditionnelle globale est nécessaire pour exprimer des algorithmes tels que :

Repeat
Itération Parallèle
Until Max of local errors $< \epsilon$

Sans le **at**, le contrôle global ne peut prendre en compte les données calculées localement.

2.3 Exemples

On présente maintenant quelques exemples qui font partie de la bibliothèque MSPML.

2.3.1 Les fonctions les plus utilisées

Quelques fonctions usuelles peuvent être définies en utilisant que les primitives. Par exemple la fonction **replicate** crée des vecteurs parallèles qui contiennent la même valeur partout. La primitive **apply** peut être utilisée rien que pour les vecteurs parallèles de fonctions qui prennent un seul argument. Pour les fonctions à deux arguments on a besoin de définir la fonction **apply2**.

let $replicate \ x =$ **mkpar** (**fun** $pid \ \rightarrow \ x$)
let $apply2 \ f \ v1 \ v2 =$ **apply** (**apply** $f \ v1$) $v2$

Il est aussi très commode d'appliquer la même fonction séquentielle à chaque processeur. Cela peut être fait en utilisant les fonctions **parfun** : elles différent seulement dans le nombre d'arguments :

let $parfun \ f \ v =$ **apply** $(replicate \ f) \ v$
let $parfun2 \ f \ v1 \ v2 =$ **apply** $(parfun \ f \ v1) \ v2$

let $parfun3\ f\ v1\ v2\ v3 =$ **apply** $(parfun2\ f\ v1\ v2)\ v3$

(applyat $n\ f_1\ f_2\ v$) applique la fonction f_1 au processeur n et la fonction f_2 aux autres processeurs :
let *applyat* $n\ f1\ f2\ v =$ **apply** (**mkpar** (**fun** $i\ \rightarrow$ (**if** $i = n$ **then** $f1$ **else** $f2$))) v
parpair_of_pairpar transforme un vecteur parallèle de paires en une paire de vecteurs parallèles :

let $fst\ (x, y) = x$ **and** $snd(x, y) = y$
let $parpair\_of\_pairpar\ vv = (parfun\ fst\ vv,\ parfun\ snd\ vv)$

2.3.2 Les fonctions de communication

La sémantique de la fonction d'échange total est donnée par :

$totex\langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle = \langle\ f\ ,\ldots,\ f\ ,\ldots,\ f\ \rangle$
où $\forall i.(0 \leq i < p-1) \Rightarrow\ (f\ i) = v_i$. Le code est comme présenté ci-dessous où, **noSome** enlève le constructeur **Some** et **compose** est la fonction de composition :

(∗ val totex: α par → (int → α) par ∗)
let totex vv = (parfun compose noSome)(mget (parfun(**fun** v i → v)vv)(replicate(**fun** i → **true**)))

Son coût parallèle est $(p-1) \times s \times g + L$, où s dénote la taille en mots de la plus grande valeur v qui se trouve sur un certain processeur n. À partir de la fonction d'échange total, on peut obtenir une version qui retourne un vecteur parallèle de listes :

(∗ val totex_list: α par → α list par ∗)
let totex_list v = (parfun2 List.map(totex v)(replicate(procs()))

où :

$$\begin{cases} \textit{(∗val List.map: (} \alpha\ \rightarrow\ \beta\ \textit{)} \rightarrow\ \alpha\ \textit{list} \rightarrow\ \beta\ \textit{list ∗)} \\ \text{List.map } f\ [v_0;\ldots;v_n]\ =\ [(f\ v_0);\ldots;(f\ v_n)] \\ \text{procs()}\ =\ [0;\ldots;\mathbf{p}()\text{-}1]. \end{cases}$$

La sémantique de la diffusion est :

$$\text{bcast}\ \langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle\ r\ \ =\ \ \langle\ v_{r\%p}\ ,\ldots,\ v_{r\%p}\ \rangle$$

La fonction `broadcast_direct` qui réalise la diffusion peut être écrite comme suit :

(∗ bcast_direct: int → α par → α par ∗)
let bcast_direct root vv = **get** vv (replicate root)

Son coût parallèle est $(p-1) \times s \times g\ +\ L$, où s dénote la taille en mots de la valeur v_n qui se trouve sur le processeur n.

La bibliothèque standard de MSPML contient une collection de ces fonctions qui facilitent l'écriture de programmes. Ainsi, il est exactement similaire d'écrire des programmes MSPML ou d'écrire des programmes en utilisant des patrons data-parallèles, mais, avec MSPML il est possible d'écrire ses propres patrons comme étant des fonctions de haut niveau si la bibliothèque standard ne fournit pas les fonctions nécessaires.

Quelques fonctions de la bibliothèque standard sont récursives. Par exemple, il existe une fonction *broadcast* qui est évaluée en $\log p$ m-étapes au lieu d'une m-étape :

```
let bcast_logp root vv =
  let from n =
    mkpar(fun i → let j=natmod (i+(p())−root) (p()) in
      if (n/2<=j)&&(j<n) then i−(n/2) else i) in
  let rec aux n vv =
    if n<1 then vv else get (aux (n/2) vv) (from n)
  in aux (p()) vv
```

2.4 Comparaison avec BSML

Bulk Synchronous Parallel ML (BSML) a les mêmes opérations que MSPML, Mais avec une seule différence : les communications (suivies immédiatement par une barrière de synchronisation) sont réalisées par la primitive **put**. Son type est le suivant :

$$\textbf{put}:(\text{int} \rightarrow \alpha \text{ option}) \textbf{ par} \rightarrow (\text{int} \rightarrow \alpha \text{ option}) \textbf{ par}$$

Considérons l'expression suivante :

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

Pour envoyer la valeur v du processeur j au processeur i, la fonction fs_j au processeur j doit être comme étant (fs_j i) évalué en (Some v). Pour n'envoyer aucune valeur à partir du processeur j au processeur i, (fs_j i) doit être évalué en None.

L'expression (2.1) s'évalue en un vecteur parallèle contenant une fonction fd_i des messages délivrés au niveau de chaque processeur. au processeur i, (fd_i j) s'évalue en None si le processeur j n'envoie aucun message au processeur i ou s'évalue en (Some v) si le processeur j envoie la valeur v au processeur i.

Cette primitive peut être utilisée pour programmer les fonctions **get** et mget.Mais ces deux fonctions sont moins efficaces que les primitives de MSPML : ils nécessitent deux super-étapes BSP, par conséquent deux barrières de synchronisation.

Donc pour le moment, la principale différence entre la programmation BSML et la programmation MSPML est que les communications sont faites avec le style get dans MSPML et le style put dans BSML.

Par exemple, le broadcast cité ci-dessus peut être écrit en BSML :

```
let bcast_direct root vv =
  let mkmsg = mkpar(fun i v dst → if i=root
  then Some v else None) in parfun noSome
    (apply (put (apply mkmsg vv)) (replicate root))
```

Les programmes BSML et MSPML qui utilisent rien que la fonction bcast_direct pour la communication doivent être identiques, mais leurs coûts vont être différents. Par conséquent, il est facile de réécrire les programmes BSML pour obtenir les programmes MSPML. La seule difficulté est quand le programme BSML utilise directement la primitive **put**, à partir des fonctions de la bibliothèque standard : quelques réécritures compliquées sont nécessaires.

Chapitre 3

Nouvelle gestion des environnements de communication

À cause de la nature asynchrone de MSPML, la sauvegarde, dans les environnements de communication, des valeurs qui peuvent être demandées par d'autres processus, dans le futur, s'avère indispensable. Pour les raisons d'implantation, la taille des environnement de communication doit être limitée. Cela rend le vidage de ces environnements nécessaire lorsqu'il sont pleins. Lorsque l'environnement de communication d'un processus est plein, celui-ci doit attendre que tous les environnements de communication des autres processus soient également pleins. Une synchronisation globale apparaît et les environnements de communication sont alors vidés. L'avantage de cette méthode est sa simplicité. Néanmoins elle peut être inefficace en terme d'espace mémoire et en terme de temps d'exécution. Dans ce chapitre on présente un nouveau mécanisme de gestion des environnements de communication pour remédier à ces problèmes.

3.1 Le mécanisme

Pour expliquer comment les environnements de communication sont vidés, la sémantique de bas niveau de MSPML doit être présentée. Durant l'exécution d'un programme MSPML, pour chaque processus i, le système possède une variable $mstep_i$ contenant le nombre du m-étape actuel. À chaque fois une expression **(get vv vi)**, est évaluée à un processus i donné :

1. $mstep_i$ est incrémentée par un.
2. La valeur que tient ce processus dans le vecteur parallèle vv est sauvegardée avec la valeur de $mstep_i$ dans l'environnement de communication. Un environnement de communication peut être vu comme une liste d'association qui relie les nombres de m-étapes avec les valeurs.
3. La valeur j que tient ce processus dans le vecteur parallèle vi est le numéro de processus duquel le processus i veut recevoir une valeur. Ainsi, le processus i

envoie une requête au processus j : il demande la valeur à la m-étape $mstep_i$. Quand le processus j reçoit la requête (des *threads* sont dédiés au traitement de telles requêtes, donc, le travail du processus j n'est pas interrompu), deux cas se présentent :

- $mstep_j \geq mstep_i$: cela veut dire que le processus j a déjà atteint la même m-étape que le processus i. Ainsi, le processus j accède dans son environnement de communication à la valeur associée à la m-étape $mstep_i$ et l'envoie au processus i.
- $mstep_j < mstep_i$: rien ne peut se faire jusqu'à ce que le processus j atteigne la même m-étape que le processus i.

