XFOR : xfor par macros

Dieckhoff Vincent

Mars 2014

1 Introduction

Devant l'évolution du nombre de coeurs physiques dans les processeurs actuels, et du fait de la limitation de l'augmentation de la fréquence, il est nécessaire de modifier notre façon de programmer, en passant d'un modèle de programmation séquentielle à un modèle de programmation, au maximum, parallèle. Dans un soucis d'améliorer les performances du principal consommateur de temps de calcul, à savoir les boucles, xfor propose de modifier et fusionner les différents domaines d'itérations de plusieurs nids de boucles afin d'augmenter la localité de leur données et donc augmenter les performances.

1.1 Problématique

xfor étant une nouvelle structure de contrôle, son utilisation implique que les compilateurs soient en mesure de générer du code basé sur cette syntaxe. Cette fonctionnalité n'étant pas encore intégrée dans les compilateurs usuels, il est intéressant de permettre son utilisation en n'en émulant le comportement au travers de macros. Le but étant de se rapprocher le plus possible de la syntaxe originelle du xfor afin de permettre une transition simple vers xfor une fois les compilateurs compatibles, et ce,tout en permettant l'utilisation des raffinements que sont le *grain* et l'*offset*. On distingue ici, la profondeur N (i.e. le nombre de for imbriqués) et K le nombre de nids de boucles que l'on souhaite fusionner.

2 Principe de l'implémentation de XFOR

Afin de pouvoir simuler le fonctionnement d'xfor, il faut se ramener à un domaine d'itération similaire. Pour ce faire, il faut tout d'abord créer tout les domaines d'itération de chaque nid de boucles tout en prenant soin de décaler chaque dimension par l'offset donné par l'utilisateur, et de dilater par le facteur grain. Puis de fusionner ces domaines d'itérations de manière à trier les valeurs (afin de pouvoir exécuter dans le bon ordre), pour enfin

pouvoir itérer dessus, et donc exécuter le code lié à chaque nid de boucle dans l'ordre dans lequel il l'aurait été avec un xfor. Tout ceci doit être fait à l'exécution afin de permettre des domaines affines. Les macros servant uniquement à créer la structure permettant ces calculs.

2.1 Description du xfor

xfor est une structure de contrôle exprimant la fusion entre différents nids de boucles for et permettant d'ajuster finement les fréquences d'itérations des boucles grâce à l'utilisation des paramètres de grain et d'offset. Ex :

```
xfor(i0=1,i1=2;i0<5,i1<4;i0++,i1++:2,1:1,0)
    xfor(j0=i0,j1=1;j0<5-i0,j1<i1+1;j0++,j1++:1,2:0,1)
```

qui équivaut à

```
for(i0=1;i0<5;i0++)                for(i1=2;i1<4;i1++)
    for(j0=i0;j0<5-i0;j0++)                for(j1=1;j1<i1+1;j1++)
```

Les paramètres de grains et d'offset offrent la possibilité d'ajuster le domaine d'itération des nids de boucles, respectivement en dilatant le domaine d'itération et en retardant la première itération (i.e. par rapport au domaine de référence).

