Analyse et réduction du chemin critique dans l'exécution d'une application

Katarzyna Porada, David Parello, Bernard Goossens

ComPAS'2014 : Parallélisme/ Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Analyse et réduction du chemin critique dans l'exécution d'une application

Katarzyna Porada, David Parello et Bernard Goossens

DALI, Université de Perpignan Via Domitia 66860 Perpignan Cedex 9 France,
LIRMM, CNRS : UMR 5506 - Université Montpellier 2 34095 Montpellier Cedex 5 France,
katarzyna.porada@etudiant.univ-perp.fr, david.parello@univ-perp.fr,
bernard.goossens@univ-perp.fr

Résumé
L'article étudie les plus longues chaînes de dépendances (LDC) entre instructions dans l'exécution des applications de la suite de *benchmarks cBench*. Deux modèles d'exécution sont mis en œuvre. Le modèle *séquentiel* reproduit le fonctionnement d'un processeur actuel. Le modèle *parallèle* correspond à un processeur idéal qui disposerait de la trace d'exécution et d'un déploiement complet de l'espace nécessaire au stockage des données et résultats. Les LDC du modèle parallèle sont composées des dépendances Lecture Après Ecriture (LAE) entre données issues de l'algorithme alors que les LDC du modèle séquentiel ajoutent d'innombrables dépendances de données architecturales, enchâssant le calcul dans un carcan qui en enlève tout le parallélisme. L'enlèvement de toutes ces dépendances parasites est nécessaire pour paralléliser l'exécution. Cela peut être fait par le matériel à condition de pouvoir i) extraire les instructions en parallèle, ii) étendre le renommage des registres à la mémoire et iii) éliminer les vraies dépendances sur le pointeur de pile et sur les compteurs de boucles vectorisables.

Mots-clés : parallélisation automatique, parallélisme d'instructions, plus longue chaîne de dépendances, séquentialisation de l'extraction, séquentialisation de la pile.

---

1. Introduction

Depuis quelques années déjà, les constructeurs emploient les transistors pour multiplier les cœurs de calcul. En 2006, l'Intel Core 2 Duo contenait deux cœurs. Selon la loi de Moore [8], sept ans plus tard, nous devrions avoir des processeurs à 16 cœurs. D'une part, le Xeon Phi a 60 cœurs, ce qui laisserait penser que nous sommes en avance. Mais d'un autre côté, le Xeon E7 n'a au mieux que 12 cœurs et les processeurs de nos ordinateurs portables n'ont que 4 cœurs (Intel Core I7) ou 8 cœurs (AMD FX 8320), ce qui traduit un léger retard.
La multiplication des cœurs sur un composant n'est pas qu'une affaire de transistors. C'est bien d'avoir des cœurs à condition d'en profiter. On peut utiliser des cœurs soit parce qu'on a autant de tâches à exécuter, soit parce qu'on a une tâche que l'on peut paralléliser et distribuer sur l'ensemble des cœurs. Dans cet article, nous nous intéressons à cette parallélisation. Les développeurs la voudraient la plus transparente possible. En clair, ils souhaiteraient pouvoir passer d'un algorithme parallèle à un programme exprimé dans le langage qu'ils utilisent, par exemple C ou C++, et l'exécuter en parallèle sur leur processeur multi-cœur.

```
somme(t[0,n-1])
  si n=2 alors t[0]+t[1]
  sinon
    somme(t[0,n/2-1]) +
    somme(t[n/2,n-1])
```

(a) Algorithme

```
unsigned long
somme(unsigned long t[],
      unsigned long n){
  if (n==2)
    return t[0]+t[1];
  else
    return somme(t,n/2) +
           somme(&t[n/2],n/2);
}
```

(b) Programme C

```
void *somme(void *st){
 ST str1,str2;
 unsigned long s,s1,s2;
 pthread_t tid1, tid2;
 if (((ST *)st)->i>2){
  str1.p=((ST *)st)->p;
  str1.i=((ST *)st)->i/2;
  pthread_create(&tid1,NULL,somme,(void *)&str1);
  str2.p=((ST *)st)->p + ((ST *)st)->i/2;
  str2.i=((ST *)st)->i/2;
  pthread_create(&tid2,NULL,somme,(void *)&str2);
  pthread_join(tid1,(void *)&s1);
  pthread_join(tid2,(void *)&s2);
 }
 else{s1=((ST *)st)->p[0];s2=((ST *)st)->p[1];}
 s=s1+s2;
 pthread_exit((void *)s);
}
```