Si $i = j$, l'étape 3 n'est pas exécutée.

Dans l'implantation réelle de MSPML, la taille de l'environnement de communication est évidemment limitée. Il est nécessaire au moment de l'exécution du programme de fournir un paramètre appelé la profondeur d'asynchronisme (*asynchrony depth*). Cette valeur appelée **mstepmax**, est la taille de l'environnement de communication en terme du nombre de valeurs qu'il peut sauvegarder (nombre de m-étapes). L'implantation de l'environnement de communication est un vecteur **com_env** de taille **mstepmax**, chaque élément de ce vecteur est une sorte de pointeur vers la valeur linéarisée sauvegardée à la m-étape dont le nombre est l'index du vecteur. Un problème peut apparaître quand les environnements de communication sont pleins. Quand le vecteur à un processus est rempli alors ce processus doit attendre puisqu'il ne peut procéder la m-étape suivante. Quand tous les environnements de communication sont pleins, une synchronisation globale apparaît et les vecteurs sont vidés. La variable **mstep** de chaque processus est aussi remise à sa valeur initiale.

L'avantage de cette méthode est sa simplicité. Néanmoins elle peut être inefficace en terme d'espace mémoire et de temps d'exécution puisque une synchronisation globale est nécessaire (le coût d'une barrière de synchronisation est $g \times (p-1) + L$, (voir la section 2.1). Ainsi un autre mécanisme est proposé [17]. Nous avons précisé et implanté ce mécanisme.

Pour éviter un gaspillage de l'espace mémoire, chaque processus doit libérer les valeurs inutiles dans son environnement de communication. Ces valeurs sont celles dont la m-étape associée (l'indice du vecteur) est plus petite que la m-étape courante de chaque processus, autrement dit, plus petite que la plus petite m-étape.

Évidement, cette information ne peut être mise à jour à chaque m-étape sans l'exécution d'une synchronisation globale, mais, elle peut l'être lors d'une communication entre deux processus. Ces processus peuvent échanger leurs connaissances sur les états courants des autres processus. Pour cela, chaque processus possède outre son vecteur **com_env** de taille **mstepmax** et son compteur de m-étapes **mstep**, une valeur **mstepmin** contenant la plus petite valeur de **mstep** connue.

L'environnement de communication devient maintenant une sorte de file. Si $mstep - mstepmin \leq mstepmax$, alors, il reste encore de l'espace pour ajouter une valeur dans l'environnement de communication à l'indice $mstep\%mstepmax$. Le problème revient à mettre à jour la variable **mstepmin**.

Ceci peut être effectué en utilisant un vecteur **msteps** de taille p, contenant les valeurs des m-étapes les plus récemment connues. La valeur du vecteur **msteps** à l'indice i au processus i est la valeur de sa variable **mstep**.

Sans changer les données échangées pour effectuer un **get**, à chaque fois qu'un processus i demande une valeur du processus j, il lui envoie sa valeur **mstep**. Cette valeur est mise dans le vecteur **msteps** du processus j à l'indice i. Quand le processus j répond, i sait que celui-ci a au moins atteint la même m-étape que lui-même et peut, donc mettre à jour son vecteur **msteps** à l'indice j.

La nouvelle valeur de **mstepmin**, qui est le minimum des valeurs contenues dans le vecteur **msteps**, peut être calculée seulement lorsqu'il est nécessaire, c'est-à-dire quand le vecteur **com_env** est plein ou bien peut être calculée à chaque fois qu'une valeur est changée dans le vecteur **msteps**. Dans le premier cas, il peut y avoir un gaspillage de l'espace mémoire, mais dans le dernier cas, il y a un petit surcoût en temps. Des tests ultérieure sur la grappe de LIFO permettront de comparer l'efficacité respective des deux méthodes. Cette comparaison n'aura toutefois rien d'absolu. En effet l'avantage de l'une ou l'autre dépendra de l'application. Nous prévoyons ainsi, dans une version ultérieure, de permettre à l'utilisateur de pouvoir choisir, au moment de l'exécution, quelle méthode appliquer.

3.2 Exemple

Comme exemple, on peut avoir un programme MSPML. dans lequel on évalue **(bcast 0 vec)** sur une machine à 3 processeurs. Au début, chaque processeur possède le vecteur **msteps** suivant :

-1	-1	-1

Après le premier **get**, les vecteurs **msteps** sont :

Processeur :	0			1			2		
msteps :	0	0	0	0	0	-1	0	-1	0

Ainsi, la première cellule du vecteur **com_env** au processeur 0 peut être libérée.

3.3 Le problème des processus isolés

Malheureusement, ce mécanisme a un inconvénient : si un processus ne communique jamais avec le reste de la machine parallèle, sa valeur **mstepmin** ne sera jamais mise à jour et ce processus sera bloqué dès que son environnement de communication deviendra plein. La solution pour éviter le blocage est qu'à chaque fois qu'un processus est bloqué à cause du remplissage de son environnement de communication, alors, il va demander, après un délai, la valeur **mstep** d'un ou de plusieurs autres processus.

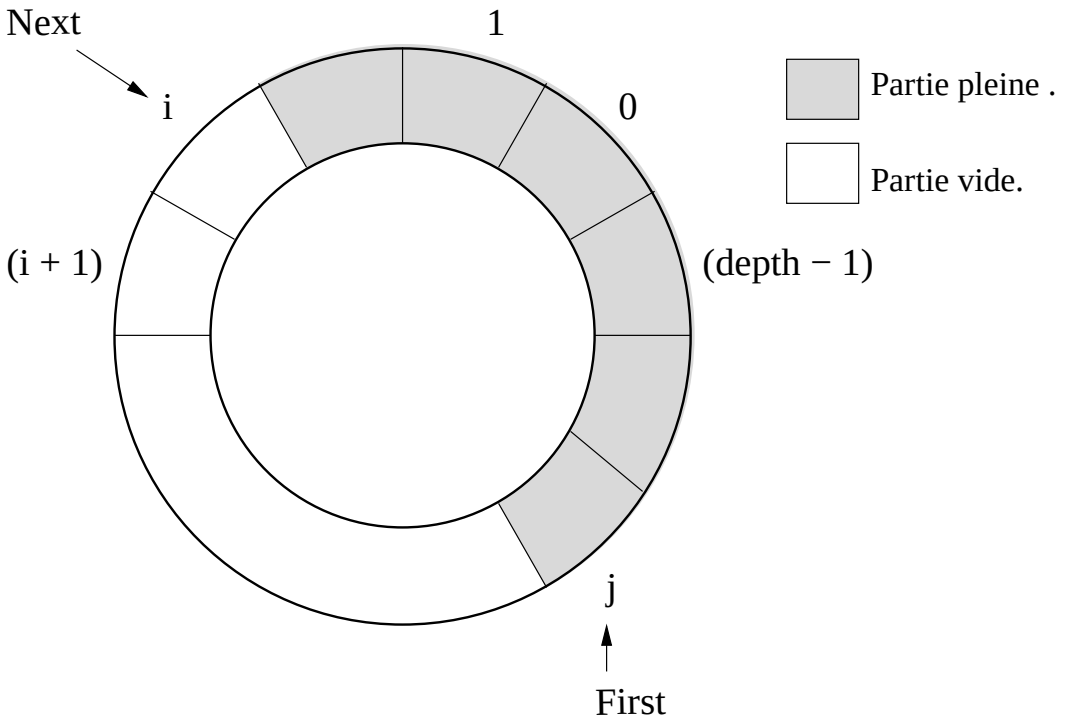

FIG. 3.1 – La structure de file implantée par un vecteur et deux curseurs First et Next (*depth* est la taille des environnements).

3.4 L'implantation

Nous avons implanté ce mécanisme en modifiant le module TCPIP de MSPML.

3.4.1 La structure de données

Pour implanter les nouveaux environnements de communication, on opte pour la structure de file implantée par un vecteur. C'est une structure de données linéaire où les insertions d'éléments se font toutes d'un même côté appelé la queue de la file et les suppressions se font toutes de l'autre coté appelé la tête de la file [29].

Plusieurs implantations des files sont possibles, on adopte celle qui utilise un vecteur pour stocker les éléments ainsi que deux curseurs appelés **first** et **next** indiquant respectivement la tête et la queue de la file (voir figure 3.1).