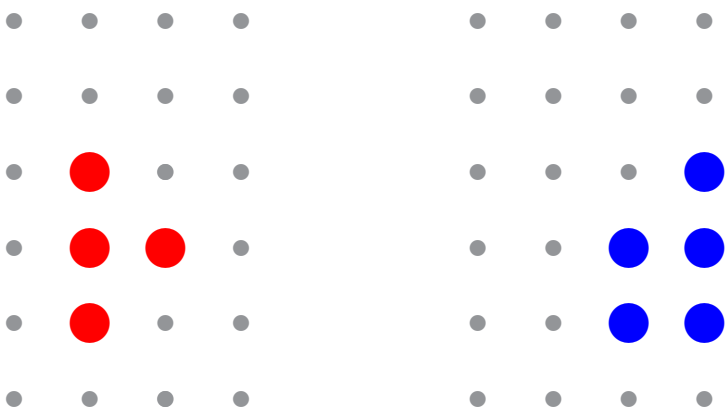

Figure 1: Domaines d'indices

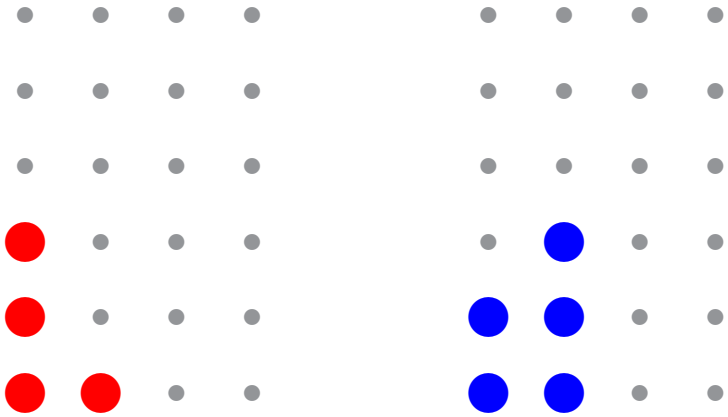

Figure 2: Domaines d'itérations

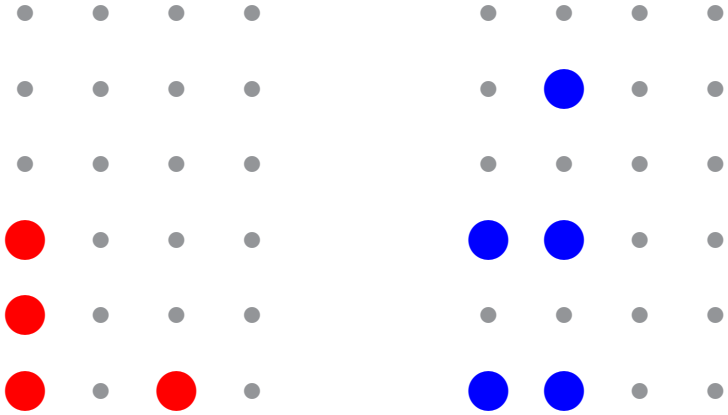

Figure 3: Application du grain

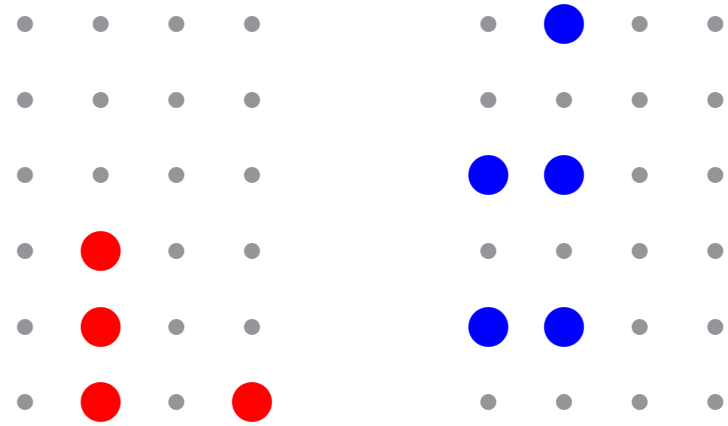

Figure 4: Application de l'offset

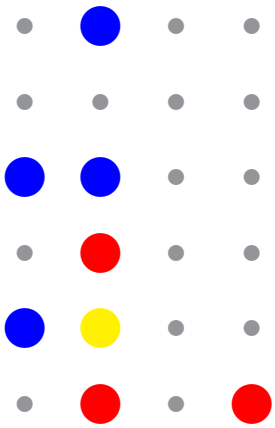

Figure 5: Domaine de référence

2.2 P99

P99 est un ensemble de macros rendant l'utilisation des nouveautés apportées par C99 plus abordables et complètes. Rajoutant aussi certains concepts comme les arguments par défaut.

2.3 P99_PASTE2(x,y)

La macro P99_PASTE2 permet de concaténer les deux arguments donnés

`Ex : P99_PASTE(int,i) donnera inti`

2.4 P99_FOR(commun,n,séparateur,fonction,liste)

La macro P99_FOR permet de répéter une action plusieurs fois et ce au moment du préprocessing.

