Paralléliser sur un processeur à beaucoup de coeurs

Djallal Rahmoune, Bernard Goossens, David Parello, Katarzyna Porada

Compas'2016 : Parallélisme/ Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Paralléliser sur un processeur à beaucoup de cœurs

Djallal Rahmoune, Bernard Goossens, David Parello et Katarzyna Porada

DALI, Université de Perpignan Via Domitia 66860 Perpignan Cedex 9 France,
LIRMM, CNRS : UMR 5506 - Université Montpellier 2 34095 Montpellier Cedex 5 France,
prénom.nom@univ-perp.fr

Résumé
Cet article montre que la parallélisation actuelle des applications par l'OS, à base de *threads*, est inefficace. Le coût architectural de la parallélisation *pthread* est mesuré et comparé au coût d'une parallélisation par le matériel. De plus, l'article montre aussi que les caches ne sont pas adaptés à la répartition des données que suppose la parallélisation. Il compare l'inefficacité de l'accès à une mémoire partagée à l'efficacité de l'élimination du stockage au profit du calcul redondant et du renommage total des sources et destinations. Enfin, l'article compare la complexité d'une parallélisation basée sur des *threads* non déterministes à la simplicité d'une parallélisation matérielle basée sur le déterminisme de l'ordre d'exécution séquentielle.

Mots-clés : Parallélisation matérielle, processeur à beaucoup de cœurs, déterminisme, calcul redondant, renommage total.

---

1. Introduction

La méthode de parallélisation actuelle des applications, basée sur le système, est inefficace et peu adaptée aux futures architectures de processeurs à beaucoup de cœurs. Une meilleure alternative consiste à paralléliser par le matériel.
L'inefficacité de la parallélisation par *threads* systèmes a au moins deux origines :

- L'appel d'une primitive système de gestion des *threads* est un surcoût.
- La communication entre *threads* se fait par partage (variable globale) ou par copie (message). L'accès à une variable partagée ne bénéficie pas de la hiérarchisation de la mémoire et l'efficacité du passage de message dépend de la proximité du destinataire.

En plus d'être inefficace, la parallélisation par *thread* est non déterministe. Cela rend l'exécution de programmes parallèles difficilement reproductible.
L'article est organisé comme suit. La section 2 mesure le coût architectural de la parallélisation par l'OS dans un environnement *pthread* en nombre d'instructions exécutées par les primitives *pthread*. La section 3 étudie le coût micro-architectural de l'emploi de caches pour accéder à des données partagées. La section 4 décrit la parallélisation d'une exécution par le matériel, basée sur l'ordre séquentiel et évalue le coût de création des sections parallèles, qui est comparé au coût de la création de *threads* mesuré dans la section 2. La section 5 présente un modèle de programmation remplaçant le stockage de données partagées par leur calcul redondant. Enfin, la section 6 présente les travaux antérieurs dans le domaine de la parallélisation et conclut.

```
unsigned int x;
init_x(){...; x=17; ...;}
main(){
 init_x(); tache_a(); tache_b();
 printf("x=%d\n",x);
}
```

```
diviser_en_deux(){
 ...; x=x/2; ...;
}
tache_a(){
 ...; diviser_en_deux(); ...;
}
```

```
doubler(){
 ...; x=x*2; ...;
}
tache_b(){
 ...; doubler(); ...;
}
```

FIGURE 1 – Un programme à paralléliser

2. Le coût architectural de la parallélisation par *threads*

La figure 1 présente un programme qui initialise une variable globale x et effectue deux tâches tache_a et tache_b a priori indépendantes sauf en ce qui concerne la variable partagée x. La variable x ayant une valeur initiale v impaire, le programme affiche $v-1$. La valeur finale de x est liée à l'ordre des tâches. Si b passe avant a, la valeur finale de x est v.
Cet exemple met l'accent sur ce qui est séquentiel dans un programme parallélisable. Ce qui est potentiellement parallèle n'apparaît que sous la forme de points de suspension. Le gain en efficacité vient de la partie parallèle. La difficulté de parallélisation vient de la partie séquentielle dont il faut préserver la sémantique imposée par l'exécution séquentielle déterministe. Dans cette section, nous évaluons le coût du déterminisme.
La figure 2 est la version parallèle *pthread* de ce programme. Il est découpé en trois *threads* créés par le *thread* principal. Ces *threads* sont synchronisés de trois façons.