Il faut considérer le vecteur comme représentant un chaînage circulaire implicite, en d'autres termes, la case 1 suit la case 0, la case 2, suit la case 1, mais aussi la case 0, suit la case $n-1$. C'est alors qu'il faut bien distinguer les deux états *file vide* et *file pleine*.

On définit d'abord un type enregistrement en Objective Caml [7] qu'on nomme **queue**. Les déclaration et les définitions seront données par la suite en pseudo-code pour les raisons d'abstraction et de lisibilité.

```
type queue = {
  first: entier;
  next: entier;
  full: booléen;
  empty: booléen;
  content: vecteur cellule_com_env }
```

où
$first$ et *next* : désignent les curseurs pour repérer la tête et la queue de la file respectivement, comme décrit ci-dessus.
$full$ et *empty* : déterminent les deux états *file pleine* et *file vide* respectivement.
content : Est un vecteur de type **cellule_com_env**, c'est-à-dire, les valeurs sauvegardées à chaque m-étape après leur linéarisation. Il présente concrètement le vecteur **com_env** dans l'ancien mécanisme.
On peut ainsi définir les opérations usuelles de création, d'insertion, de suppression et de réinitialisation sur les files comme suit :

Soient f une file, *len* sa taille et x un élément de celle-ci :

- **La création :** Implantée par la fonction **make_Q** qui reçoit en arguments *nom* et *taille* et retourne une file nommée *nom* et de taille *taille*. Les champs de la nouvelle file sont initialisés comme suit : $nom.first = 0$, $nom.next = 0$, $nom.full = false$ et $nom.empty = true$.

- **L'insertion :** Implantée par la fonction **enQ** qui reçoit en arguments un *nom* d'une file existante, la valeur à insérer dans la file et le *mode* de sauvegarde. Ce dernier correspond dans notre cas aux deux modes **Get** et **Mget** (Voir le chapitre 2). **enQ** retourne la file *nom* avec la valeur à insérer en queue. En implantant cette fonction, on doit être attentif au cas de la file pleine.

- **La suppression :** Implantée par le fonction **deQ** qui reçoit en arguments le *nom* d'une file existante et retourne celle-ci après avoir supprimer l'élément se trouvant en sa tête. En implantant cette fonction, on doit être attentif au cas de la file vide.
- **La réinitialisation :** Implantée par la fonction **reset_Q** qui reçoit en arguments le *nom* d'une file existante et retourne la même file dans son état initial.

3.4.2 Les fonctions de gestion des environnements de communication

On présente ici les deux fonctions principales de notre mécanisme qui sont **update_mstepmin** : int → float− > unit et **clean_env** : unit → unit. Les autres changements induits sur le module TCPIP ne seront pas présentés ici pour les raisons

de longueur.

- **update_mstepmin :**
  Cette fonction implante la mise à jour de la variable **mstepmin** et évite le blocage (comme décrit dans la section 3.3) en demandant la m-étape actuelle des processus qui sont à l'origine du blocage.
- **clean_env :** Le pseudo code de cette fonction est le suivant :

```
debut
    update_mstepmin 10 2;
    si ((mstep % mstep_depth) = (mstep_depth/clean_freq) ou (com_env.full))
        alors
            Pour i = 1 a ((mstepmin − com_env.first) % mstep_depth) faire
                deQ com_env
            finpour;
    fsi;
fin
```

  Cette fonction appelle la fonction *update_mstepmin* puis libère les valeurs inutiles dans l'environnement de communication suivant une fréquence déterminée par le paramètre **clean_freq** au début de l'exécution du programme ou si l'environnement de communication est plein.

3.5 Généralisation

Pour augmenter le taux de mise à jour de la valeur **mstepmin**, on peut changer les données échangées pendant un **get**. Quand un processus j répond à la requête d'un processus i, il peut lui envoyer la réponse plus sa valeur **mstep**. Le processus i peut, ainsi, mettre à jour son vecteur **msteps** à l'indice j.

Il est aussi possible d'échanger des sous-parties du vecteur **msteps** pendant un **get** pour améliorer la mise à jour de la variable **mstepmin**. Pour faire ainsi, on garde un nombre fixe d'identificateurs de processus dont l'information sur leurs m-étapes a été récemment mise à jour. On utilise un paramètre qu'on appelle **sizex** pour fixer ce nombre au début de l'exécution du programme et on définit, ainsi, la taille des sous-parties du vecteur **msteps** à échanger entre les processus qu'on appelle **sp_msteps** qui est un vecteur de paires où la première composante contient l'identificateur du processus et la seconde contient la valeur associée dans le vecteur **msteps** . Pour sauvegarder les identificateurs des processus dont l'information sur leurs m-étapes a été récemment mise à jour, on utilise une structure de données qu'on nomme **sp_msteps_ids** qui est similaire à celle des environnements de communication, c'est à dire, une file d'attente implantée par un vecteur mais sans l'indice de début de la file (**First**). **sp_msteps_ids** n'est jamais pleine (on écrase les ancienne valeurs dans ce cas), elle contient à tout moment soit des identificateurs de processus (entre 0 et $(p-1)$), soit la valeur -1 (correspond à l'initialisation au début de l'exécution).

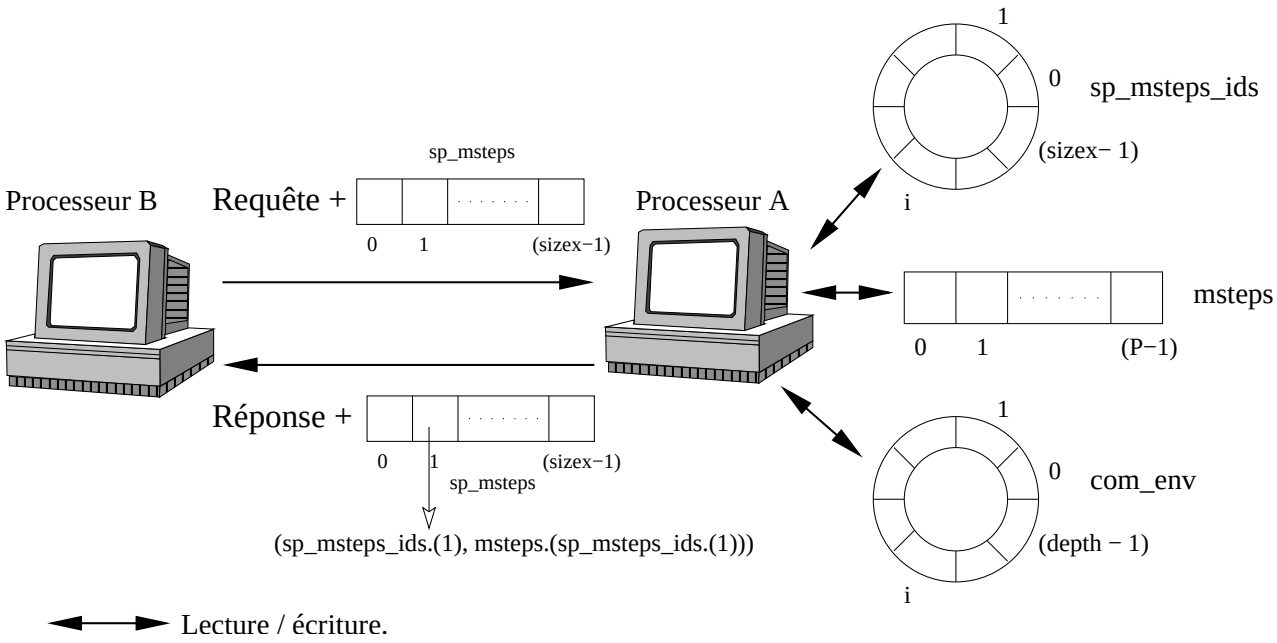

FIG. 3.2 – Le Processeur A reçoit une requête du processeur B contenant une sous-partie de son vecteur **msteps** (**sp_msteps**), il met à jour son vecteur **msteps**, récupère la valeur demandée par B de son environnement de communication et construit la sous-partie de son vecteur **msteps** à partir de son vecteur **sp_msteps_ids**. Il envoie celle-ci avec la réponse au processeur B.

Ce dernier mécanisme doit avoir un surcoût – négligeable – en espace mémoire puisque de nouvelle structures de données sont utilisées par chaque processus et un surcoût en temps de communication puisque la taille des données échangées à chaque opération de communication est plus grande. Cependant, pour une valeur raisonnable de **sizex**, ce surcoût ne doit pas dégrader les performances de façon significative puisqu'en pratique la valeur de la latence du réseau L est beaucoup plus grande que le temps pour échanger un mot entre deux processeurs g (voir le modèle de coût dans 2.1). Ainsi, ce mécanisme est comparable à celui décrit précédemment avec l'avantage d'avoir une mise à jour plus efficace de la variable **mstepmin** et par conséquent, un vidage plus efficace des environnements de communication. Il reste, toutefois, plus intéressant que l'ancien mécanisme qui nécessite une barrière de synchronisation globale – coûteuse et contraire au modèle MPM – pour vider les environnements de communication.