- L'argument commun, est un argument qui sera disponible pour toutes les différentes itérations. Cela peut être un nombre ou un nom que l'on voudra utiliser à chaque fois.
- L'argument n, donne le nombre de fois que sera appliquée la macro.
- L'argument séparateur est une macro indiquant comment lier les résultats de chaque itération.
- L'argument fonction est la macro qui sera appliquée à chaque itération.
- L'argument liste, est une liste variadic, l'élément à l'index correspondant à l'itération actuelle sera disponible pour la fonction.

```
Ex : On suppose que __VA_ARGS__ contient A,B
SER(NAME,I,REC,RES) REC; RES
MAC(NAME,X,I) NAME[I]=X

P99_FOR(iter,2,SER,MAC,__VA_ARGS__)

donnera

iter[0]=A
iter[1]=B
```

2.5 Macro sur les variadics

Une fonction ou une macro est dite variadic si elle accepte un nombre non défini d'arguments. La partie variadic de la fonction est délimitée par des points de suspensions. Qui sont toujours placés après les arguments obligatoires fixes.

```
Ex : TEST(a,b,...)
```

On accède aux arguments par la macro __VA_ARGS__

2.5.1 P99_SUB(i,n,liste)

La macro P99_SUB permet de créer une nouvelle liste d'argument contenant n arguments de liste à partir du ième élément.

```
Ex : __VA_ARGS__ = A,B,C,D
P99_SUB(1,2,__VA_ARGS__)=B,C
```

2.5.2 P99_REVS(liste)

La macro P99_REVS permet d'inverser la liste des variadics.

```
Ex : __VA_ARGS__ = A,B,C
P99_REVS(__VA_ARGS__)=C,B,A
```

2.5.3 P99_SKP(n,liste)

La macro P99_SKP permet de créer une nouvelle liste d'argument commençant à l'argument K

```
Ex : __VA_ARGS__ = A,B,C
P99_SKP(1,__VA_ARGS__)=B,C
```

3 Preprocessing

La difficulté principale de ce projet, est le fait que les macros soient expendiées par le preprocesseur, et que celui-ci travaille d'une manière linéaire, en ne faisant qu'une seule passe. Cela oblige donc à penser d'une manière différente que les programmes fonctionnels habituels. Il faut aussi distinguer ce qui sera exécuté (et expendié) au préprocessing, de ce qui sera effectué à l'exécution.

4 En Pratique

4.1 Fonctionnement général

Récupération des informations nécessaires au calculs des domaines de référence (afin d'ordonner les itérations), et des domaines d'indices dans les XFOR. Récupération des informations de la dernière profondeur puis calcul des domaines et enfin itération sur le domaine.

4.2 Limitations

Afin de permettre une première implémentation, il a été necéssaire de poser quelques conditions :

- Le grain doit être positif
- Les nids de boucles sont parfaits
- L'incrément est positif et fixé à 1
- La profondeur des nids de boucles est limitée à 10 (par un define)

La positivité du grain et des incréments est nécessaire afin de garder un ordre toujours croissant dans les domaines. Ce qui simplifie la tâche lors de la fusion.

4.3 Exemple d'utilisation

```
XFOR_INIT

XFOR(0,2,i0,i1,1,2,5,4,2,1,1,0)
    XFOREND(1,2,j0,j1,i0,1,5-i0,i1+1,1,2,0,1)
    {
        XFOR_CASE(0) code
        XFOR_CASE(1) code
    }
```

4.3.1 Logique

XFOR

```
for(...){
    stockage d'informations
```

XFOREND

```
    for(...){
        stockage d'informations
        calculs d'indices
        calculs de références
        ajout dans la liste correspondante
    }
    tri fusion des listes
}
for(itération sur la liste finale)
    switch(...)
    {
```

XFORCASE(0)

```
        cas 0
```

XFORCASE(1)

```
        cas 1
```

```
        .
        .
        .
    }
```

4.4 XFOR_INIT

Afin de pouvoir exécuter notre macro xfor il est nécessaire de déclarer au préalable un certains nombre de variables qui nous permettent de stocker des informations qui seront accessible à n'importe quel macro xfor. Ces variables sont :

- deux tableaux de pointeurs sur nos itérateurs
- un tableau de listes d'itérations
- un pointeur sur la tête de la liste finale

4.5 Double exemplaire superIterator

Lors de la création des domaines d'itération, chaque boucle FOR est exécutée jusqu'à ce que les k bornes supérieures soient atteintes. Afin de déterminer

si une bornes supérieure est déjà atteinte (et donc de ne plus continuer à exécuter le code lié) on place le pointeur vers l'itérateur correspondant à NULL. Sachant que l'on a encore besoin de garder la référence à ces itérateurs par la suite pour pouvoir faire les affectation des valeurs du domaine d'indice, il est nécessaire de les stocker quelque part d'autre. On se retrouve donc avec deux tableaux superIterator, l'un pour le traitement (qui fini avec toutes ses valeurs à NULL ou presque), et un pour les affectations.

