Bulk Synchronous Parallel ML avec exceptions

Louis Gesbert[1], Frédéric Gava[1], Frédéric Loulergue[2] and Frédéric Dabrowski[3]

[1] : Laboratoire d'Algorithmique, Complexité et Logique – Université Paris XII Val-de-Marne
[2] : Laboratoire d'Informatique Fondamentale d'Orléans – Université d'Orléans
[3] : Institut de Recherche en Informatique et Automatique – Sophia-Antipolis

Résumé
Bulk Synchronous Parallel ML est un langage de programmation parallèle basé sur Objective Caml et sur le modèle BSP. Visant à être un langage générique, BSML tente d'offrir toutes les fonctionnalités des langages modernes. La gestion des exceptions est une de ces caractéristiques très souhaitables, mais elle pose des problèmes spécifiques au parallélisme sur une machine multi-processeurs. Cet article offre une méthode de gestion des exceptions en BSML, et présente son implantation ainsi qu'un exemple concret d'utilisation dans un algorithme parallèle.

Mots-clés : Programmation parallèle, Programmation fonctionnelle, Exceptions, Calcul haute performance

1. Introduction

Les langages et outils de développement répandus pour la programmation parallèle restent difficiles d'accès pour les utilisateurs non spécialistes. En effet ils se trouvent confrontés à des problèmes de programmation concurrente – interblocage, indéterminisme – ce qui les écarte de leur domaine d'application. Les utilisateurs novices ne sont donc pas uniquement intéressés par les performances pures de leurs applications mais également au temps mis à développer ces applications, et en particulier au temps nécessaire à obtenir une version correcte de celle-ci. Notons également que si l'on prend en compte l'utilisation des ressources pour un programme erroné qui finalement ne délivre pas de résultats, la performance moyenne de l'application peut diminuer très sensiblement.
Nous nous employons donc à développer des langages de programmation basés sur des langages de haut-niveau, et plus spécifiquement les langages fonctionnels de la famille ML. Bulk Synchronous Parallel ML (BSML) est l'un de ces langages [14]. Il tente de se placer au point d'équilibre entre l'expressivité des aspects parallèles et la simplicité d'utilisation. BSML laisse à l'utilisateur la possibilité de concevoir ses propres algorithmes parallèles, tout en le libérant des contraintes de la gestion bas niveau de la concurrence. Une bibliothèque standard permet dans une certaine mesure – de plus en plus importante au fur et à mesure que la bibliothèque standard se développe [7] – à l'utilisateur de programmer son application par l'utilisation de fonction d'ordre supérieur capturant un certain nombre de patrons parallèles. La programmation BSML est alors ainsi similaire aux approches par squelettes algorithmiques [5] à la différence qu'il est toujours possible en BSML de programmer dans le même langage de nouveaux squelettes – mais en utilisant les primitives – ce qui n'est pas le cas pour les approches à squelettes courantes. De part la sémantique même de BSML – basée sur une extension confluente du λ-calcul [15], les erreurs que peuvent commettre les utilisateurs sont beaucoup plus réduites que dans le cas de C+MPI par exemple. Il est même possible d'aller plus loin dans la confiance que l'on peut avoir des programmes BSML en extrayant les programmes BSML [6] de preuves faites avec l'assistant de preuve Coq [1].
BSML suit le modèle BSP (pour Bulk Synchronous Parallel [18, 2]. Ce modèle, outre qu'il offre un modèle de coûts simple et efficace, évite complètement les interblocages. La phase de communication, en BSP, est collective : elle implique tous les processeurs. BSML, pour l'instant implanté en tant que bibliothèque pour Objective Caml, dispose de toutes les facilités que fournit son langage hôte, ce qui lui permet d'être un langage généraliste à la hauteur des langages séquentiels modernes en termes de fonctionnalités.