- Le *thread* principal se synchronise sur la fin des trois *threads* créés, pour afficher la valeur finale de x (pthread_join en lignes 27 à 29).
- Les accès à la variable partagée x sont atomiques par l'emploi du *mutex* proteger_x (lignes 14, 16, 33, 35, 50 et 52).
- L'enchaînement séquentiel de l'initialisation de x, de sa division par deux et de la multiplication par deux de cette moitié est garanti par l'emploi des variables conditionnelles cond_stop_a et cond_stop_b (lignes 17 à 20, 38 à 42, 44 à 47 et 55 à 59).

Cet exemple illustre les difficultés de la parallélisation par les *threads*. Le code de la figure 1 est en rouge sur la figure 2. La cause de ces difficultés est le non déterminisme du calcul des *threads* souligné par Edward Lee dans [8]. Lee écrit : « Les *threads* ... rendent les programmes absurdement non déterministes et dépendent du style de programmation pour contraindre ce non déterminisme de façon à atteindre un but déterministe ». L'entrelacement des instructions machines exécutées peut être quelconque par le jeu des interruptions. Il faut éliminer tous les entrelacements menant à un résultat différent de celui du programme séquentiel, en employant des mécanismes de synchronisations des *threads*. Néanmoins, cette synchronisation ne doit pas se faire au détriment du parallélisme. Tous les entrelacements valides doivent rester possibles.
Avec gdb, nous avons mesuré le surcoût de la création de *threads* et des mécanismes de synchronisation employés. La mesure a été effectuée sur un ordinateur équipé d'un processeur Intel® Core™ i7-4900MQ et piloté par Ubuntu 14.04. Le code *pthread* a été compilé avec la version Ubuntu 4.8.4-2 de gcc et la version libpthread-stubs0-dev0.3-4 de la librairie *pthread* (options −O3 et −static). Le résultat de cette mesure est résumé dans la table de la figure 3.
Pour les variables conditionnelles, la mesure englobe toute la procédure : verrouillage, attente et déverrouillage pour pthread_cond_wait (lignes 38 à 42 et 55 à 59) et verrouillage, signalisation et déverrouillage pour pthread_cond_signal (lignes 17 à 20 et 44 à 47).

```
1  pthread_mutex_t
2    proteger_x=PTHREAD_MUTEX_INITIALIZER;
3  pthread_mutex_t
4    proteger_stop_a=PTHREAD_MUTEX_INITIALIZER;
5  pthread_mutex_t
6    proteger_stop_b=PTHREAD_MUTEX_INITIALIZER;
7  pthread_cond_t
8    cond_stop_a=PTHREAD_COND_INITIALIZER;
9  pthread_cond_t
10   cond_stop_b=PTHREAD_COND_INITIALIZER;
11 bool stop_a=true, stop_b=true;
12 unsigned int x;
13 void *init_x(){
14   ...; pthread_mutex_lock(&proteger_x);
15   x=17;
16   pthread_mutex_unlock(&proteger_x);
17   pthread_mutex_lock(&proteger_stop_a);
18   stop_a=false;
19   pthread_cond_signal(&cond_stop_a);
20   pthread_mutex_unlock(&proteger_stop_a); ...;
21 }
22 main(){
23   pthread_t tid1, tid2, tid3;
24   pthread_create(&tid1, NULL, init_x, NULL);
25   pthread_create(&tid2, NULL, tache_a, NULL);
26   pthread_create(&tid3, NULL, tache_b, NULL);
27   pthread_join(tid1, NULL);
28   pthread_join(tid2, NULL);
29   pthread_join(tid3, NULL);
30   printf("x=%d\n",x);
31 }
```

```
32 diviser_en_deux(){
33   ...; pthread_mutex_lock(&proteger_x);
34   x=x/2;
35   pthread_mutex_unlock(&proteger_x); ...;
36 }
37 void *tache_a(){
38   ...; pthread_mutex_lock(&proteger_stop_a);
39   while (stop_a)
40     pthread_cond_wait(&cond_stop_a,
41                       &proteger_stop_a);
42   pthread_mutex_unlock(&proteger_stop_a);
43   diviser_en_deux();
44   pthread_mutex_lock(&proteger_stop_b);
45   stop_b=false;
46   pthread_cond_signal(&cond_stop_b);
47   pthread_mutex_unlock(&proteger_stop_b); ...;
48 }
```