4.6 XFOR

4.7 Tableaux de pointeurs sur tableaux de pointeurs (superIterator)

Derrière cette idée barbare se trouve l'une des clés possible permettant de résoudre les deux problèmes principaux de ce projet, à savoir la communication inter-macros et le pré-processing linéaire et unique. A chaque profondeur (i.e chaque XFOR/XFOREND), est créé un tableau de pointeur vers les itérateurs données par l'utilisateur. Ceci afin de pouvoir itérer dessus suivant k.

```
int **P99_PASTE(iter,p);
```

Il est donc possible d'accéder a l'itérateur correspondant à la profondeur p, et au nid de boucle k.

```
P99_PASTE(iter,p)[k];
```

De fait si l'on veut effectuer un test sur chaque profondeur et parcourir en même temps les k nids de boucles, on utilisera la macro P99_FOR avec les fonctions adéquat pour se retrouver avec

```
P99_PASTE(iter,I)[k];
```

Et englober le tout dans un

```
for(int k=0; k<5; ++k).
```

Malheureusement, dans notre cas, il est nécessaire d'effectuer un test sur la profondeur précédente ainsi que l'actuelle tout en itérant sur K. Il n'est donc pas possible d'utiliser de macro, qui reviendrait à :

```
P99_PASTE(iter,p-1)[k];      ---->   iterp-1[k]
```

Ce qui n'a aucun sens. On stocke donc nos pointeurs vers nos tableaux de pointeurs sur les itérateurs dans un tableau de pointeur. Permettant ainsi l'accès à nos itérateurs suivant la profondeur et la "largeur" (i.e. k).

```
superIterator[p][k]
```

```
XFOR(depth,k,(k iterateurs),(k bornes inf),
(k bornes sups),(k grains),(k offsets))

Ex : XFOR(0,2,i0,i1,1,1,3,5,2,1,0,1)
```

4.7.1 Initialisation et Génération de code mort

Afin d'initialiser ce qui à besoin de l'être, on vérifie si notre XFOR est la première profondeur (i.e. 0), et on y initialise notre tableau de listes d'itérations. Ce test qui n'est utile que lors de la première profondeur provoque donc une génération de code mort pour les autres profondeurs du fait que les macros sont expandiées de manière systématique.

```
depth 0:                                depth 1+:
if(0==0) -> code executé                if(1==0) -> code mort
```

4.8 Récupération des informations

Ici on déclare deux tableaux de pointeurs, qui vont regrouper les pointeurs vers les itérateurs donnés par l'utilisateur (ici i0,i1). Chacun d'eux est initialisé grâce à P99_FOR.
Le premier n'ayant qu'un usage local il est donc alloué statiquement

```
n=3, k=2, __VA_ARGS__ = {i0, i1}

#define FUNC(NAME,X,I) &X

int *P99_PASTE2(iter,n)[k]={P99_FOR(,k,FUNC,P00_SEQ,__VA_ARGS__)};
=>
int *iter3[2]={&i0,&i1};
```

Celui-ci servira à contrôler l'exécution lors de la création du domaine d'itération. Lorsqu'une boucle dépasse son nombre d'itération on met son pointeur vers l'itérateur à NULL. De manière à savoir que celle-ci ne doit plus continuer.
Le deuxième servira quant à lui une fois sorti de nos calculs de domaines afin d'affecter les vraies valeurs des itérateurs aux bons itérateurs, et à donc

besoin d'être alloué de manière dynamique. Encore une fois, P99_FOR nous permet de remplir ce tableau.

```
    n=3, k=2, __VA_ARGS__ = {i0, i1}

    #define FUNC(NAME,X,I) NAME[I]=&X

    int **P99_PASTE2(iterr,n)=malloc(sizeof(int*)*k);
P99_FOR(P99_PASTE2(iterr,n),k,FUNC,P00_SEP,__VA_ARGS__)};
=>
int **iterr3=malloc(sizeof(int*)*k);
iterr3[0]=&i0;
iterr3[1]=&i1;
```

Il nous faut aussi stocker les valeurs de grain, d'offset ainsi que les bornes inférieures de nos itérations.