Cependant, toutes ces fonctionnalités ne supportent pas bien le parallélisme et certaines ne sont pas compatibles avec les propriétés de sûreté que doit respecter BSML. La gestion des exceptions telle qu'elle est gérée dans Objective Caml entre dans cette catégorie ; cet article propose de l'étendre et de l'adapter aux contraintes de la programmation parallèle en BSML. Il n'y a pas à notre connaissance de travaux similaires : il y a de nombreux travaux sur les exceptions dans des langages concurrents, mais la structure plus forte des programmes BSP permet une solution tout à fait différente, et ce problème pour les bibliothèques BSP existantes n'est pas traité puisque la plupart de ces bibliothèques sont pour Fortran ou C.
La section 2 introduit le langage BSML. La section 3 donne une étude des problèmes que posent les exceptions d'Objective Caml si on les porte telles quelles en BSML, suivie par notre proposition de solution qu'explique la section 4. L'implantation de la gestion d'exceptions sûre en BSML est décrite dans la section 5, accompagnée d'un exemple d'utilisation et de résultats section 6. Nous concluons et donnons les pistes futures en section 7.

2. Programmation fonctionnelle Bulk Synchronous Parallel

bsp_p: unit→int
bsp_g: unit→float
bsp_l: unit→float

mkpar: (int →α) →α **par**
apply: (α →β) **par** →α **par** →β **par**
put: (int→α option) **par** →(int→α option) **par**
proj: α option **par** →int →α option

FIG. 1 – Primitives

Nous supposons que le lecteur a des connaissances de base du modèle BSP [20, 17, 18]. Il n'y a pas d'implantation d'un langage Bulk Synchronous Parallel ML complet mais une implantation sous forme d'une bibliothèque pour le langage Objective Caml [11]. Cette bibliothèque est basée sur les primitives données à la figure 1. Il est à noter que les primitives ont évolué par rapport à la première proposition [12] pour se rapprocher de la sémantique, le BSλ-calcul, qui en retour a également évolué [13]).
Tout d'abord, BSML offre l'accès aux paramètres BSP de l'architecture parallèle sous-jacente. Ainsi **bsp_p**() donne le nombre p de processeurs (constant durant l'exécution tant que la juxtaposition parallèle n'est pas utilisée), **bsp_g**() et **bsp_l**() les paramètres BSP g et L.
Ensuite BSML offre une structure de données parallèle, le vecteur parallèle, de type α **par** : ceci indique qu'on a p valeurs (une par processeur) de type α. Ce type α ne doit pas être un type contenant lui-même une occurrence de **par**. Les programmes BSML s'écrivent comme des programmes Caml usuels en utilisant les quatre opérations pour la création et la manipulation des vecteurs parallèles. Un programme BSML est donc un programme séquentiel sur une structure de données parallèle.
Ceci est très différent de la programmation SPMD telle qu'on la trouve en utilisant par exemple le langage C et la bibliothèque MPI. Dans ce cas, un programme parallèle est p copies d'un programme séquentiel contenant des appels à des fonctions de communication. Le même programme est utilisé sur tous les processeurs et le comportement dépend de l'identifiant de processeur (pid) qui est lié extérieurement au programme. Ainsi le comportement local à un processeur ne peut être distingué du comportement global du programme parallèle que dynamiquement, ce qui rend la conception et la correction de tels programmes difficiles. Notons que l'utilisation des fonctions collectives de MPI à de nombreux avantages par rapport à l'utilisation des fonctions de communication point-à-point dont celui de donner une vue plus globale de l'algorithme parallèle et sont donc préférables [9]. Il existe d'ailleurs des approches visant à remplacer automatiquement des appels de fonctions de communication point à point par des appels à des fonctions collectives [16]. De plus seules les fonctions collectives peuvent être implémentées de façon plus sûre permettant la vérification des arguments des fonctions collectives à l'exécution [19].
La phase de calcul asynchrone d'une super-étape BSP peut être programmée en BSML à l'aide des deux primitives **mkpar** et **apply**. Des appels successifs à ces primitives ne nécessitent aucune barrière de synchronisation.