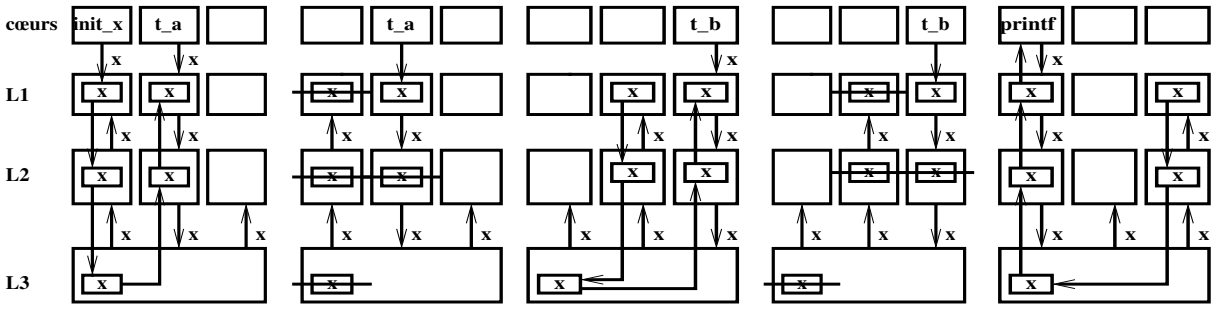
```
49 doubler(){
50   ...; pthread_mutex_lock(&proteger_x);
51   x=x*2;
52   pthread_mutex_unlock(&proteger_x); ...;
53 }
54 void *tache_b(){
55   ...; pthread_mutex_lock(&proteger_stop_b);
56   while (stop_b)
57    pthread_cond_wait(&cond_stop_b,
58                      &proteger_stop_b);
59   pthread_mutex_unlock(&proteger_stop_b);
60   doubler(); ...;
61 }
```

FIGURE 2 – Le programme parallélisé par trois *threads*

pthread_	create	join	mutex_lock	mutex_unlock	cond_wait	cond_signal
instructions	726	143	29	18	49	78

FIGURE 3 – Nombre d'instructions exécutées par les primitives *pthreads*

A première vue, le coût de création d'un *thread* est faible : 726 instructions s'exécutent en moins de 500 cycles, ce qui représente moins de 150ns sur un processeur à 3Ghz. Mais ce coût est à mettre en regard du travail à faire dans le *thread*. Cela condamne la parallélisation de grain très fin. Par exemple, la vectorisation d'un calcul simple ne peut être mise en œuvre par des *threads* comme elle l'est sur un GPU. L'incapacité pour un processeur à beaucoup de cœurs à vectoriser au-delà de la largeur des registres AVX a conduit les constructeurs à intégrer un GPU qui représente par exemple 40% de la surface d'un processeur Skylake à 4 cœurs.
Le verrouillage par *mutex* semble bon marché (29 instructions et 18 pour le déverrouillage, soit 47 au total). Mais dans l'exemple, il est à mettre en regard de la complexité de la portion verrouillée, qui se réduit à une instruction machine (verrouillage des lignes 15, 34 et 51). Par ailleurs, le prix doit être payé à chaque accès, ce qui, dans l'exemple, multiplie par 47 le nombre d'instructions exécutées par accès à x.



(a) lecture de x par init_x et tache_a | (b) écriture de x par tache_a | (c) lecture de x par tache_b | (d) écriture de x par tache_b | (e) lecture de x par printf

FIGURE 4 – Accès à la variable x en mémoire hiérarchisée

3. Le coût micro-architectural de la parallélisation par *threads*

Le nombre d'instructions exécutées est un premier élément de la mesure du coût de la parallélisation. Mais le temps d'exécution d'une instruction peut varier dans de grandes proportions dans les micro-architectures d'aujourd'hui. Par exemple, un accès à la mémoire peut coûter entre quelques cycles si l'adresse accédée est dans le premier niveau de cache et quelques centaines de cycles si elle n'est pas dans le processeur.

