Extension de VSL+ au multithreading

Matthieu Rodet, Manon Sourisseau

Décembre 2021

1 Introduction

La grande majorité des processeurs actuels possèdent plusieurs cœurs, il est alors nécessaire, pour tirer la plus grande puissance de calcul possible, de diviser un même processus en plusieurs sous-processus indépendants, que l'on appelle threads. Le multithreading est alors l'exécution d'un processus sur plusieurs threads. L'enjeu principal de cette programmation est le gain de temps considérable que cela peut représenter. Une même action peut voir son temps diviser par un facteur aussi grand que la machine possède de cœurs.

Cette problématique de multithreading a pris son essor à l'apparition des multiprocesseurs (à plusieurs cœurs par exemple) pour ensuite devenir un domaine de recherche très actif et très demandé. Nous avons eu l'occasion de nous y confronter dans le module Système d'Exploitation Linux ce semestre, avec la bibliothèque `pthread` du langage `C`, et avons par conséquent décidé de porter notre extension sur l'ajout au langage VSL+ des threads.

Notre travail s'est donc voué à la création de nouvelles instructions du langage, permettant la création de thread, avec la possibilité de faire des jointures.

De plus, nous proposons une nouvelle instruction de `fork` sur les tableaux, de manière multithreadé, s'inspirant du modèle MapReduce [1].

Enfin, nous avons également implémenté la compilation de ces nouvelles instructions vers le langage `LLVM`.

Nous détaillons dans ce rapport comment sont définies ces nouvelles instructions, ainsi que leurs détails de compilation.

2 Contribution

2.1 Refonte des fonctions

Initialement, les paramètres étaient considérés comme des expressions qu'on évaluait puis qu'on passait via l'instruction LLVM `call`, nous avions fait au plus simple compte tenu des consignes. Cependant, les types de chaque fonction dépendaient alors de ses arguments et de sa valeur de retour. Or l'utilisation des fonctions de multithreading nous imposait une uniformisation du typage de nos fonctions compilées.

Une première idée avait été de créer un nouveau mot clef `ROUTINE`, avec un usage similaire aux `FUNC` ou `PROTO`. Néanmoins cette approche limitait les fonctions dites routines à des fonctions sans argument et de type de retour `VOID`, ce qui nous semblait pas très intéressant. Nous avons donc abandonné cette idée et pris pour parti de retravailler entièrement le passage des paramètres des fonctions de manière plus complexe pour uniformiser leurs types et ainsi correspondre au types requis par les fonctions de création de thread.

À présent, le type de toute fonction compilée est `i8* (i8*)`, ce qui correspond en langage `C` à une fonction ayant en unique paramètre un pointeur du type `void*` et retournant également une valeur du type `void*`. On génère en réalité, en parallèle de chaque fonction, une structure contenant tous ses arguments. On crée alors, à chaque appel de fonction, la structure adéquate que l'on remplit avec l'évaluation des paramètres. On utilise ensuite l'opération `bitcast` pour transformer le pointeur de la structure en un pointeur `void*`. Le début de chaque fonction est alors consacré à l'extraction des paramètres contenus dans la structure passée en argument. De même, lors du renvoie de la valeur de retour entière d'une fonction, on utilise `inttoptr` pour convertir l'entier renvoyé en un pointeur `void*`. Cette opération pouvant paraître très dangereuse, ne l'est en réalité aucunement puisque ce pointeur est totalement opaque pour l'utilisateur, il ne pourra jamais

le manipuler. Lors de l'utilisation d'un appel de fonction dans une expression, on utilise enfin un `ptrtoint` pour convertir le pointeur renvoyé en un entier.

2.2 Création de threads et jointure

2.2.1 Langage

Pour permettre l'utilisation de thread dans VSL+, nous avons rajouter à notre langage un nouveau type, les `TID`, et deux nouvelles instructions, via les mots-clés `THREAD` et `JOIN`.

L'utilisation du type `TID` est similaire à celle des `INT`, à l'exception qu'on ne peut se servir d'un `TID` que dans les instructions `THREAD` ou `JOIN`. Les deux types ne sont en aucun cas interchangeables, un `TID` ne peut pas être un `INT` et vice-versa.