4.9 Itération sur la profondeur

Reste enfin à itérer selon les valeurs données par l'utilisateur. Toujours lié à la limitation imposée par la séquentialité il est nécessaire de parcourir les k domaines en simultané. L'idée est alors d'effectuer une boucle FOR ne terminant que lorsque toutes les bornes supérieures sont atteintes. Afin de savoir si l'on doit continuer à boucler, on vérifie que le pointeur sur l'itérateur précédent n'est pas NULL, puis si le pointeur de notre profondeur n'est lui non plus pas NULL et enfin si ce dernier itérateur n'est pas supérieure à la borne supérieure donnée. On effectue enfin un OU logique entre ces conditions pour chaque K, ce code est généré grâce à un P99_FOR.
Ex : A la profondeur 1

```
    k=2

    for(i0=1,i1=2;
        (superIterator[p-1][0]!=NULL
        && superIterator[1][0]!=NULL
        && superIterator[1][0]<bornessup1)
        ||
        (superIterator[p-1][1]!=NULL
        && superIterator[1][1]!=NULL
        && superIterator[1][1]<bornesup2);)
```

Enfin, le dernier traitement effectué consiste à vérifier si les itérations ont atteints leur borne supérieure. Si ce n'est pas le cas, on incrémente l'itérateur, sinon on le met à NULL. Test type :

```
if(superIterator[p-1][k]!=NULL
    && superIterator[p][k]!=NULL
    && P99_PASTE2(iter,p)[k]<bornesup)
    ++(*P99_PASTE2(iter,p)[k]);
else
    P99_PASTE2(iter,p)[k]=NULL;
```

4.9.1 Utilisation des variables de l'utilisateur pour les domaines

Il est absolument indispensable d'utiliser les variables données par l'utilisateur afin de calculer les domaines d'indices/d'itération, car l'on veut donner la possibilité à l'utilisateur de donner une borne inférieure et/ou supérieure dépendant d'un des itérateurs englobants. Par exemple :

```
XFOR(0,2,i0,i1,1,1,3,5,2,1,0,1)
    XFOREND(1,2,j0,j1,1,1,3-i0,5,2,3,1,0)
```

Ici on remarque très bien qu'il faut itérer sur les bons itérateurs afin de permettre le bon déroulement de nos calculs.

4.10 Fix ternaire

Lorsque l'on effectue les tests afin de savoir si notre "branche" doit être exécutée ou non, on regarde si le pointeur vers l'itérateur précédent est à NULL ou non. Pour ce faire, on effectue un p-1, le problème est donc que dans le cas d'une profondeur de 0 on se retrouve Out Of Bounds (OOB). Afin d'éviter ce phénomènes, tout les occurrences de p-1 sont remplacées par le test ternaire suivant :

```
p-1>=0?p-1:0
```

Quitte à effectuer deux fois le même test, dans le cas où l'on est à la profondeur 0.

4.11 XFOR_END

De la même manière que dans XFOR, on récupère les données entrée par l'utilisateur. Une fois entrée dans la dernière profondeur de for, il est alors possible de calculer les domaines d'itérations et d'indices car nous possédons toutes les informations nécessaires. Comme l'on parcourt tout les domaines d'itérations en même temps de part la forme même de nos boucles for imbriquées, on se retrouve avec un corps de boucle unique. De fait il faut pouvoir distinguer les points des domaines de chacun des domaines (i.e. nid

de boucle). Il faut donc parcourir chacun des k afin de déterminer si l'on doit rajouter (ou non) un point au domaine et si oui, auquel. Cela pourrait potentiellement être fait à l'aide de P99_FOR comme pour les tests de bonnes supérieures dans XFOR (cf Itération sur la profondeur), malheureusement il nous serait alors impossible de parcourir la profondeur à l'aide d'un autre P99_FOR lors de chacune des itérations de l'autre P99_FOR.
Nous sommes donc obligé de parcourir k ou p à l'exécution à l'aide d'une boucle for classique. On choisit d'itérer sur k et d'utiliser P99_FOR sur p. Pour chacun des k, on vérifie si l'itération est valide (i.e. l'itérateur ne dépasse pas la borne supérieure). Dans le cas précédent, nos tests étaient créés par P99_FOR. Ici nous n'avons plus qu'un test, qui est parcouru k fois. Les bornes supérieures doivent donc être accessibles suivant k. On créer donc un tableau dans lequel on les place afin d'y avoir un accès direct et indicé.