La hiérarchie mémoire a un effet bénéfique sur le coût moyen de l'accès à x dans l'exécution séquentielle du code de la figure 1. Le premier accès à x par init_x traverse toute la hiérarchie. Une fois x caché dans l'unique cœur d'exécution, les accès suivants se font dans son cache L1.

La figure 4 montre en 5 étapes (a) à (e) l'inefficacité de la hiérarchie mémoire sur le coût moyen de l'accès à x dans l'exécution parallèle du code de la figure 2. On suppose une répartition du code sur 3 cœurs. A l'étape (a), l'écriture de x par init_x dans le L1 ne profite pas à la tâche a. A l'étape (b), la modification de x par la tâche a invalide le x initialisé par init_x. A l'étape (e), printf doit accéder au cache L1 du cœur ayant exécuté la tâche b (étapes (c) et (d)). La hiérarchisation de la mémoire, si utile à la performance d'une exécution séquentielle est inefficace dans le cas d'une exécution distribuée.

Les accès aux variables partagées passent par le L3. Les processeurs actuels ont des caches L3 gigantesques (le cache L3 du Xeon E7-v3 à 18 cœurs est de 45MO). Ceci, ajouté à la nécessité d'intégrer un GPU, explique le faible nombre de cœurs des CPU actuels.

4. Répartir matériellement le calcul à partir de l'ordre séquentiel

Pour paralléliser sans perdre le déterminisme, il faut répartir le code à exécuter en respectant les dépendances producteurs/consommateurs issues de l'ordre total séquentiel.

En s'appuyant sur l'instruction machine d'appel de fonction pour dédoubler le flux de lecture des instructions, on peut découper le code de la figure 1 en 6 sections parallèles S1 à S6 :

- S1 commence à la fonction main, se poursuit dans init_x et se termine au retour d'init_x.
- S2 commence au retour d'init_x, appelle tache_a puis diviser_en_deux qui la termine.
- S3 commence au retour de diviser_en_deux et se termine à la fin de tache_a.
- S4 commence au retour de tache_a, appelle tache_b puis doubler qui la termine.
- S5 commence au retour de doubler et se termine à la fin de tache_b.

— S6 commence au retour de `tache_b` et se termine à la fin de `main`.

La section S1 dédouble le flux de lecture des instructions à l'appel d'`init_x` et crée S2. La section S2 est lue en parallèle avec la suite de S1. D'autres dédoublements ont lieu lors des appels à `diviser_en_deux`, `tache_a`, `doubler` et `tache_b`.

En supposant qu'un cœur soit capable d'envoyer un compteur de programme à un autre cœur, celui qui lit les instructions de S1 peut, lorsqu'il décode l'appel à `init_x`, transmettre au cœur voisin l'adresse du code à lire après le retour d'`init_x` (début de S2).

La figure 5 montre la répartition de l'exécution sur 6 cœurs. Chaque flèche noire désigne un dédoublement de la lecture des instructions (elle se poursuit localement et une nouvelle section démarre sur le cœur visé par la flèche). Chaque flèche rouge désigne le successeur d'une section dans l'ordre séquentiel. En suivant les flèches rouges, on reconstitue l'ordre séquentiel.