La première instruction `THREAD` correspond à l'instruction `pthread_create` du langage `C`. Nous avons décidé ce calquer l'utilisation cette fonctionnalité du langage `C` afin de facilité l'étape de la compilation vers LLVM. Ainsi, cette instruction permet de créer un nouveau thread, en lui donnant un `TID`, une fonction de routine de démarrage du thread appelée à sa création, ainsi que tous les arguments de sa routine de démarrage. Cette nouvelle instruction prend donc en paramètre, tous séparés d'une virgule comme pour les instructions `PRINT` ou `READ` :

- Un premier `TID`, correspondant à l'identifiant du thread créé.
- Un second identifiant, correspondant à la routine de démarrage.
- Une liste d'expressions, correspondant aux paramètres de la routine de démarrage.

En plus de la création de thread, nous avons rajouté la fonctionnalité `JOIN` qui nous semble être indispensable dans l'utilisation concrète des threads, puisqu'elle permet de forcer l'attente de threads entre eux. Ainsi, notre langage comprend la nouvelle instruction `JOIN`, qui là aussi s'inspire du `pthread_join`, présent dans le langage `C`. Cette nouvelle instruction prend en paramètre :

- Un `TID`, correspondant à l'identifiant du thread que l'on souhaite attendre.
- Un `INT` optionnel, dans le cas où la routine de démarrage du thread créé avec ce `TID` renvoie un entier et que l'utilisateur veut le récupérer.

```
1  FUNC INT routine(a)
2  {
3      PRINT "Hello World !"
4      RETURN a
5  }
6
7  FUNC VOID main()
8  {
9      TID t1
10     INT a
11
12     THREAD t1, routine, 1
13     JOIN t1, a
14     PRINT a
15 }
```

Ce morceau de code est un exemple simple de création de thread. Le thread `t1` est déclaré au début de la fonction `main`, puis créé à l'aide de l'instruction `THREAD`, appelant la fonction `routine`. Le processus se joint alors à ce thread, avec l'instruction `JOIN`

2.2.2 Compilation

Les `TID` sont représentés en LLVM par des entiers 64 bits `i64`, puisque les fonctions `pthread_create` et `pthread_join` requièrent un tel type. C'est pourquoi nous disions qu'ils ne sont pas interchangeables avec les `INT` classiques. Un typage aussi fort apporte également plus de clarté pour l'utilisateur.

L'instruction `THREAD` s'occupe tout d'abord d'évaluer les arguments de la routine et de construire la structure adaptée, comme le ferait un appel classique à une fonction. La différence est qu'au lieu d'utiliser l'instruction LLVM `call`, on appelle l'instruction de la bibliothèque `pthread` du langage `C`, `pthread_create`, donnant en argument le `TID`, la routine de démarrage du thread, ayant pour type `i8* (i8*)` suite à notre refonte des appels de fonction, et enfin le pointeur de la structure contenant les arguments, comme pour un appel classique de fonction. La routine étant une fonction quelconque, elle s'occupera elle-même d'extraire les arguments de la structure qui lui est passée en paramètre.

L'instruction `JOIN` utilise simplement l'instruction de la bibliothèque `pthread` du langage `C`, `pthread_join`, et convertit le résultat du thread, s'il est demandé par l'utilisateur, d'un pointeur `i8*` vers un entier à l'aide de l'instruction LLVM `ptrtoint`, comme lors d'un appel de fonction en tant qu'expression.