Chapitre 4

Tolérance aux pannes

Étant un outil de meta-computing, MSPML permet la mis en oeuvre des applications à grande échelle comme les grilles de calcul et les grappe de PC [20]. Ce grand potentiel de calcul est heurté par la susceptibilité aux pannes. On présente, dans ce chapitre, deux protocoles de tolérance aux pannes pour MSPML, afin de le rendre plus fiable et plus disponible. Mais d'abord, on présente un aperçu des protocoles existants afin d'ajouter la fiabilité et la disponibilité aux systèmes parallèles et distribués ainsi que les problèmes qui leur sont inhérents.

4.1 Aperçu des approches du Rollback-Recovery

Le *Rollback-Recovery*, noté par la suite RR, est une technique pour rendre un système distribué plus fiable et tolérant aux pannes. On trouve principalement, deux approches pour implémenter un système de RR. Ces deux approches sont le *checkpoint-based RR* et le *log-based RR*. Avant de les décrire brièvement, on donne d'abord quelques définitions qui vont être utiles par la suite.

4.1.1 Définitions

Un état global consistant du système : un état global d'un système est une collection des états individuels de tout ses processus et des états de tous ses canaux de communication [8]. Un état global consistant du système est celui qui apparaît durant une exécution correcte et sans pannes d'une application distribuée, c'est-à-dire, c'est celui dans lequel si l'état d'un processus indique la réception d'un message, alors l'état du processus expéditeur du message indique qu'il a effectivement envoyé ce message [6]. La figure 4.1 montre deux états globaux d'un système distribué : le premier consistant et l'autre inconsistant.
Les messages en transit (in-transit messages) : ce sont des messages qui ont été expédiés mais pas encore reçus.
Support fiable de stockage (reliable storage medium) : il est utilisé par les protocoles du RR pour sauvegarder les points de reprise, l'enregistrement des événements

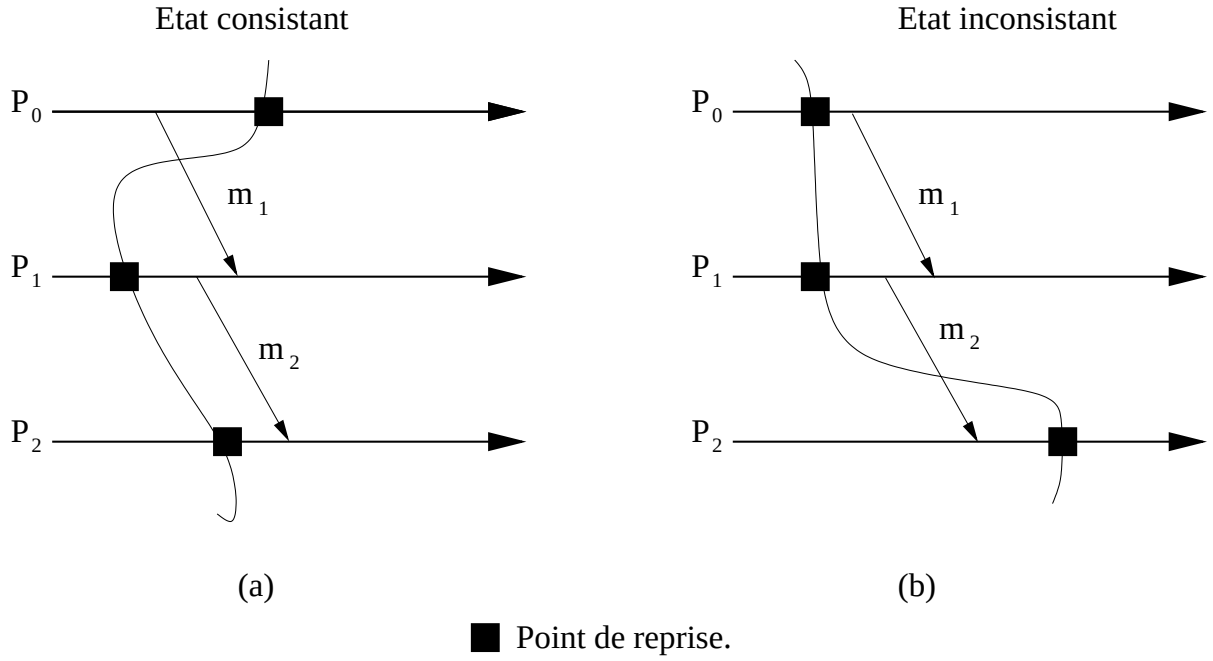

FIG. 4.1 – Exemple d'un état globale consistant(a) et d'un état globale inconsistant(b).

(*events log*) et les autres informations relatives à la ré-exécution (*recovery*). Il doit assurer que les données de la ré-exécution soient persistantes entre les processus en panne et leurs ré-exécutions correspondantes.
L'effet domino (the domino effect) : sous quelques scénarios, la propagation du *Rollback* peut aller jusqu'à l'état initial du calcul, favorisant, ainsi, la perte de tout le travail effectué avant l'apparition de la panne. Cette situation est connue sous le nom de l'effet domino [25]. La figure 4.2 montre un exemple de l'effet domino.

4.1.2 Les approches du Rollback-Recovery

On distingue deux approches pour implanter un système de Rollback-Recovery :

Le Rollback-Recovery basé sur les points de reprise (*checkpoint-based Rollback-Recovery***)**

À la suite d'une panne, ce type de protocole restaure l'état du système associé au plus récent ensemble consistant de points de reprise, cet ensemble est appelé le *Recovery Line* [25]. Ces protocoles sont moins restrictifs et plus simples à implanter par rapport aux protocoles du RR basé sur l'enregistrement (*log-based RR*), mais ils ne garantissent pas que l'exécution avant la panne puisse être régénérée de façon déterministe après le *Rollback* [8]. Les techniques du RR basé sur les points de reprise

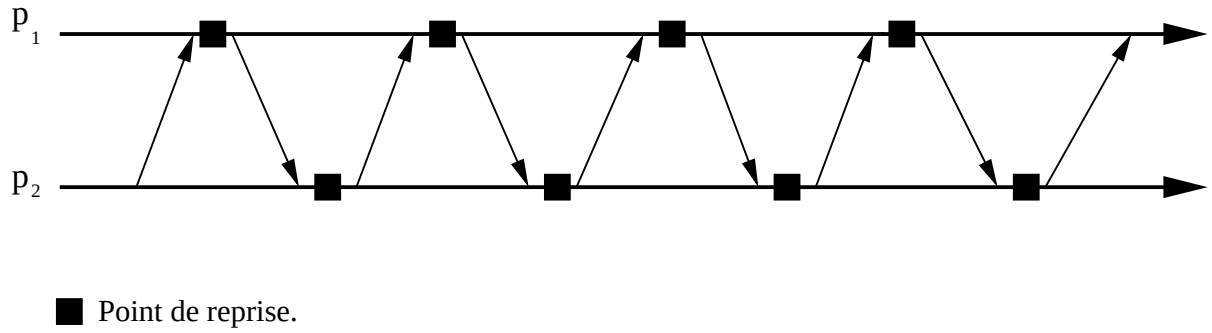

FIG. 4.2 – *L'effet domino :* un scénario comme celui-ci oblige le programme à redémarrer depuis le début dans le cas d'une panne, rendant ainsi tout les points de reprise inutiles !

peuvent être classifiées en trois catégories :

Les points de reprise non coordonné : Chaque processus décide quand un point de reprise doit être effectué. L'avantage est que cette autonomie permet à chaque processus d'effectuer son point de reprise en moment le plus convenable. Par exemple, un processus peut réduire le surcoût de temps en effectuant les points de reprise quand la quantité d'informations est minimale [31]. Les inconvénients de ce protocole sont d'abord, la possibilité de l'apparition de *l'effet domino*, un processus peut effectuer un point de reprise inutile quand ceci ne pourra faire part d'un état global consistant et plusieurs points de reprise doivent être maintenus par chaque processus. Enfin, ce protocole nécessite des algorithmes de *Glanage de Cellules (Garbage Collection)* pour éliminer les points de reprise inutiles.

Les points de reprise coordonnés : Comme son nom l'indique, ce protocole nécessite que tout les processus coordonnent leurs points de reprise pour constituer un état global consistant. Ce protocole simplifie la ré-exécution et n'est pas susceptible d'entraîner l'*effet domino* puisque chaque processus est toujours ré-exécuté à partir de l'état correspondant à son point de reprise le plus récent. Pour le mécanisme basé sur les points de reprise coordonnés, chaque processus n'a besoin de maintenir que son dernier point de reprise sur le support fiable de stockage, diminuant ainsi l'espace de stockage requis et éliminant le besoin pour des algorithmes complexes de glanage de cellules. Le principal inconvénient de ce protocole, à notre avis, est le besoin de synchroniser tous les processus pour effectuer les points de reprise, ce qui va à l'encontre de la philosophie de MSPML.