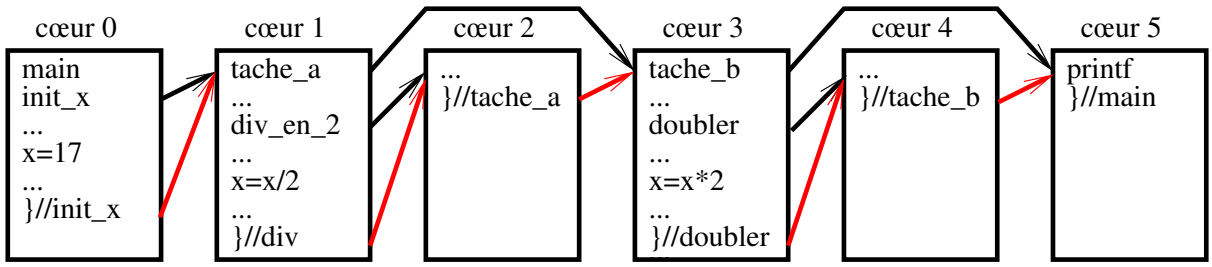


FIGURE 5 – Répartition de l'exécution sur 6 cœurs

Une telle construction de la répartition doit être comparée à la création de *threads*. L'installation d'une nouvelle section sur un cœur dure le temps de la transmission du compteur de programme (communication cœur à cœur). En comparaison, la création d'un *thread* dure le temps de l'exécution des 726 instructions de `pthread_create`.

A cette efficacité s'ajoute l'avantage de pouvoir reconstruire l'ordre séquentiel. Cette reconstruction ne coûte qu'une communication cœur à cœur, pour lier une section à son successeur quand sa fin est atteinte (matérialisation de la flèche rouge).

Grâce à l'ordre séquentiel, on peut associer tout consommateur à son producteur. Par exemple, l'instruction $x = x/2$ sur le cœur 1 lit la valeur de x écrite par l'instruction $x = 17$ sur le cœur 0. Un parcours de l'ordre séquentiel en marche arrière depuis le lecteur s'arrête au premier écrivain rencontré. De même, l'instruction $x = x * 2$ sur le cœur 3 lit la valeur de x écrite par l'instruction $x = x/2$ sur le cœur 1.

Cette association entre un consommateur et son producteur garantit un calcul parallèle déterministe si on synchronise matériellement la lecture avec l'écriture.

Ainsi, l'instruction $x = x * 2$ peut être lue et décodée avant l'instruction $x = x/2$ qui est son producteur. Mais tant que celui-ci n'est pas exécuté, l'emplacement où $x = x/2$ doit écrire reste vide, ce qui bloque la lecture de x dans $x = x * 2$.

Le coût de cette synchronisation doit être comparé à celui de la sérialisation par variables conditionnelles dans l'environnement *pthread*. La synchronisation matérielle nécessite la transmission cœur à cœur de la variable à lire, en partant du cœur consommateur et en allant jusqu'au cœur producteur. Là, on attend que l'instruction productrice soit exécutée. Puis, la valeur écrite est transmise directement au cœur consommateur. Le coût est proportionnel à la distance du producteur au consommateur. Dans l'environnement *pthread*, le coût de la synchronisation de

parties séquentielles est élevé mais constant (78 instructions exécutées par le cœur producteur et 49 instructions exécutées par le cœur consommateur).