La première permet de créer un vecteur parallèle à partir d'une fonction. (**mkpar** f) va donner le vecteur parallèle suivant :

v_0	$\cdots$	v_i	$\cdots$	v_{p-1}

où au processeur i, $(f\ i)$ a été évaluée en v_i.
La seconde permet d'appliquer un vecteur parallèle de fonctions à un vecteur parallèle d'arguments :

$$(\textbf{apply}\ \boxed{f_0 \mid \cdots \mid f_i \mid \cdots \mid f_{p-1}}\ \boxed{v_0 \mid \cdots \mid v_i \mid \cdots \mid v_{p-1}})$$
$$= \boxed{v'_0 \mid \cdots \mid v'_i \mid \cdots \mid v'_{p-1}}$$

où au processeur i, $(f_i\ v_i)$ a été évaluée en v'_i.
Le coût BSP pour ces deux primitives est $\max_{0 \leq i < p} w_i$ où w_i est le temps nécessaire à l'évaluation au processeur de i de l'expression $(f\ i)$ pour **mkpar** ou de l'expression $(f_i\ v_i)$ pour **apply**.
Contrairement à BSPlib [10] ou PUB [3] nous ne distinguons pas phase de communication et phase de synchronisation globale. Les deux primitives **put** et **proj** permettent de programmer la phase de communication et indissociablement la phase de synchronisation.
put utilise le type α option défini par : **type** α option = None | Some **of** α
L'argument de cette primitive est un vecteur parallèle de fonctions qui décrivent les messages *à envoyer*. La fonction f_i au processeur i indique pour chaque processeur de destination j, soit la donnée à envoyer $(f_i\ j) =$ Some v, soit la constante None signalant qu'aucune donnée ne sera envoyée de i vers j.
Le résultat est encore un vecteur parallèle de fonctions, qui décrivent les messages *reçus*. Ainsi la fonction g_j au processeur j appliquée à i donnera Some v si le processeur i a envoyé la valeur v au processeur j, et donnera None si i n'a rien envoyé à j.
Le coût BSP est le coût d'une super-étape entière. Les messages à envoyer sont évalués (phase de calcul asynchrone) puis envoyés et l'étape se termine par une barrière de synchronisation.
La dernière primitive, **proj**, est utilisée pour permettre de prendre des décisions globales sur le contrôle parallèle d'après des valeurs locales, par exemple pour des algorithmes comme :

Repeat Parallel Iteration **Until** Max of local errors $< \epsilon$.

proj prend en argument un vecteur parallèle de valeurs optionnelles $\langle\ v_0\ , \ldots,\ v_{p-1}\ \rangle$ et diffuse les valeurs qui ne sont pas None pour obtenir à la fin une fonction f de type *séquentiel* telle que $\forall i.(f\ i) = v_i$.

3. Les exceptions d'Objective Caml et BSML

Objective Caml offre un puissant système d'exceptions pour gérer les cas exceptionnels aussi bien que les erreurs. La déclaration d'une nouvelle exception Exc, contenant un paramètre de type typ, s'écrit **exception** Exc **of** typ. On peut considérer les exceptions Objective Caml comme un type variant qui aurait la spécificité d'être extensible : le filtrage de motifs fonctionne de la même façon. On déclenche une exception avec le mot-clé **raise**, soit la syntaxe **raise** (Exc x) si x est de type typ. Une exception levée se propage en remontant dans la pile jusqu'à ce qu'elle rencontre la limite d'un bloc de rattrapage de la forme **try**...**with** Exc x →t qui reconnaisse le motif de l'exception levée. Dans ce cas, le comportement exceptionnel t est suivi.
Cela fonctionne parfaitement pour les programmes fonctionnels. Cependant, en BSML, des problèmes apparaissent si des exceptions sont levées pendant une évaluation parallèle. L'exemple ci-dessous le montre :

```
let f = function
  | 0 →raise Failure "0"
  | x →fun _ →Some x in
let v = mkpar f in put v
```