Les points de reprise induits par les communications : Ce protocole évite *l'effet domino* sans faire recours à la coordination des points de reprise. Un processus effectue deux types de points de reprise : *locaux* et *forcés*. Un point de reprise

local peut être effectué indépendamment des autres processus tandis qu'un point de reprise forcé doit être effectué pour garantir la progression du *Recovery Line* (l'ensemble consistant des points de reprise de tout les processus) et éviter, ainsi, d'effectuer des points de reprise inutiles. Dans ce protocole il n'existe aucun échange de messages de coordination entre les processus pour déterminer quand un point de reprise doit être effectuer, mais plutôt cette information est portée par les messages de l'application. Le destinataire du message décide, alors, s'il doit (ou non) effectuer un point de reprise forcé. Cette décision est prise par le destinataire en fonction du fait que la configuration des communications et des points de reprise ultérieurs puisse mener à la création de points de reprise inutiles ou non. Dans le premier cas, un point de reprise est effectué pour éliminer cette configuration. Cette démarche à été formalisée par une théorie basée sur les notions du *chemin en Z* (*Z-Path*) et le *Cycle en Z* (*Z-cycle*) [8].

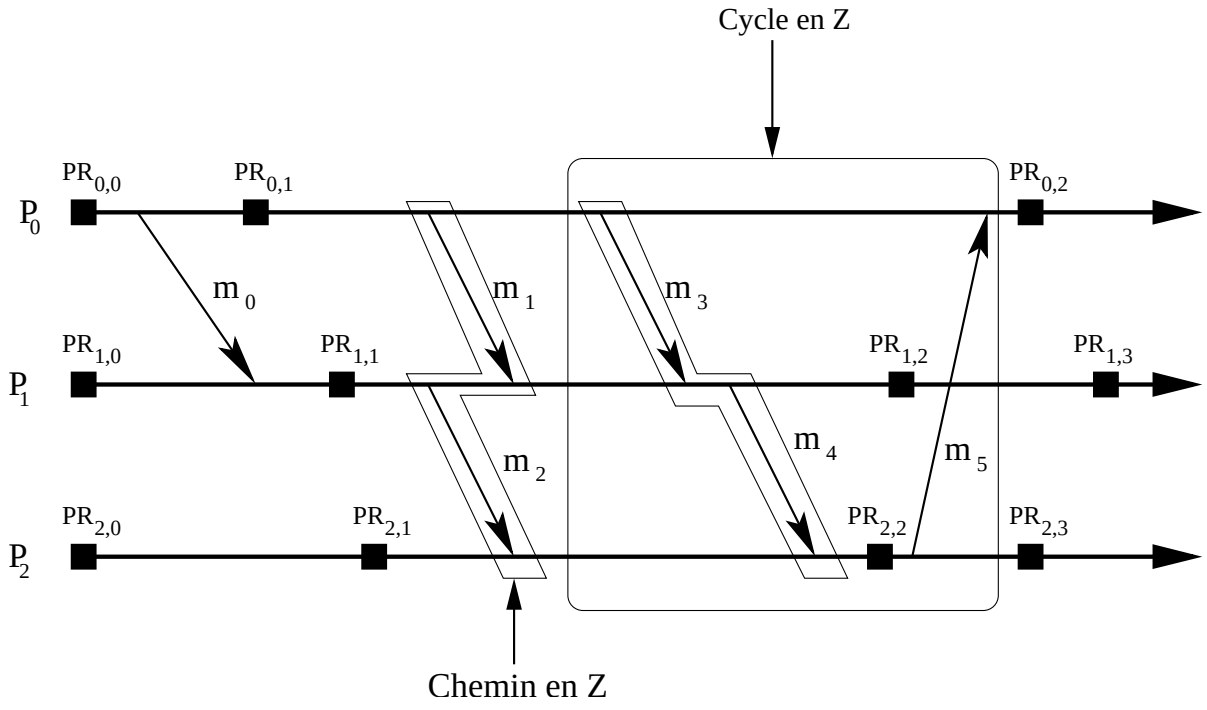

FIG. 4.3 – *Les chemins en Z et les cycles en Z*

.

Un chemin en Z est une séquence particulière des messages qui relient deux points de reprise. Dans la figure 4.3, [m_1,m_2] et [m_3,m_4] représentent des chemins en Z entre les points de reprise $PR_{0,1}$ et $PR_{2,2}$. Un cycle en Z est un chemin en Z qui commence et se termine par le même point de reprise. Le chemin [m_5,m_3,m_4] est un cycle en Z qui commence et se termine par le point de reprise $PR_{2,2}$. On s'intéresse aux cycles en Z puisqu'il a été prouvé qu'un point de reprise est inutile *si et seulement si* il fait part d'un cycle en Z [21]. Ainsi, on peut éviter les points de reprise inutiles en

s'assurant que les chemins en Z ne deviennent jamais des cycles en Z.

Le protocole des points de reprise induits par les communications possède des avantages en théorie tel que plus d'autonomie des processus pour effectuer les points de reprise par rapport à l'approche coordonnée ou encore l'adaptation aux systèmes à grand échelle sans une dégradation significative des performances. Néanmoins, en pratique, il a été montré que les performances se dégradent dès qu'on augmente la taille du système de façon significative et qu'on ne peut prédire l'espace de stockage requis pour une application donnée à cause des points de reprise forcés qui apparaissent à des moments aléatoire de l'exécution. D'autre part, des études expérimentales ont montré qu'en pratique, les points de reprise forcés sont aux moins deux fois plus nombreux que les points de reprise locaux, ce qui signifie une perte du degré d'autonomie des processus qui est lié aux points de reprise locaux [13].

Le *Rollback Recovery* basé sur l'enregistrement (*log-based rollback recovery*)

Ce genre de protocoles utilise le fait qu'une exécution d'un processus peut être modélisée comme étant une séquence d'intervalles d'états déterministes, chacun commence par l'exécution d'un événement indéterministe [28]. Un tel événement peut être la réception d'un message d'un autre processus ou un événement interne au processus lui-même, comme par exemple, les évènements probabilistes. L'envoi d'un message, par contre, n'est pas un événement indéterministe [8]. Le RR basé sur l'enregistrement suppose que tout les événements indéterministes peuvent être identifiés et que les déterminants correspondants peuvent être enregistrés sur un support fiable de stockage.

Durant l'exécution, chaque processus enregistre (*log*) les déterminants de tout les événements indéterministes qu'il observe sur le support fiable de stockage. Un déterminant est une somme d'informations qui permet de ré-exécuter les programmes de telle sorte que les évènements indéterministes soient exécutés à chaque fois de la même façon (leurs effets sur le déroulement du programme est le même). De plus, chaque processus effectue des points de reprise pour réduire le *rollback* à la suite d'une panne et avant la ré-execution. A la suite d'une panne, le processus défaillant est ré-exécuté en utilisant les points de reprise et les déterminants déjà enregistrés pour se comporter de la même façon que l'exécution défaillante (notamment l'ordre de réception des messages). On distingue trois type de protocoles :

L'enregistrement pessimiste (*Pessimistic loging*) : Ce protocole fait l'hypothèse qu'une panne peut apparaître après chaque événement indéterministe durant l'exécution. Ainsi, le déterminant de chaque événement indéterministe est enregistré sur un support fiable de stockage avant que cet événement ne puisse affecter l'exécution, on parle alors d' *enregistrement synchrone* (*synchronous logging*). Les avantages de ce protocole sont d'abord que les processus qui tombent en panne peuvent être ré-exécutés à partir de leur plus récent point de reprise et que la ré-exécution des processus en panne est simplifiée puisque les effets d'une panne n'affectent que le pro-

cessus concerné. Enfin, le glanage de cellules est simple puisque on peut supprimer les anciens points de reprise car ils ne seront plus jamais utilisés.

L'enregistrement optimiste (*Optimistic logging*) : On parle ici d'*enregistrement asynchrone* (*asynchronous logging*) puisque les processus enregistrent les déterminants de façon asynchrone sur le support fiable de stockage [28]. Ce protocole suppose que l'enregistrement va être accompli avant qu'une panne n'apparaisse. Les déterminants sont gardés sur une mémoire volatile et sont périodiquement enregistrés sur le support fiable de stockage. Ainsi, ce protocole ne nécessite pas de bloquer l'application en attendant que les déterminants soient réellement stockés sur le support fiable de stockage. Par contre, on y trouve un mécanisme de ré-exécution et de glanage de cellules plus compliqués.

Si un processus tombe en panne, les déterminants enregistrés dans la mémoire volatile vont être perdus et les intervalles d'états qui débutent par les états correspondants à ces événement ne peuvent, par conséquent, être restaurés au moment de la ré-exécution. Certains auteurs considèrent que ce protocole est le meilleur pour implanter un système de tolérance aux pannes [24]. Ainsi, il sera utilisé dans notre mécanisme puisqu'il est de plus, le plus proche et même naturel par rapport à la conception de MSPML.