5. Remplacer le stockage par le calcul redondant

La section précédente a fait apparaître un inconvénient de la parallélisation déterministe : pour associer un consommateur à son producteur, il faut examiner toutes les destinations intermédiaires, c'est-à-dire parcourir l'ordre séquentiel de leurs sections. On peut imaginer qu'entre un producteur et un consommateur se situent de nombreuses sections correspondant à la parallélisation des itérations d'une boucle. Si la boucle ne touche pas à la variable consommée, son producteur est une section précédant toutes celles de la boucle. Dans ces conditions, la récupération de la valeur produite nécessite un long parcours.
Dans l'environnement *pthread*, le producteur écrit en mémoire. Le consommateur est synchronisé (par exemple avec une variable conditionnelle) et vient lire en mémoire une fois l'écriture faite. Si le producteur et le consommateur sont éloignés, alors l'accès mémoire coûte plus cher parce qu'il passe par une plus grande partie de la hiérarchie des caches. Le calcul du consommateur commence une fois la valeur consommée reçue, c'est-à-dire après production, puis signalisation de cette production, puis finalisation de l'attente, puis accès à la valeur produite en mémoire. La signalisation et l'attente à elles seules représentent 127 instructions exécutées.
La figure 6 montre une version du programme séquentiel où la variable x a été supprimée et remplacée par des fonctions. L'intérêt est d'éliminer le stockage et de le remplacer par un calcul redondant. De cette façon, on évite l'emploi de la hiérarchie mémoire dont on a montré qu'elle est inefficace quand le calcul est réparti. On rend toute consommation dépendante d'une production locale, ce qui élimine toute communication entre cœurs. La contre-partie est un emploi plus important des ressources de calculs. En effet, toute valeur consommée localement doit être produite localement. On doit recalculer x dans la tâche a, dans la tâche b et pour la fonction d'affichage. Ainsi, au lieu d'une multiplication et une division, on fait 3 divisions et 2 multiplications. Dans un processeur à beaucoup de cœurs il vaut mieux utiliser les opérateurs de calculs, disponibles en nombre (au moins une UAL par cœur), plutôt que le stockage en mémoire.

```
get_main_x1(){return 17;}
get_main_x2(){
 return get_main_x1()/2;
}
get_main_x3(){
 return get_main_x2()*2;
}
init_x(){
 unsigned int x;
 ...; x=get_main_x1(); ...;
}
main(){
 init_x(); tache_a(); tache_b();
 printf("x=%d\n",get_main_x3());
}
```

```
get_a_x(){return 17;}
diviser_en_deux(){
 unsigned int x;
 ...; x=get_a_x()/2; ...;
}
tache_a(){
 ...
 diviser_en_deux();
 ...
}
```

```
get_b_x2(){return 17;}
get_b_x1(){
 return get_b_x2()/2;
}
doubler(){
 unsigned int x;
 ...; x=get_b_x1()*2; ...;
}
tache_b(){
 ...
 doubler();
 ...
}
```

FIGURE 6 – Un programme parallélisable sans communication

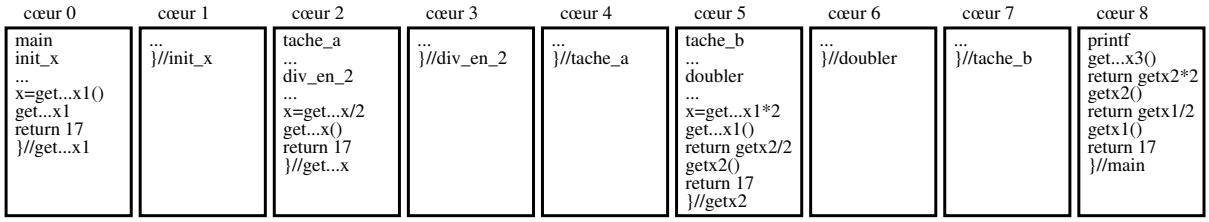


FIGURE 7 – Répartition de l'exécution sans communication

Le programme de la figure 6 peut être parallélisé en 9 sections distribuées comme indiqué sur la figure 7. Toutes les sections sont indépendantes. Par exemple, la première section sur le cœur 0 appelle `init_x` puis `get_main_x1`. La seconde section sur le cœur 1 est la fin de `init_x` après le retour de `get_main_x1`.
Les calculs sont redondants, avec 3 divisions (cœurs 2, 5 et 8) et 2 multiplications (cœurs 5 et 8). Le cœur 8 reproduit le calcul séquentiel, depuis l'initialisation de `x` jusqu'à son affichage par `printf`. Le gain de la parallélisation se situe dans l'exécution des autres parties du code, correspondant aux points de suspension. Les parties parallèles sont indépendantes et évaluées sur des cœurs différents et sans communications.