Évaluation au processeur 0	Évaluation au processeur 1
let v = <**raise** Failure "0", ...> **in put** v	**let** v = <..., **fun** _ →Some 1, ...> **in put** v
*** Exception levée ***	**put** : essaie d'envoyer «Some 1» à 0

Dans cet exemple, l'exception levée localement par le processeur 0 n'est pas détectée par le processeur 1 qui continue à suivre le flux d'exécution normal. Jusqu'au moment où il atteint la barrière de synchronisation (**put** v). Là, il attend le processeur 0 qui ne le rejoindra jamais : c'est un interblocage.

Pire, si on avait englobé cet extrait de code dans un **try**...**with** Failure $\rightarrow$..., les processeurs 0 et 1 auraient pris des embranchements d'exécution (globale) différents, l'embranchement normal et l'embranchement de traitement d'exception, ce qui aurait entraîné un état d'exécution incohérent par rapport au modèle BSP.

Notre solution restera aussi familière que possible au programmeur Objective Caml séquentiel. Plutôt que de redéfinir un système de gestion d'exceptions spécifique, nous avons pu étendre le système existant pour qu'il gère les cas particuliers au parallélisme. La syntaxe utilisée pour définir et lever les exceptions est donc la syntaxe habituelle. Trois cas différents sont susceptibles d'advenir quand une exception Objective Caml est levée en BSML :

– Pendant une phase de calcul parallèle : un processeur lève localement une exception mais la rattrape avant la fin de la section locale. Dans ce cas, aucune opération globale ni aucune communication ne sont affectées ; la fonction parallèle renvoie un résultat comme prévu. Ce cas est légitime et n'affecte pas BSML.
– Pendant une section globale : tous les processeurs lèvent la même exception en même temps. Aucune incohérence n'apparaît, tous les processeurs suivront le même chemin d'exécution et rattraperont l'exception ou produiront la même erreur. Comme pour le premier cas, il n'y a pas d'interférence entre le parallélisme et l'exception.
– Pendant une phase de calcul parallèle : un processeur lève une exception localement mais ne la rattrape pas à temps. C'est le cas de l'exemple ci-dessus. Le processeur fautif va omettre d'exécuter le code global et mettre le système dans un état incohérent. Il y a des chances qu'il n'arrive même pas à la barrière de synchronisation comme prévu et cause un interblocage.

4. Un mécanisme d'exceptions pour BSML

4.1. Syntaxe

On constate que la pièce manquante pour avoir un système d'exceptions cohérent en parallèle est un outil permettant de rattraper globalement les exceptions locales, avant qu'elles n'entraînent des divergences d'exécution sur le processeur où elles ont été levées. Les deux principaux points à prendre en compte pour la conception de ce système sont :

– Éviter la divergence. En particulier, une exception locale ne doit jamais empêcher l'exécution de code global, sans quoi la cohérence de tout le système est mise en cause.
– Effectuer un traitement approprié sur l'exception. En particulier, elle ne peut pas être ignorée.

Pour éviter la divergence sans ignorer l'exception, il faut que celle-ci déclenche une décision globale. Cela ne pourra être fait avant la fin de la super-étape, puisque l'exception n'aura pas encore pu être communiquée. Après, il serait trop tard car le processeur incriminé n'aura pas pu communiquer les informations aux autres comme prévu. Il faut donc globaliser l'exception exactement à la fin de la super-étape en cours. Pourtant, cette "exception globalisée" reste un objet différent d'une exception standard, ne serait-ce que parce que plusieurs exceptions pourraient avoir été levées par différents processeurs. Des structures syntaxiques spécifiques sont donc proposées pour récupérer ces exceptions globalisées.

L'exemple de la figure suivante 2 en montre l'utilisation. La définition et le déclenchement sont faits de la façon habituelle, avec les mots-clés **exception** et **raise**.