L'enregistrement causal (*Causal logging*) Ce protocole possède les performances de l'enregistrement optimiste en retenant les avantages de l'enregistrement pessimiste. Comme l'enregistrement optimiste, il évite l'accès synchrone au support de stockage et comme l'enregistrement pessimiste, Il donne plus d'autonomie aux processus en les isolant des effets des pannes qui peuvent apparaître dans les autres processus. Pour ce faire, l'enregistrement causal assure que le déterminant de chaque événement indéterministe qui précède causalement l'état du processus soit stocké dans le support fiable de stockage ou bien disponible localement dans la mémoire volatile du processus lui même. Dans l'exemple de la figure 4.4, les messages m_4 et m_5 peuvent être perdus après une panne des processus P_1 et P_2, le processus P_0 à l'état X doit avoir enregistré les déterminants des événements indéterministes qui précèdent causalement cet état, à savoir, la réception des messages m_0,m_1,m_2 et m_3. Les déterminants de chacun de ces événements indéterministes sont soit enregistrés sur le support fiable de stockage, soit disponibles dans la mémoire volatile du processus P_0. Ces déterminants contiennent l'ordre dans lequel leurs récepteurs d'origine ont reçu les messages correspondants. L'expéditeur du message enregistre le contenu de celui-ci. Ainsi, le processus P_0 sera capable de guider la ré-exécution de P_1 et P_2 puisqu'il connaît l'ordre dans lequel P_1 doit recevoir le message m_0 et m_2 pour atteindre l'état dans lequel P_1 envoie le message m_3 et ainsi de suite.

Dans ce protocole, un processus fait porter les déterminants existants dans sa mémoire volatile sur les messages qu'ils envoie. Après la réception d'un message, le processus d'abord ajoute tout les déterminants portés par celui-ci dans sa mémoire volatile, puis, délivre le message à l'application.

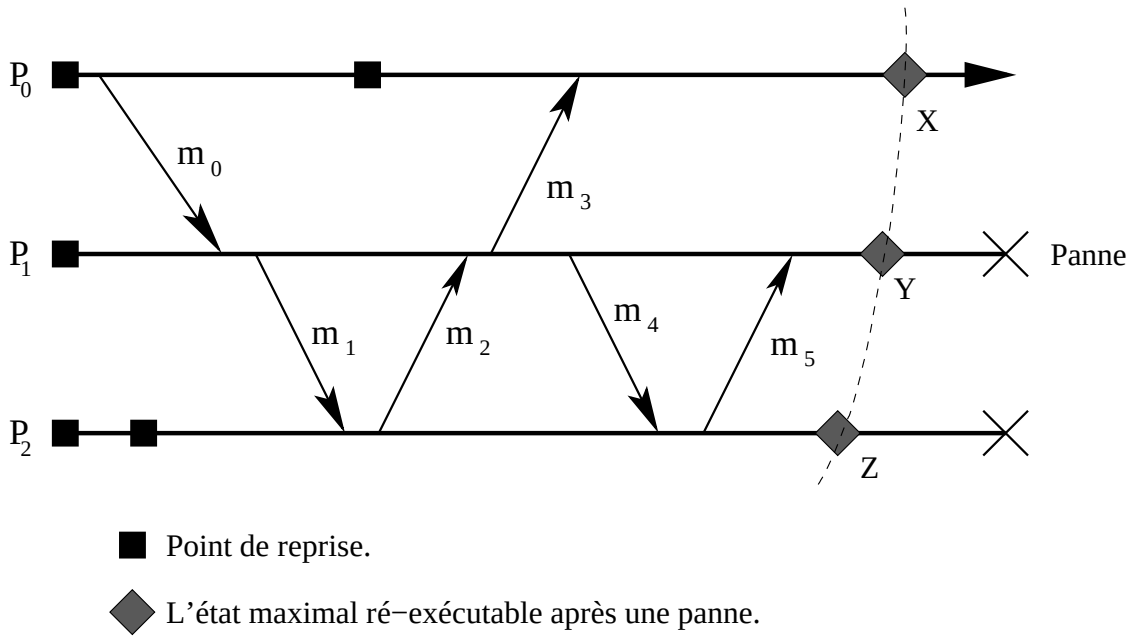

FIG. 4.4 – L'enregistrement causal.

L'enregistrement causal limite le *Rollback* d'un processus qui tombe en panne au plus récent point de reprise. Cela réduit l'espace de stockage utilisé et la quantité du travail à refaire en cas de panne (et le temps supplémentaire requis) au détriment d'un mécanisme de ré-exécution plus complexe.

4.2 Un premier mécanisme de tolérance aux pannes pour MSPML

Dans un premier temps on propose un mécanisme qui suit le protocole d'enregistrement optimiste et qu'on appel Fault Tolerant Minimally Synchronous Parallel ML 1 (FT-MSPML1). Ce choix est justifié, d'une part, par l'efficacité et la simplicité de ce protocole et d'autre part, par son adaptation totale aux avantages de l'asynchronisme qu'offre MSPML. FT-MSPML1 n'utilise pas les points de reprise, il s'appuie complètement sur l'enregistrement des messages et la ré-exécution d'un processus en panne depuis son état initial. Cette propriété qui peut être perçue comme étant pénalisante possède des avantages en performances puisque les exécutions correctes seront beaucoup moins pénalisées par des points de reprise souvent coûteuses. De plus, FT-MSPML1 ne gère pas les programmes probabilistes.

4.2.1 Le mécanisme

Par ce mécanisme on essaye de fournir la propriété de tolérance aux pannes pour MSPML tout en effectuant un minimum de changements sur le fonctionnement de celui-ci. Durant une exécution d'un programme MSPML, les processus enregistrent dans leurs environnements de communication leur valeurs respectives à chaque m-étape (voir le mécanisme décrit dans 3.1). Ceci peut être utilisé pour enregistrer ces valeurs avec les numéros des m-étapes sur le support de stockage. A la suite d'une panne d'un processus, ceci reprend son exécution depuis la m-étape numéro 0. Ce qui change dans le protocole décrit précédemment (dans la section 3.1) est qu'un processus qui reçoit un *request*, dans le cas où il ne trouve pas la valeur demandée dans son environnement de communication doit accéder à son support de stockage, la récupérer et la renvoyer au processus demandeur.

Il est à noter qu'il faut prêter l'attention au mécanisme de vidage du contenu des environnements de communication pour qu'il n'efface des valeurs avant qu'elles ne soient stockées dans le support de stockage.

4.2.2 L'architecture

Pour que le système détecte les pannes des processus et les ré-exécute on dote chaque processus d'un programme MSPML d'un *Démon*. L'ensemble des Démons ont pour mission de gérer les communications entre les processus et détecter les pannes éventuelles de ceux-ci. Dans ce dernier cas, les Démons déroutent les erreurs liées à la communication vers un *dispatcher* qui se charge de la ré-exécution des processus déclarés en panne. Pour ce faire, le dispatcher possède des informations globales sur la machine parallèle et peut donc désigner de manière efficace le noeud où un processus en panne doit être ré-exécuté suivant une politique prédéfinie (le noeud qui possède plus de capacité de calcul, qui a moins de charges, etc.). Les Démons et le dispatcher échangent des messages de contrôle.

4.3 Un deuxième mécanisme de tolérance aux pannes pour MSPML

Si dans le premier mécanisme on avait le but de garder, tant que possible, le fonctionnement actuel de MSPML, dans ce deuxième mécanisme nos effort se concentrent plus sur la fiabilité (notamment des programmes MSPML probabilistes ou aléatoires). Supposant un programme MSPML indéterministe (on pense ici à des algorithmes avec hasard ou probabilistes) qui s'exécute en suivant le protocole FT-MSPML1, un processus qui tombe en panne se voit ré-exécuté depuis son état initial, ce qui pose un vrai problème puisque la ré-exécution ne sera peut être pas identique à l'exécution initiale, rendant ainsi l'état global du système inconsistant. Dans l'exemple de la figure 4.6, pour obtenir un état global consistant après la ré-