Le code précédent compilé donne alors :

```
1  ; Target
2  target triple = "x86_64-pc-linux-gnu"
3  ; External declaration of the printf function
4  declare i32 @printf(i8* noalias nocapture, ...)
5  declare i32 @scanf(i8* noalias nocapture, ...)
6
7  %union.pthread_attr_t = type { i64, [48 x i8] }
8  declare i32 @pthread_create(i64*, %union.pthread_attr_t*, i8* (i8*)*, i8*)
9  declare i32 @pthread_join(i64, i8**)
10
11 ; Actual code begins
12
13 @read_1 = global [3 x i8] c"%d\00"
14 %routine_args = type {i32}
15 @.fmt_4 = global [ 14 x i8 ] c"Hello World !\00"
16 %main_args = type {}
17 @.fmt_18 = global [ 3 x i8 ] c"%d\00"
18
19 define i8* @routine(i8* %_args) {
20 %tmp_2 = bitcast i8* %_args to %routine_args*
21 %a_3 = getelementptr %routine_args, %routine_args* %tmp_2, i32 0, i32 0
22 call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.fmt_4, i64 0, i64 0) )
23 %tmp_5 = load i32, i32* %a_3
24 %tmp_6 = inttoptr i32 %tmp_5 to i8*
25 ret i8* %tmp_6
26 ret i8* null
27 }
28
29 define i8* @main(i8* %_args) {
30 %tmp_7 = bitcast i8* %_args to %main_args*
31 %t1_8 = alloca i64
32 %a_9 = alloca i32
33 %tmp_10 = alloca %routine_args
34 %tmp_12 = getelementptr %routine_args, %routine_args* %tmp_10, i32 0, i32 0
35 store i32 1, i32* %tmp_12
36 %tmp_11 = bitcast %routine_args* %tmp_10 to i8*
37 call i32 @pthread_create(i64* %t1_8, %union.pthread_attr_t* null, i8* (i8*)* @routine, i8* %tmp_11)
38 %tmp_13 = load i64, i64* %t1_8
39 %tmp_14 = alloca i8*
40 call i32 @pthread_join(i64 %tmp_13, i8** %tmp_14)
41 %tmp_15 = load i8*, i8** %tmp_14
42 %tmp_16 = ptrtoint i8* %tmp_15 to i32
43 store i32 %tmp_16, i32* %a_9
44 %tmp_17 = load i32, i32* %a_9
45 call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.fmt_18, i64 0, i64 0), i32 %tmp_17 )
46 ret i8* null
47 }
```

Figure 1: Sortie du compilateur pour un exemple de programme utilisant les threads

En dehors des sorties standards de notre compilateur, notre refonte des fonctions donne les changements suivants. Les lignes 14 et 16 déclarent les structures (potentiellement vides) pour les arguments des fonctions `routine` et `main`. Les lignes 20 et 21 extraient les arguments de la fonction. La ligne 24 convertit l'entier retourner en un pointeur `void*`.

Ainsi, notre compilateur sort, pour la déclaration du `TID`, la ligne 31.

Pour l'instruction `MAP`, le compilateur génère les lignes 33 à 37. Les 4 première définissent la structure

contenant les arguments de la routine `routine` puis convertit le pointeur vers cette structure, tandis que la dernière appelle la fonction `pthread_create` avec les bon arguments. Nous notons que nous avons pas donné la possibilité de gérer les attributs des threads, cela pourrait être une extension intéressante.

Enfin, pour l'instruction `JOIN`, le compilateur génère les lignes 38 à 43. La ligne 38 récupère le `TID`, la ligne 39 alloue l'espace mémoire nécessaire à la valeur de retour de la routine, la ligne 40 appelle l'instruction `pthread_join` et enfin les lignes 41 à 43 convertissent et stockent la valeur de retour du thread.

2.2.3 Performances

Pour tester l'efficacité de nos threads, nous avons chronométré cent sessions de mille exécutions d'un programme, avec plusieurs threads puis avec un unique thread. Les programmes sont disponibles dans l'archive aux emplacements `tests/threads/test_unit_thread.vsl` et `tests/threads/test_unit_not_thread.vsl`. Les resultats sont également disponibles dans l'archive dans le dossier `stats`. Il est possible de relancer les expériences à l'aide de la commande `make experiment`, cela prendra approximativement 500 secondes.

Les résultats sont convainquant : en moyenne, le programme multithreadé s'exécute en 9.73 secondes tandis que son équivalent monothreadé s'exécute en 11.90 secondes. On a alors un gain de 18% du temps d'exécution sur un court exemple n'ayant que trois threads, laissant présager des bénéfices d'autant plus grands que l'on pourrait augmenter le nombre de threads et de cœurs de calcul.