4.11.1 Création des points

Si tel est le cas, on incrémente l'itérateur lié, et on alloue un élément de liste que l'on ajoute ensuite à la liste k correspondante (tableau de liste k crée par XFOR_INIT et initialisé à la profondeur 0). Chaque élément de liste est composé de :

- un numéro de nid de boucle
- une tableau d'int portant les valeurs du point dans le domaine d'indices
- une tableau d'int portant les valeurs du point dans le domaine de référence
- un pointeur vers l'élément suivant

Il faut alors initialiser nos deux tableaux d'int (de taille p + 1), ainsi que mettre le pointeur vers le élément de la liste à NULL. Enfin on affecte les valeurs. Le numéro correspondant au nid de boucle actuel par notre itérateur sur k. On affecte ensuite à chacune des cases de nos deux tableaux les valeurs correspondante. Pour se faire on utilise encore une fois P99_FOR, celui-ci effectue le traitement suivant. Pour le domaine d'indice, on récupère simplement les valeurs de nos itérateurs (que l'on utilise pour créer le domaine). Pour le domaine de référence, on prend les valeurs des itérateurs auxquelles on soustrait la borne inférieure correspondante (nous indexant ainsi sur 0), puis on multiplie par le grain et enfin on ajoute l'offset.

```
Pour l'itérateur k de la profondeur 1
(*iter1[k]-ref1[k])*grain1[k]+offset1[k]
```

Il est à noter qu'il est nécessaire de rajouter manuellement le traitement de la dernière profondeur, car notre profondeur p qui est donné par l'utilisateur est indexée sur 0, or P99_FOR effectue ses traitements de 0 jusqu'à p-1. De fait le traitement est incomplet par l'utilisation seule de P99_FOR. Et comme le préprocesseur n'effectue pas de calcul, il n'est pas possible de donner p+1 à P99_FOR.

4.11.2 Création du domaine de référence

On effectue ici la fusion de toutes les listes en comparant toutes les têtes de liste et en prenant celle dont les valeurs de références sont les plus petites, ou si plusieurs ont les mêmes valeurs on prend celle correspondant à la liste d'indice le plus petit.

4.11.3 Utilisation d'une struct dans le for de parcours du domaine et destruction de la liste

```
for(struct{int superSize; int found;List* last; List *act;}s
= {p+1,0,NULL,final};
s.act!=NULL;s.found=0, s.last=s.act, s.act=s.act->next, destruction(s.last))
```

Comme on le voit en 4.3.1, les macros créent deux blocs, le premier permettant le calcul du domaine de référence, et le deuxième son parcours. Il est donc impossible de déclarer des variables hors de ces deux blocs, sinon, lors de la réutilisation de nos macros, on se retrouverait avec un conflit. Cependant, pour effectuer le parcours de notre domaine, il est nécessaire de déclarer quelques variables. L'idée était de les déclarer dans l'initialisation du for, comme on le ferait pour un compteur. Mais il est n'est pas possible de déclarer plusieurs élément de type différents. Il est alors nécessaire de déclarer toutes nos variables dans une structure "anonyme" (partie rouge) et les initialiser (partie bleue). Il sera alors possible d'accéder à ces différentes variables comme on le ferait avec une structure normale.
Pour le parcours, on utilise un pointeur "voyageur", et au fur et à mesure du parcours de la liste, on libère les différents éléments avec la fonction destruction.

4.11.4 Utilisation d'un switch

L'utilisation d'un switch est dû à deux contraintes, premièrement il permet d'énumérer les différents cas, à savoir ici, le code à exécuter. Deuxièmement, afin de réduire le nombre d'accolades à rajouter par l'utilisateur. On place le switch juste sous le for, ce qui enlève la nécessité de mettre des accolades pour le bloc de la boucle for.