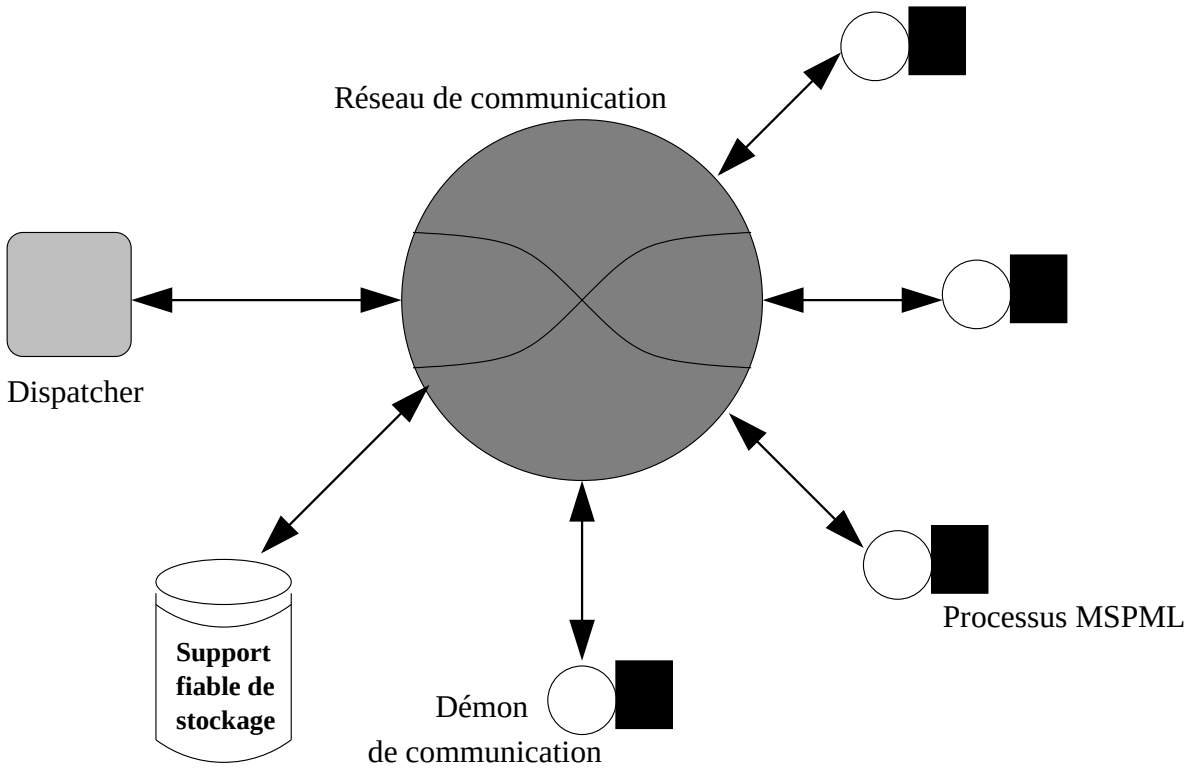

FIG. 4.5 – L'architecture de **FT-MSPML1** et **FT-MSPML2**.

exécution du processus P_1 à partir de son dernier point de reprise X, le processus P_0 doit être ré-exécuté à partir de la m-étape numéro i puisque le processus P_1 a effectué des opérations aléatoires (par exemple, une valeur tirée au hasard qui est communiquée au processus P_0) après le point de reprise X et avant sa panne.

Pour remédier à cette problématique, on propose un second mécanisme qu'on appel Fault Tolerant Minimally Synchronous ML 2 (FT-MSPML2).

4.3.1 Le mécanisme

Ce mécanisme combine le protocole de l'enregistrement optimiste avec le protocole des points de reprise non coordonnés. On utilise ce dernier avec un mécanisme d'enregistrement des *request* reçues par les processus à chaque m-étape. Les points de reprise sont effectués de façon indépendante par chaque processus rendant le calcul effectué jusqu'à présent définitif. En cas d'une panne d'un processus, il sera ré-exécuté à partir de son dernier point de reprise valide. Ceci peut rendre les états des autres processus inconsistants, notamment ceux qui ont demandé des valeurs au processus ré-exécuté dans une opération de communication entre le dernier point de reprise et le moment de l'apparition de la panne. Pour éviter ce problème, chaque processus enregistre les requests reçues par les autres processus à chaque m-étape

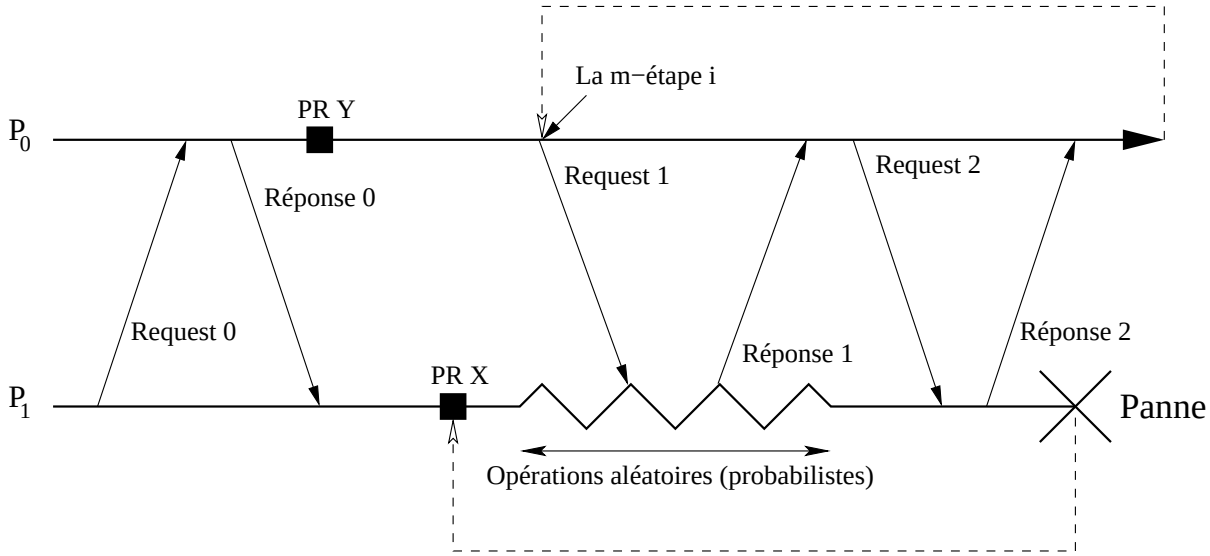

FIG. 4.6 – Un programme MSPML en exécution suivant le protocole FT-MSPML2.

dans une structure de données (une pile par exemple) dont le contenu est stocké sur le support de stockage à chaque fois qu'elle est pleine. On peut avoir, ainsi, une liste des processus qui ont fait des requests à un processus donné à chaque m-étape. Lors de sa ré-exécution, si un processus recommence à s'exécuter à partir de la m-étape numéro i, il doit recenser tout les processus qui lui ont adressé des requests à partir de celle-ci en précisant pour chacun parmi eux la m-étape associée à son premier request après i. Il leur demande, ainsi, de se ré-exécuter à partir de cette m-étape. Cette solution permet d'éliminer tout état inconsistant d'un processus durant l'exécution. Il faut prévoir un mécanisme de glanage de cellules qui se charge de supprimer les points de reprise inutiles.

4.3.2 L'architecture

Ce mécanisme possède presque la même architecture que FT-MSPML1. On garde les Démons qui gèrent les communications et le dispatcher qui se charge en plus d'ordonner les processus de se ré-exécuté à partir d'une m-étape donnée après avoir relancer l'exécution d'un processus qui tombe en panne. Ces opérations sont effectuée grâce à des messages de contrôle entre les Démons et les dispatcher. Un module de points de reprise est ajouté à l'architecture de FT-MSPML1 afin d'être utilisé par chaque processus durant l'exécution. Il se charge aussi d'éliminer les points de reprise inutiles.

4.4 L'implantation

En ce moment, des efforts sont fournis pour mettre en oeuvre le mécanisme de FT-MSPML1 afin qu'il soit présent dans la prochaine distribution de MSPML. Les difficultés se trouvent essentiellement dans l'implantation efficace des Démons et du dispatcher qui constitue la pierre angulaire de ce mécanisme. Initialement, on s'est intéressé au toolkit de groupes de communication appelé *Ensemble* [11]. Il offre des fonctionnalités intéressantes pour gérer efficacement les communications entre les processus via des Démons et détecte les pannes des processus. Ce qui est intéressant aussi dans *Ensemble* est le fait qu'il soit implanté en Objective Caml, ce qui facilite son intégration dans MSPML. Malheureusement, il s'est avéré que ce système n'est pas bien entretenu puisque les versions antérieures sont liée à des versions dépassées d'Objective Caml et les dernières versions éliminent des fonctionnalités primordiales pour MSPML tel que le mode de communication TCP/IP (c'est une tendance qui suit les objectifs des travaux de recherche de ses concepteurs qui ne vont pas forcément dans la direction des nôtres). Il est à noter que le mécanisme de stockage des valeurs des environnements de communication dans des fichiers sur le disque ainsi que leur lecture lors d'une panne nécessite la prudence et la minutie lors de son développement.

Du point de vue des performances, FT-MSPML1 nécessite un surcoût en temps peu important dans le cas d'une exécution correcte (sans pannes) puisque le stockage des valeurs des environnements de communication sur le disque peut être effectué sans interrompre l'exécution des processus en attribuant des processus léger (*Threads*) dédiés à cette tâche. En cas d'une panne, le surcoût en temps devient plus important puisque le processus en ré-exécution doit demander toutes les valeurs qu'il a déjà demandé précédemment aux autres processus qui doivent éventuellement accéder à leurs supports de stockage pour récupérer ces valeurs sachant qu'il y aura, probablement, des processus bloqués qui attendent que le processus ré-exécuté avance pour récupérer une valeur de celui-ci qui soit liée à une m-étape postérieure à la m-étape où la panne est apparue.