L'exécution de f en parallèle lève une exception locale au processeur 0. Cette exception est ensuite récupérée au niveau global, évitant l'échec local de ce processeur, grâce à la structure **trypar**...**withpar**. Cette structure, qui permet de récupérer au niveau global des exceptions locales, est très similaire au **try**...**with** habituel, sauf au niveau du filtrage de motifs. En effet, on a potentiellement affaire à un ensemble d'exceptions, et le nom eset sera donc simplement lié à l'ensemble des exceptions levées avant le **withpar**. Dans cet exemple, le code de traitement d'exception donné après la flèche itère sur cet ensemble et affiche les exceptions qu'il contient sur la sortie standard. Le résultat de l'exécution sera donc : Failure "0"

```
trypar
  let f = function
    | 0 → raise Failure "0"
    | x → x
  in ignore (mkpar f)
withpar eset → Exception_set.iter (fun e → prerr_endline Printexc.to_string e.exc) eset
```

FIG. 2 – Exceptions BSML

4.2. Principe de fonctionnement

Tout n'est pas toujours aussi simple, car du code local et du code global peuvent être juxtaposés dans la même super-étape. Il faut donc s'écarter des méthodes de gestion d'exception habituelles pour assurer la non-divergence entre le moment où l'exception est levée et le moment où elle pourra être globalisée (à la fin de la super-étape). Concrètement, le processeur qui a subi une exception locale doit continuer d'exécuter les instructions globales sans tenir compte de son état local, où il y aurait divergence. Mais par ailleurs, s'il tente de continuer sa séquence d'exécution normale, il est susceptible de rencontrer des calculs impliquant des valeurs locales qu'il n'a pas calculées, calculs qu'il serait donc dans l'impossibilité d'effectuer. Le processeur dans cet état, en attendant de pouvoir communiquer sa détresse, doit donc effectuer les opérations globales normalement mais ignorer les opérations locales. Par chance, la construction de BSML rend cela tout à fait réalisable.

À la fin de la super-étape, l'état d'exception est communiqué à tous, pour permettre un traitement global approprié. Les résultats des calculs locaux de cette super-étape, par contre, seront partiels ou inexistants. L'approche la plus évidente est de suivre le même comportement qu'en séquentiel et d'abandonner tout résultat intermédiaire pour adopter le traitement d'exception prévu. Pourtant, lors des calculs parallèles il arrive qu'on veuille conserver les résultats restants : après tout, les processeurs qui n'ont pas levé d'exception peuvent avoir des résultats valides qui ont coûté du temps de calcul. L'approche que nous avons choisie pour le moment est la première, par souci de simplicité, mais nous travaillons sur un moyen de récupérer des résultats partiels.

Nous avons mentionné que les exceptions étaient globalisées à la fin de la super-étape. Pourtant, il est peu souhaitable qu'une exception levée par un processeur à l'intérieur d'un bloc **trypar**...**withpar** puisse s'échapper de ce bloc si jamais la super-étape n'est pas finie. Pour cette raison, **withpar** impose une barrière, assurant qu'aucune exception locale ne s'échappera de son domaine.

5. Implantation

5.1. Modification des primitives

Les exceptions sont une structure de contrôle très différente des boucles, conditionnelles et appels de fonctions habituels, et elles permettent de sortir de la séquence d'exécution normale du programme. Elles ne peuvent donc théoriquement être implantées qu'au niveau du compilateur. Cependant, les exceptions d'Objective Caml sont une base suffisamment puissante – et nos exceptions en sont suffisamment proches – pour qu'on puisse construire dessus notre propre système d'exceptions parallèles, sans avoir besoin de contrôler l'exécution à bas niveau.