4.11.5 Fix profondeur unique

```
XFOREND(0,2,j0,j1,1,1,4,3,1,2,1,0)
```

Dans le cas où l'on voudrait n'avoir qu'un XFOR de profondeur 1 il faut que les variables "générales" (i.e. les variables créées par XFOR_INIT), soient initialisées, on met donc aussi le code d'initialisation de ces structures dans XFOREND et non plus seulement dans XFOR, quitte à ce que le code généré soit du code mort (il le sera dans la grande majorité des cas), code qui sera de toute façon éliminé par le compilateur.

4.12 XFOR_CASE

Cette macro permet de désigner le code à exécuter lorsque l'on est sur un point du domaine de référence. Il doit y avoir autant de XFOR_CASE que de nids de boucles fusionnés.

```
XFOR_CASE(0) printf("test\n");
XFOR_CASE(1) printf("test2\n");
```

4.12.1 switch sans break

Afin de donner une écriture la plus simple possible à l'utilisateur, j'ai décidé de ne pas l'obliger à rajouter de break après le code. On utilise donc la spécificité du switch qui dit qu'une fois que l'on à vérifier un cas, on exécute tout les autres cas jusqu'à arriver à un break ou la fin du switch. Pour contourner le problème d'éxécuter tout les cas (ce que l'on ne veut pas), on utilise un flag nommé found que l'on met à vrai aussitôt que l'on rentre dans un cas, et que l'on teste en rentrant dans un autre cas, s'il est à vrai on break.

4.12.2 Affectation des variables

A chaque point du domaine de référence correspond un vecteur de valeurs (de taille N), contenant les coordonnées du point dans le domaine d'indice), il faut donc les affecter aux itérateurs donnés par l'utilisateur afin que celui-ci puisse faire des traitements avec, comme il le ferait avec plusieurs boucles for.

4.12.3 Libération de la mémoire

Dans le cas où l'itération serait le dernier point du domaine de référence, il faut alors libérer la mémoire allouée lors du reste du xfor.

5 Conclusion

Le projet est fonctionnel, bien que quelques limitations soient pour le moment en place. La forme finale étant très proche de la syntaxe originel d'xfor, il est tout à fait possible de développer un projet en utilisant ces macros, puis de remplacer les macros par les vrais appels lorsqu'un compilateur grand public intégrera xfor. Tout en gardant à l'esprit que, du fait que le calcul du domaine de référence est effectué à l'exécution et non à la compilation et du fait de l'utilisation de listes chainées, les performances sont très inférieures à ce qu'elles seraient avec xfor. Les macros XFOR sont donc un outil pour développer un projet utilisant xfor, afin de valider les modifications que l'on veut effectuer sur les différents domaines d'itérations.

6 Perspectives

6.1 Optimisation

6.1.1 Listes chainées

Un premier axe d'amélioration serait de se débarasser des listes chainées et de les remplacer, soit par une structure dynamique de donnée plus efficace, soit utiliser des tableaux. L'utilisation des tableaux serait très certainement la technique la plus efficace, cependant il est nécessaire de calculer (ou au minimum d'approximer) la taille des tableaux.

6.1.2 Parallélisation de la fusion

Un autre axe d'amélioration serait de paralléliser la fusion des listes, les fusionner deux à deux. puis refusionner les listes résultantes deux à deux, jusqu'à n'en avoir plus qu'une seule. L'implémentation serait faite avec des directives OpenMP.

6.2 Utilisation de macros pour tests

L'implémentation de nouvelles structures de contrôles par le biais de macros est intéressant, il permet l'utilisation de ces nouvelles structures sans que celle-ci soit encore supportée par les compilateurs habituels. Il est donc possible de créer une base reposant sur les spécificités de la nouvelle structure et d'en tester les limites, ou les capacités.

References

[1] JM Bull. Measuring synchronisation and scheduling overheads in openmp. *Citeseer*, 1999.

[2] Jens Gustedt. P99 - preprocessor macros and functions for c99. *http://p99.gforge.inria.fr/*.

[3] Björn Karlsson. *Beyond the C++ Standard Library An introduction to boost*. 2005.

[4] L Dagum; R Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science and Engineering, IEEE*, 1998.

[5] Imèn Fassi; Matthieu Kuhn; Philippe Clauss; Yosr Slama. Multifor for multicore. 2013.