2.3 *MapReduce*, un outil multithreadé

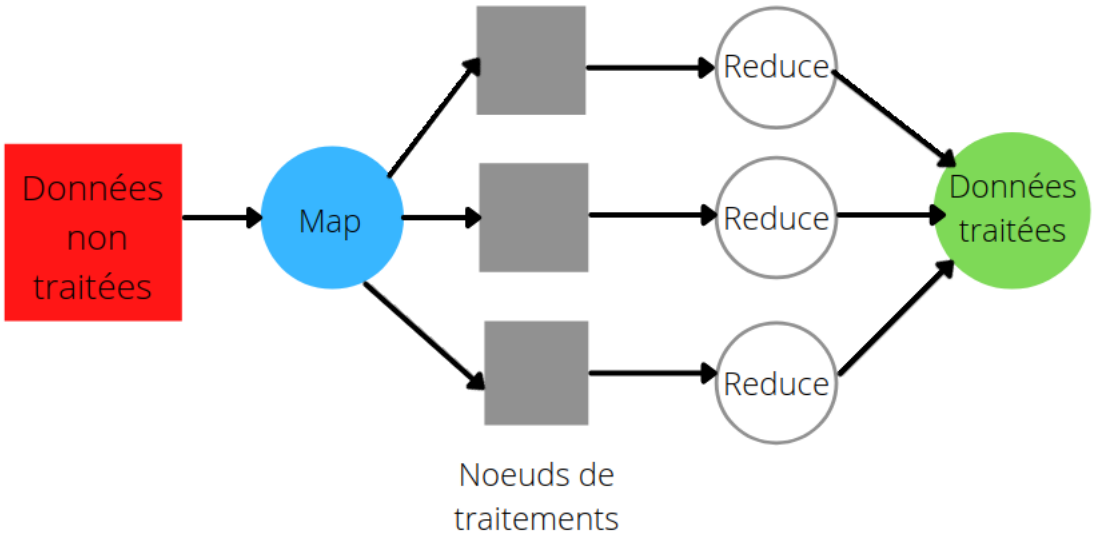
2.3.1 Interêt

Inventé par Google et déposé en 2010, le *MapReduce*[1] est un modèle de développement pour le calcul parallèle (souvent distribué), principalement utilisé pour le traitement d'un nombre important de données. Il consiste en la division d'une large source d'information en plusieurs sous-blocs d'informations traités en parallèle avant d'être à nouveau rassemblés.

Le *MapReduce* fonctionne en deux parties :

La première (*Map*), s'occupe de traiter un bloc de données, en distribuant ces données vers des noeuds de traitement choisis.

Le seconde partie (*Reduce*), s'occupe de récupérer tous les blocs de données traitées, et de les ré-assembler pour en faire des données traitées cohérentes.



L'intérêt de ce modèle générique, est le traitement parallèle des données, qui permet entre autre, une optimisation du temps de calcul. En pratique, de part la taille des données traitées, plusieurs *MapReduce* évoluent en parallèle sur des blocs de sous-données.

2.3.2 Implémentation

Notre extension de VSL+ offre un outil de traitement de tableau s'inspirant du MapReduce.

La nouvelle instruction `MAP` permet d'effectuer un calcul sur tous les élément d'un tableau, de manière multithreadé : Le tableau est découpé en bloc, et pour chaque bloc, un thread est créé et va effectuer une fonction de calcul sur chaque élément du tableau.

L'instruction `MAP` prends donc en paramètre :

- L'identifiant du tableau traité
- Un premier entier, donnant le nombre de threads à créer (égal au nombre de nombre de subdivision du tableau)
- Un second entier, correspondant à la taille du tableau
- La routine de calcul
- Une liste optionnelle d'expressions, correspondant aux arguments de la routine de calcul

```
1  FUNC VOID routine(a[], b, c)
2  {
3      WHILE b DO {
4          b := b-1
5          a[b] := 2*b + c
6      } DONE
7  }
8
9  FUNC VOID main()
10 {
11     INT t[1000]
12
13     MAP t, 2, 1000, routine, 10
14 }
```

Ce morceau de code est un exemple de l'utilisation de l'instruction `MAP`. Ici, le tableau `t` est initialisé via un `MAP`, où 10 threads sont créées. La fonction de calcul `routine` est la fonction de calcul du `MAP`, qui va se charger d'initialiser le tableau.