Pour maintenir la cohérence de l'exécution, il est souhaitable que les exceptions locales soient dans un premier temps invisibles depuis l'exécution globale. Grâce à la stricte distinction entre exécution locale et globale en BSML, il nous suffit pour cela de protéger l'exécution locale en ajoutant une sorte de filet de sûreté dans les primitives **mkpar**, **apply**, **put** et **proj**. Le premier rôle de ce filet est de rattraper toute exception avant qu'elle ne s'échappe de l'exécution locale, à l'aide d'un **try**...**with**. Ces exceptions ne pouvant par contre pas être ignorées dans de nouvelles opérations locales, elles sont mémorisées par le processeur concerné dans une variable status: (Fine | Stopped **of** int, exn) **ref**. Le second rôle du filet est alors d'interdire toute nouvelle opération locale sur ce processeur, en attendant la fin de la super-étape. Cette procédure assure que si certains résultats sont absents (à cause d'une exception), il ne pourra être fait aucune tentative pour y accéder.

À la fin de la super-étape BSML, provoquée par une primitive **put**, **proj** ou un **withpar**, la phase de communication est initiée par un échange des tailles des données qui vont être communiquées. On profite de cet échange pour transmettre les états respectifs des processeurs : en cas d'exception sur l'un ou plusieurs d'entre eux, la communication normale des résultats n'a pas lieu et est implicitement remplacée par un échange total des exceptions et de leurs paramètres. De cette façon, tous les processeurs se retrouvent dans un état cohérent où le même ensemble d'exceptions est levé. Cet ensemble est propagé en utilisant les exceptions d'Objective Caml : on définit de façon interne à BSML l'exception Global_exn **of** Exception_set.t, qui sera alors levée de façon globale.

5.2. Traitement des exceptions

Les nouveaux mots-clés **trypar** et **withpar**, ainsi que quelques modifications annexes, ont été implantés à l'aide du pré-processeur générique pour Objective Caml, Camlp4. Les ensembles d'exceptions étant propagés par l'intermédiaire d'exceptions Objective Caml, le pré processeur remplace cette structure par un **try**...**with** standard, qui précède le **with** d'une barrière.
Plusieurs autres problèmes doivent être pris en considération. En particulier, il faut s'assurer que les exceptions locales sont récupérées au niveau d'imbrication de **trypar**...**withpar** auquel elles ont été levées, et non à celui auquel intervient la fin de la super-étape, qui peut être différent. Un indice de Bruijn indiquant le niveau de l'exception levée par rapport à l'exécution courante, incrémenté à chaque **trypar**, est utilisé pour lever à nouveau l'exception globalisée si elle n'appartient pas au niveau d'imbrication où elle est rattrapée dans un premier temps. D'autre part, des conflits peuvent intervenir entre exceptions. Des ajouts syntaxiques permettent de s'assurer que les exceptions globalisées ne sont pas rattrapées par l'utilisateur autrement qu'avec un **withpar**, d'une part, et d'autre part qu'une exception globale ne permet la sortie d'un bloc **trypar**...**withpar** que s'il n'y a aucune exception locale.

6. Expériences

Nous avons choisi comme exemple un algorithme de recherche en profondeur d'abord, ou *backtracking*, parce qu'il se prête très bien à être implanté à l'aide d'exceptions. Pour valider notre implantation, nous présentons donc dans cette partie une implantation naïve de *backtracking* parallèle. Il est important de noter que la façon dont le *backtracking* est implanté importe peu : la méthode employée s'appliquerait de la même façon si nous avions employé des mécanismes de répartition de la tâche complexes, et les exceptions y seraient de la même utilité.
Le *backtracking* consiste à rechercher une solution dans un arbre en choisissant systématiquement le premier fils qui n'a pas encore été parcouru, et en remontant si le nœud courant n'a pas de fils. Si l'on dispose d'une fonction récursive qui effectue cette recherche, lui faire lever une exception dès que la solution est trouvée permet d'éviter toute disjonction de cas en fonction des résultats qui lui sont renvoyés, et d'extraire directement la solution. La parallélisation de ce procédé se fait en explorant plusieurs nœuds à la fois, ce qui complique encore la récupération de solution.
Notre implantation du *backtracking* prend en paramètre une fonction séquentielle qui renvoie la liste des fils d'un nœud donné de l'arbre, ou lève une exception si une solution est trouvée. Appliquer cette fonction récursivement sur chacun des éléments de la liste qu'elle renvoie reviendrait à une implantation séquentielle. En BSML, nous procédons en trois étapes :