(c) Programme pthreads

FIGURE 1 – Une réduction de somme

La réalité est moins idyllique. Le chemin le plus classique est de transformer un programme à la main pour lui donner du parallélisme au travers d'une bibliothèque de primitives du système, par exemple *pthreads* [9] ou *mpi* [6]. Cette transformation est pénible, source d'erreurs, nécessitant une délicate mise au point et débouchant sur un code peu lisible et difficile à faire évoluer. Pour toutes ces raisons au moins, peu d'applications ont été parallélisées et profitent aujourd'hui des cœurs d'un processeur. Les programmes parallèles naissent rarement et meurent vite.
La figure 1 montre à gauche un algorithme de réduction de somme, au centre, son implémentation en C et à droite sa parallélisation en *pthreads*. Le code C a le mérite d'être proche de l'algorithme. Le code *pthreads* est moins lisible par la faute de ses redondances. Le code C pourrait suffire, à condition que son exécution puisse se faire en parallèle. Rien dans l'expression C ne l'interdit. Seules la traduction architecturale et son implémentation micro-architecturale séquentialisent l'exécution des instructions du langage machine.
Le système parallélise des *threads* qu'il faut exhiber, synchroniser et faire communiquer. Le coût se compte en milliers de cycles. Dans l'exemple de la réduction de somme, il faut avoir quelques centaines de milliers d'éléments à sommer pour qu'un bénéfice puisse être observé.
Pour paralléliser un code, plutôt que de faire appel au système, on peut s'appuyer sur le matériel. Le matériel parallélise les instructions dès lors qu'elles sont indépendantes entre elles. La parallélisation par le matériel est difficile parce qu'il faut examiner l'ensemble des instructions d'une exécution pour en déterminer les liens de dépendances. Cet examen se fait traditionnellement pendant la phase de renommage, qui détermine les dépendances existant entre les instructions juste extraites et celles en attente. Le renommage dans les cœurs actuels permet de paralléliser une centaine d'instructions, c'est-à-dire ce qu'on arrive à extraire entre deux prédictions incorrectes du prédicteur de sauts.
Dans l'exemple de la réduction de somme en C, on peut exécuter les instructions du second appel récursif en même temps que celles du premier. Mais l'écart entre ces deux portions de code peut être arbitrairement grand, si bien que le matériel actuel ne peut paralléliser l'exécution.
La séquentialisation des exécutions provient de quatre types de dépendances [2] :

- les dépendances de flot, qui lient les blocs de base par leur extraction ;
- les vraies dépendances sur le pointeur de pile et les fausses dépendances sur le contenu de la pile, qui lient les appels de fonctions ;
- les vraies dépendances sur les variables de contrôle des boucles, qui lient les itérations ;
- les vraies dépendances sur les variables intermédiaires, qui lient calculs et mouvements.

En éliminant ces quatre types de dépendances, on permet la parallélisation. L'élimination des dépendances de flot permet de paralléliser l'extraction du code. L'élimination des dépendances de pile permet de paralléliser la construction des cadres d'appel. L'élimination des dépendances du contrôle des boucles permet d'en vectoriser l'exécution. L'élimination des mouvements permet de replier les calculs en n'en conservant que l'enchaînement des opérations.
Le chemin critique d'une exécution, représenté par une plus longue chaîne de dépendances (LDC) [1], est constitué principalement de telles dépendances. Ce chemin s'allonge proportionnellement à la taille de la donnée traitée. En enlevant ces dépendances, le chemin critique ne contient plus que des dépendances de données de l'algorithme.