Pour remédier à ce problème, on peut changer légèrement le mécanisme de FT-MSPML1 en mettant l'enregistrement des valeurs du coté des récepteur des messages. Ceci est fait dans une structure de données dont le contenu sera régulièrement stocké sur le support de stockage. Ainsi, lors de la ré-exécution d'un processus en panne, ceci sera totalement autonome jusqu'au niveau où la panne est apparue puisqu'il avait déjà enregistré toutes les valeurs qu'il a reçu précédemment. Cela complique le mécanisme de tolérance aux pannes mais peut avoir des avantages en terme de performances.

Pour FT-MSPML2, l'effort doit de plus se concentrer autour du module des points de reprise. Des systèmes tolérants aux pannes tel que *MPICH-V2* [5] utilisent des bibliothèques dédiées (Condor dans ce cas). D'autres utilisent leurs propres implantation comme dans le cas de *Starfish* [1]. Dans notre cas, on a pensé à explorer un langage fonctionnel dédié aux systèmes distribués appelé Acute [22]. Ce langage

est très intéressant du fait qu'il possède des primitives dans sa bibliothèque standard qui servent à la migration de processus et qui peuvent être utilisées pour effectuer les points de reprises. L'inconvénient qui nous rend hésitants à son utilisation pour notre mécanisme est sa lenteur reporté par ces concepteurs dans son manuel de référence (il peut être jusqu'à 3000 fois plus lent qu'Objective Caml). Ce qui est sûr est qu'un mécanisme de points de reprise - même non coordonné - augmente le surcoût en temps puisqu'ils doit bloquer les processus au moment d'effectuer les points de reprise.

Chapitre 5

Conclusions

Dans ce mémoire, on a présenté dans une première partie un nouveau mécanisme de gestion des environnements de communications qui sont à la base du mécanisme de communication dans MSPML [17]. On a ensuite présenté une implantation de celui en expliquant les structures de de données et les fonctions nécessaires. Pour aller plus loin et être fidèle aux aux proposition de [17], on a aussi exploré d'autres possibilités qui améliorent ce mécanisme en augmentant la quantité d'informations échangées entre les processus d'un programmes MSPML. Grâce à ce travail, les versions ultérieures de MSPML qui seront équipées de ce mécanisme sophistiqué de communication doivent se montrer plus performantes puisque les barrières de synchronisation seront éliminée avec leur surcoût en temps. Cela donne aussi plus d'asynchronisme à MSPML et renforce la démarche pour laquelle il a été conçu.

Dans une seconde partie, on a étudié la possibilité de doter MSPML d'un mécanisme de tolérance aux pannes, un aspect qui est aujourd'hui indispensable pour les systèmes parallèles et distribués, notamment avec le phénomène de *Grids* de calcul et de *Clusters* de PC's qui sont de plus en plus nombreux et où le risque de pannes est plus grand. Après avoir étudié les différents techniques développées dans ce domaine, on a proposé deux mécanismes de tolérance aux pannes pour MSPML. Le premier mécanisme étant très proche du fonctionnement de MSPML a pour but de fournir plus de fiabilité à celui-ci avec un minimum de changements et sans dégrader ses performances. Ce mécanisme suit une approche d'enregistrement optimiste.

Pour introduire plus de fiabilité et de disponibilité, on a proposé un deuxième mécanisme qui couple l'enregistrement optimiste avec les point de reprise non coordonnées. Notre étude nous a permis de soulever les difficultés liée à l'implantation de ce type de mécanismes. On peut également explorer la possibilité d'effectuer des points de reprise au niveau de la machine virtuelle pour adapter notre système aux architectures hétérogènes, concrètement, pouvoir exécuter un processus dans une machine A en effectuant des points de reprises périodiquement et le ré-exécuté, ensuite, dans une autre machine B d'architecture différente [2]. Cette propriété peut aussi être utile pour la migration de processus dans un mécanisme d'équilibrage de charges pour des travaux futurs.

Sur le plan personnel, ces travaux que j'ai mené au sein de l'équipe de recherche du projet **PROPAC** (PROgrammation PArallèle Certifiée) dans le cadre d'un stage d'initiation à la recherche ont été très enrichissants pour moi. Outre les compétence techniques que j'ai acquis telles que :

- L'approfondissement des mes connaissances acquises durant la partie théorique dans le domaine du parallélisme ;
- La mise en pratique de mes connaissance sur le paradigme fonctionnel de programmation en utilisant le langage Objective Caml ;
- Acquérir une expérience dans la programmation concurrente et la programmation distribuée et parallèle ;

Ce stage m'a permis d'être impliqué dans une activité de recherche concrète, dans un domaine d'actualité où la recherche est très active. Ce stage m'a permis, aussi, de valider mon choix de poursuivre mes études dans une voie de recherche.

Bibliographie

[1] A. Agbaria and R. Friedman. Starfish : Fault-tolerant dynamic MPI programs on clusters of workstations. In *Proceedings of IEEE Symposium on High Performance Distributed Computing*, pages 167–176, 1999.

[2] A. Agbaria and Roy Friedman. Virtual-machine-based heterogeneous checkpointing. *Softw. Pract. Exper.*, 32(12) :1175–1192, 2002.

[3] G. Akerholt, K. Hammond, S. Peyton-Jones, and P. Trinder. Processing transactions on GRIP, a parallel graph reducer. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, Munich, June 1993. Springer.

[4] V. Blanco, J. A. González, C. León, C. Rodríguez, G. Rodríguez, and M. Printista. Predicting the performance of parallel programs. *Parallel Computing*, 30 :337–356, 2004.

[5] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2 : a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Super Computing 2003*, 2003.

[6] K. M. Chandy and L. Lamport. Distributed Snapshots : Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1) : 63–75, 1985.

[7] G. Cousineau and M. Mauny. *Approche Fonctionnelle de la Programmation*. Ediscience International, 1995.

[8] Y. M. Wang E. N. Elnozahy, L. Alvisi and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3) : 375–408, 2002.

[9] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 215–229. Springer Verlag, 2003.

[10] G. Hains and F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation : Theory and Practice, pages 165–178. Nova Science Publishers, august 2002.

[11] M. Hayden. The ensemble system. Technical Report TR98-1662, 1998.

[12] Y. Kee and S. Ha. An Efficient Implementation of the BSP Programming Library for VIA. *Parallel Processing Letters*, 12(1) :65–77, 2002.

[13] S. Rao S. A. Husain L. Alvisi, E. N. Elnozahy and A.D. Mel. An analysis of communication induced checkpointing. In *Symposium on Fault-Tolerant Computing*, pages 242–249, 1999.

[14] F. Loulergue. Distributed Evaluation of Functional BSP Programs. Talk at the International Workshop on High Level Parallel Programming, Orléans, march 2001.

[15] F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In S. Gilmore, editor, *Trends in Functional Programming, Volume 2*, pages 77–88. Intellect Books, 2001.

[16] F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14*th IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.

[17] F. Loulergue. Management of Communication Environments for Minimally Synchronous Parallel ML. In Z. Juhasz, P. Kacsuk, and D. Kranzimuller, editors, *Distributed and Parallel Systems (DAPSYS 2004)*, pages 185–192. Springer, 2004.

[18] F. Loulergue, F. Gava, M. Arapinis, and F. Dabrowski. Semantics and Implementation of Minimally Synchronous Parallel ML. *International Journal of Computer and Information Science*, 5(3) :182–199, 2004. W. Dosch, editor, special issue on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing.

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

[20] Frédéric Loulergue. *Programmation fonctionnelle d'ordinateurs parallèles et de méta-ordinateurs : sémantiques, systèmes et preuves.* Mémoire d'habilitation à diriger des recherches, University Paris Val de Marne, décembre 2004.

[21] R.H.B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2) :165, 1995.

[22] K. Wansbrough F. Z. Nardelli M. A. Williams P. Habouzit V. Vafeiadis P. Sewell, J. J. Leifer. Acute : high-level programming language design for distributed computation. In *The 10th ACM SIGPLAN International Conference on Functional Programming*, 2005.

[23] P. Panangaden and J. Reppy. The essence of concurrent ML. In F. Nielson, editor, *ML with Concurrency*, Monographs in Computer Science. Springer, 1996.

[24] J. S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, 1997.

[25] B. Randell. System Structure for Software Fault Tolerance. *IEEE Trans. on Software Engineering*, 1(1) : 220–232, 1975.

[26] J. L. Roda, C. Rodríguez, D. G. Morales, and F. Almeida. Predicting the execution time of message passing models. *Concurrency : Practice and Experience*, 11(9) :461–477, 1999.

[27] C. Rodriguez, J.L. Roda, F. Sande, D.G. Morales, and F. Almeida. A new parallel model for the analysis of asynchronous algorithms. *Parallel Computing*, 26 :753–767, 2000.

[28] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computing Systems*, 3(3) : 204–226, 1985.

[29] V. Sébastien. *Approches impérative et fonctionnelle de l'algorithmique :Applications en C et en CAML Light*. Springer, 1999.

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

[31] Y. M. Wang. Reducing message logging overhead for log-based recovery. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 1925–1928, 1993.