1. La liste actuelle de nœuds est distribuée en un vecteur parallèle
2. La fonction séquentielle est exécutée sur une partie de l'élément de ce vecteur présent sur chaque processeur
3. Tous les enfants ainsi obtenus sont rendus globaux, et on reprend la recherche sur cette liste.

Pour remonter dans l'arbre, on revient à l'étape 2 pour parcourir les nœuds laissés en suspens si l'étape 3 n'a pas donné de résultat. S'il n'y a plus de nœuds à l'étape 2, la branche concernée ne contenait pas de résultat, la fonction termine et laisse la main à son appelante pour explorer les autres branches. Il est évident que cet algorithme n'est pas optimal, en particulier au niveau des communications, mais ce qu'il est intéressant de noter est qu'il suffit de l'enclore dans un **trypar**...**withpar** pour récupérer la solution dès qu'elle sera trouvée.
Victimes de la mode, nous avons testé ce module en implantant un solveur de sudoku simpliste. Le jeu de sudoku de dimension n^2 consiste à remplir une grille de taille $n^2 \times n^2$ d'entiers de 1 à n^2 en

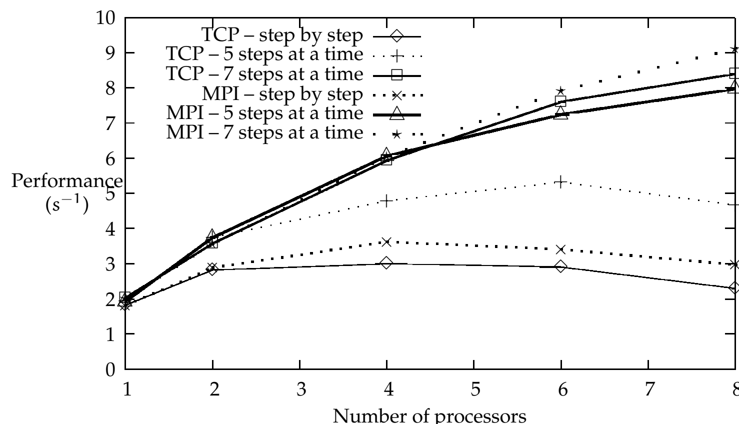

FIG. 3 – Performances pour la solution d'une grille de sudoku de dimension 9

respectant certaines contraintes. Les nombres fournis sur la grille de départ et les contraintes assurent l'unicité de la solution. Notre propos n'est pas de nous soucier des spécificités mathématiques de ces contraintes pour accélérer la solution.

Les nœuds de l'arbre de *backtracking* sont des grilles de sudoku. Leurs enfants sont obtenus en essayant tous les nombres de 1 à n^2 dans la première case libre et en filtrant selon les contraintes mathématiques. Une optimisation raisonnable consiste à composer la fonction calculant les descendants plusieurs fois pour en obtenir un nombre permettant une distribution équitable, et pour augmenter la taille des super-étapes.

La figure 3 montre, en s^{-1}, l'accélération obtenue lors de la résolution d'une grille de dimension 9 fixe sur un nombre de processeurs variant entre 2, 4, 6 et 8, avec différents niveaux d'optimisation. Cette expérience a été conduite sur une grappe de PC Pentium IV cadencé à 2,8 GHz liés par un réseau Gigabit Ethernet, en code natif compilé ; les valeurs utilisées sont la médiane d'un échantillon de tests conséquent. Comme le modèle BSP permet de le prévoir, on obtient une accélération raisonnable si on fait plusieurs niveaux de descente dans l'arbre dans une super-étape.