2.3.3 Compilation

La compilation de l'instruction `MAP` utilise tous les outils de multithreading implémentés précédemment. Le principe est alors relativement direct : On évalue les arguments de la routine puis on divise le tableau en `n` parties, pour chaque partie on crée un nouveau thread et on le lance. Enfin, on réalise la jointure des `n` threads ainsi créés.

Le résultat de la compilation de l'exemple précédent est donné en annexe. L'instruction `MAP` sépare le tableau en deux, comme demandé par l'utilisateur. On remarque dans le code compilé la répétition du code entre les lignes 47 à 57 et 58 à 68, avec comme seule différence les lignes 50 et 61, la première prenant un pointeur vers l'indice 0 du tableau et la seconde prenant un pointeur vers la case d'indice 500 (la taille du tableau étant 1000). Le programme compilé traite donc bien le tableau dans deux threads distincts après l'avoir séparé en deux sous-tableaux, l'un allant de l'indice 0 et de taille 500 (donc jusqu'à l'indice 499) et l'autre allant de l'indice 500 et de taille 500 (donc jusqu'à la fin du tableau). On a également généré un `TID` par thread créé, et les lignes 69 à 72 s'assurent de la jointure de tous les threads.

2.3.4 Perfomances

Pour tester l'efficacité de notre *MapReduce*, nous avons chronométré cent sessions de mille exécutions d'un programme, en utilisant l'instruction `MAP` avec dix sous-tableaux puis sans l'utiliser (en ne créant en réalité qu'un sous tableau). Les programmes sont également disponibles dans l'archive aux emplacements `tests/threads/test_unit_map_hard.vsl` et `tests/threads/test_unit_not_map_hard.vsl`. Les résultats

sont également disponibles dans l'archive dans le dossier `stats`. Il est possible de relancer les expériences à l'aide de la commande `make experiment`, cela prendra approximativement 500 secondes.

Les résultats sont, tout comme ceux des threads, convainquant : en moyenne, le programme utilisant `MAP`, et divisant le tableau en dix, s'exécute mille fois en 6.47 secondes tandis que son équivalent avec un seul sous tableau s'exécute mille fois en 10.74 secondes. On a alors un gain de 40% du temps d'exécution sur un court exemple.

3 Conclusion

Notre travail s'est porté sur l'ajout au langage VSL+ de la création de thread appelant des fonctions de démarrage, avec leur jointure. Ces threads doivent être déclarés à l'avance, avant leur création. De plus, nous proposons une nouvelle instruction de `fork` sur les tableaux, de manière multithreadé, s'inspirant du MapReduce.

Cette extension aurait peut-être dû être implémentée après d'autres, telles que les pointeurs ou la programmation orientée objets. Cela aurait considérablement facilité le passage des arguments aux threads sans avoir à manipuler les structures ni revoir le passage des arguments des fonctions.

Cependant, nos résultats sont très bons et prouvent d'une parfaite réalisation du contrat initial. Nous sommes même allé au delà de ce qui était initialement prévu. En effet, à l'origine, l'extension proposée ne projetait pas de pouvoir donner n'importe quel argument aux routines des threads ou de l'instruction `MAP`. La refonte du passage des arguments des fonctions a été un enjeu considérable nous demandant de revoir entièrement notre vision des fonctions. Nous sommes pleinement satisfait du résultat, puisque nous avons réussi à gagner 40% de temps d'exécution avec l'utilisation du *MapReduce*, avec un exemple simple.

Bibliographie

[1] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing, January 19 2010. US Patent 7,650,331.