6. Travaux antérieurs et conclusion

En 2012, Diaz, Munoz-Caro et Nino [2] ont publié une étude des modèles de programmation parallèle et des outils disponibles pour l'emploi des processeurs à plusieurs cœurs.
En compilation, on automatise la parallélisation des boucles en s'appuyant sur le modèle polyhédrique [3], [1]. La communauté du parallélisme parallélise en partie à la main et en partie avec des outils et des langages (API *pthread*, MPI, openMP, CUDA, OpenCL ...).
La parallélisation par le matériel (voir par exemple [7][9][10][11]) se heurte d'un côté au faible parallélisme d'instructions (ILP) des applications [12] et de l'autre aux usages basés sur la spéculation et les caches [6]. Les caches sont mal adaptés aux calculs répartis. Il en est de même de la spéculation qui engendre des calculs inutiles. Dans [5], les auteurs décrivent un modèle de processeur à beaucoup de cœurs où le contrôle est calculé plutôt que prédit. Le calcul du contrôle est moins rapide que sa prédiction mais il se parallélise. De plus, les cœurs remplacent la hiérarchie mémoire de données par un ensemble de ressources de renommage (extension du renommage de registres à la mémoire). Ce choix est adapté au modèle de programmation décrit dans la section 5. Dans [4], les auteurs ont montré que le faible ILP des applications vient de l'architecture cible du compilateur. En enlevant les mouvements sur la pile, en parallélisant le calcul du contrôle et en généralisant le renommage, on atteint plusieurs milliers d'instructions exécutables par cycle. De plus, cet ILP augmente avec la taille de la donnée du *benchmark*. En éliminant les dépendances parasites introduites par l'architecture, on récupère le parallélisme de l'algorithme initial de l'application.
Aujourd'hui, la technologie nous permet de placer plusieurs milliers de petits cœurs de calcul sur un composant, comme c'est le cas sur un GPU. Il est urgent de pouvoir paralléliser automatiquement toute exécution, c'est-à-dire d'en distribuer le code sur les ressources disponibles pour en exploiter le parallélisme d'instructions. Cela s'étend à l'OS lui-même, dont le noyau a été conçu à une époque où le partage d'une ressource unique était la règle.
Le modèle de répartition des calculs présenté dans cet article a deux vertus : il préserve le

déterminisme du calcul séquentiel et il évite les communications en substituant le calcul redondant au stockage. Toute exécution parallèle est reproductible, ce qui n'est pas le cas des parallélisations basées sur des *threads*. En minimisant les communications, les exécutions sur un processeur à beaucoup de cœurs sont efficaces et « scalables ».

Bibliographie

1. Benabderrahmane (M.-W.), Pouchet (L.-N.), Cohen (A.) et Bastoul (C.). – The polyhedral model is more widely applicable than you think. – In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, *CC'10/ETAPS'10*, CC'10/ETAPS'10, pp. 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.
2. Diaz (J.), Munoz-Caro (C.) et Nino (A.). – A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, n8, 2012, pp. 1369–1386.
3. Feautrier (P.). – Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, vol. 20, n1, 1991, pp. 23–53.
4. Goossens (B.) et Parello (D.). – Limits of instruction-level parallelism capture. *Procedia Computer Science*, vol. 18, n0, 2013, pp. 1664–1673. – 2013 International Conference on Computational Science.
5. Goossens (B.), Parello (D.), Porada (K.) et Rahmoune (D.). – Toward a core design to distribute an execution on a manycore processor. *In : Parallel Computing Technologies*, éd. par Malyshkin (V.), pp. 390–404. – Springer International Publishing, 2015.
6. Hennessy (J. L.) et Patterson (D. A.). – *Computer Architecture, Fifth Edition : A Quantitative Approach*. – San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2011, 5th édition.
7. Kim (H.-S.) et Smith (J.). – An instruction set and microarchitecture for instruction level distributed processing. – In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pp. 71–81, 2002.
8. Lee (E. A.). – The problem with threads. *Computer*, vol. 39, n5, mai 2006, pp. 33–42.
9. Mehrara (M.), Hao (J.), Hsu (P.-C.) et Mahlke (S.). – Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. – In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '09*, PLDI '09, pp. 166–176, New York, NY, USA, 2009. ACM.
10. Ranjan (R.), Latorre (F.), Marcuello (P.) et Gonzalez (A.). – Fg-stp : Fine-grain single thread partitioning on multicores. – In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 15–24, 2011.
11. Sharafeddine (M.), Jothi (K.) et Akkary (H.). – Disjoint out-of-order execution processor. *Transactions on Architecture and Code Optimization (TACO)*, vol. 9, n3, sept 2012, pp. 19 :1–19 :32.
12. Wall (D. W.). – Limits of instruction-level parallelism. – In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS IV*, ASPLOS IV, pp. 176–188, 1991.