7. Conclusions et perspectives

Nous avons introduit un mécanisme sûr pour la gestion des exceptions dans le langage parallèle fonctionnel Bulk Synchronous Parallel ML. Pour ce faire nous avons défini la notion d'ensemble d'exceptions simultanées d'un point de vue logique – c'est-à-dire des exceptions lancées localement à la même super-étape – avec de nouvelles constructions dont l'objectif est de rattraper au niveau parallèle (par tous les processeurs) ces exceptions levées localement par un ou plusieurs processeurs. Une sémantique formelle de cette extension a été conçue [8]. Elle permet de prouver que BSML avec exceptions a les mêmes propriétés que BSML sans exception : l'absence d'interblocages et d'indétermisme.

Comme nous l'avons fait pour BSML, plusieurs autres sémantiques vont suivre. Celle qui existe actuellement présente le modèle de programmation de BSML avec exceptions. Comme c'est classique en sémantique du parallélisme de données [4], nous concevrons une sémantique formelle du modèle d'exécution et prouverons l'équivalence de ces deux sémantiques. Pour BSML nous allons également jusqu'au niveau de la description formelle de machines virtuelles parallèles qui seront également montrées correctes par rapport à la sémantique du modèle d'exécution. Au niveau de l'implantation, le mécanisme de gestion des exceptions sera utilisé dans d'autres applications.

Bibliographie

1. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
2. R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.

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. L. Bougé. Le modèle de programmation à parallélisme de données : une perspective sémantique. *RAIRO Technique et Science Informatiques*, 12(5), 1993.
5. M. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation*. MIT Press, 1989.
6. F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3) :365–376, 2003.
7. F. Gava. Implementation of Parallel Data Structures in BSML. In F. Loulergue and A. Tiskin, editors, *Third International Workshop on High-Level Parallel Programming and Applications (HLPP 2005)*, pages 161–174, June 2005.
8. L. Gesbert and F. Loulergue. Semantics of bulk synchronous parallel ml with exceptions. In Zoltán Horváth, editor, *Draft proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06)*. to appear, 2006.
9. S. Gorlatch. Message passing without send-receive. *Future Generation Computer Systems*, 18(6) :797–805, 2002.
10. J.M.D. Hill, W.F. McColl, and al. BSPlib : The BSP Programming Library. *Parallel Computing*, 24 :1947–1980, 1998.
11. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.09, 2005. web pages at www.ocaml.org.
12. F. Loulergue. BSML : Programmation BSP purement fonctionnelle. In D. Méry and G.-R. Perrin, editors, *Dixièmes Rencontres Francophones du Parallélisme (Renpar'10)*, pages 243–246, Strasbourg, june 1998.
13. F. Loulergue. A Calculus of Functional BSP Programs with Projection. In *International Parallel & Distributed Processing Symposium, 8th Workshop on Advances in Parallel and Distributed Computational Models*. IEEE Computer Society Press, 2006.
14. F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML : Modular Implementation and Performance Prediction. In Vaidy S. Sunderam, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science, Part II*, number 3515 in LNCS, pages 1046–1054. Springer, 2005.
15. F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3) :253–277, 2000.
16. B. Di Martino, A. Mazzeo, M. Mazzocca, and U. Villano. Parallel program analysis and restructuring by detection of point-to-point interaction patterns and their transformation into collective communication constructs. *Science of Computer Programming*, 40(2–3) :235–263, 2001.
17. W. F. McColl. Universal computing. In L. Bouge and al., editors, *Proc. Euro-Par '96*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.
18. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3) :249–274, 1997.
19. J. L. Traeff and J. Worringen. Verifying Collective MPI Calls. In *Proceedings of the 11th EuroPVM/MPI conference*, LNCS. Springer, 2004.
20. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103–111, August 1990.