Annexe

```
; Target
target triple = "x86_64-pc-linux-gnu"
; External declaration of the printf function
declare i32 @printf(i8* noalias nocapture, ...)
declare i32 @scanf(i8* noalias nocapture, ...)

%union.pthread_attr_t = type { i64, [48 x i8] }
declare i32 @pthread_create(i64*, %union.pthread_attr_t*, i8* (i8*)*, i8*)
declare i32 @pthread_join(i64, i8**)

; Actual code begins

@read_1 = global [3 x i8] c"%d\00"
%routine_args = type {i32*, i32, i32}
%main_args = type {}

define i8* @routine(i8* %_args) {
%tmp_2 = bitcast i8* %_args to %routine_args*
%tmp_6 = getelementptr %routine_args, %routine_args* %tmp_2, i32 0, i32 0
%a_3 = load i32*, i32** %tmp_6
%b_4 = getelementptr %routine_args, %routine_args* %tmp_2, i32 0, i32 1
%c_5 = getelementptr %routine_args, %routine_args* %tmp_2, i32 0, i32 2
br label %While_17
While_17:
%tmp_7 = load i32, i32* %b_4
%tmp_8 = icmp ne i32 %tmp_7, 0
br i1 %tmp_8, label %Bloc_18, label %EndWhile_19
Bloc_18:
%tmp_9 = load i32, i32* %b_4
%tmp_10 = sub i32 %tmp_9, 1
store i32 %tmp_10, i32* %b_4
%tmp_11 = load i32, i32* %b_4
%tmp_12 = load i32, i32* %b_4
%tmp_13 = mul i32 2, %tmp_12
%tmp_14 = load i32, i32* %c_5
%tmp_15 = add i32 %tmp_13, %tmp_14
%tmp_16 = getelementptr i32, i32* %a_3, i32 %tmp_11
store i32 %tmp_15, i32* %tmp_16
br label %While_17
EndWhile_19:
ret i8* null
}

define i8* @main(i8* %_args) {
%tmp_20 = bitcast i8* %_args to %main_args*
%t_21 = alloca [1000 x i32 ]
%tmp_23 = alloca %routine_args
%tmp_24 = getelementptr %routine_args, %routine_args* %tmp_23, i32 0, i32 0
%tmp_25 = getelementptr %routine_args, %routine_args* %tmp_23, i32 0, i32 1
%tmp_26 = getelementptr [1000 x i32], [1000 x i32]* %t_21, i32 0, i32 0
store i32* %tmp_26, i32** %tmp_24
store i32 500, i32* %tmp_25
%tmp_28 = getelementptr %routine_args, %routine_args* %tmp_23, i32 0, i32 2
store i32 10, i32* %tmp_28
%tmp_27 = bitcast %routine_args* %tmp_23 to i8*
%tmp_22 = alloca i64
call i32 @pthread_create(i64* %tmp_22, %union.pthread_attr_t* null, i8* (i8*)* @routine, i8* %tmp_27)
%tmp_30 = alloca %routine_args
%tmp_31 = getelementptr %routine_args, %routine_args* %tmp_30, i32 0, i32 0
%tmp_32 = getelementptr %routine_args, %routine_args* %tmp_30, i32 0, i32 1
%tmp_33 = getelementptr [1000 x i32], [1000 x i32]* %t_21, i32 0, i32 500
store i32* %tmp_33, i32** %tmp_31
store i32 500, i32* %tmp_32
%tmp_35 = getelementptr %routine_args, %routine_args* %tmp_30, i32 0, i32 2
store i32 10, i32* %tmp_35
%tmp_34 = bitcast %routine_args* %tmp_30 to i8*
%tmp_29 = alloca i64
call i32 @pthread_create(i64* %tmp_29, %union.pthread_attr_t* null, i8* (i8*)* @routine, i8* %tmp_34)
%tmp_37 = load i64, i64* %tmp_22
call i32 @pthread_join(i64 %tmp_37, i8** null)
%tmp_36 = load i64, i64* %tmp_29
call i32 @pthread_join(i64 %tmp_36, i8** null)
ret i8* null
}
```

Figure 2: Sortie du compilateur pour un exemple de programme utilisant l'instuction `MAP`