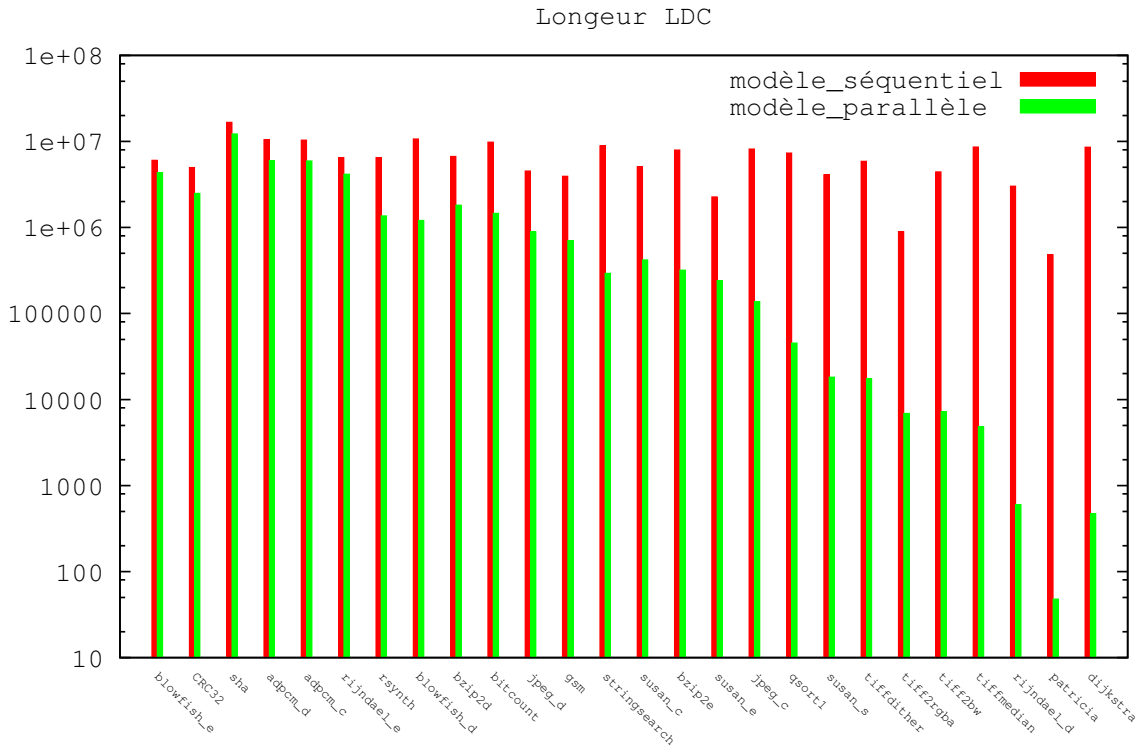

FIGURE 2 – Les LDC des modèles séquentiel et parallèle pour les *benchmarks* de la suite *cBench*

La figure 2 présente les LDC des exécutions des *benchmarks* de la suite *cBench* [4]. Deux modèles d'exécution sont comparés. Le premier est le modèle séquentiel qui s'apparente au modèle d'exécution adopté par les processeurs actuels. Le second modèle est parallèle dans lequel les quatre types de dépendances ont été enlevés. La figure montre que les exécutions dans le modèle séquentiel sont contraintes par des chemins critiques très longs : les LDC comptent 10M d'instructions en moyenne pour des tailles d'exécutions de l'ordre de 50M d'instructions (voir la table 1). Cela donne un ILP (Instruction-Level Parallelism) d'environ 5 instructions par cycle, qui correspond aux mesures publiées (par exemple [10]). Dans le modèle parallèle, certaines LDC ne sont pas sensiblement plus courtes. Cela tient au fait que les *benchmarks* correspondants sont issus d'algorithmes séquentiels. Mais pour d'autres, issus d'algorithmes pa-

rallèles, le chemin critique est considérablement raccourci, se réduisant à quelques centaines d'instruction, ce qui donne un ILP de plusieurs dizaines de milliers d'instructions par cycle.
L'article est organisé comme suit. La section 2 décrit les deux modèles d'exécution étudiés. La section 3 décrit l'expérience de simulation effectuée pour obtenir les LDC d'applications standards (issues de la suite *cBench*) dans les deux modèles. La comparaison de ces LDC met en évidence d'une part les raisons de la séquentialisation du modèle d'exécution actuel et d'autre part la parallélisation qu'on peut atteindre par le matériel et le bénéfice qui peut en résulter.

2. Les deux modèles étudiés

Pour observer la séquentialisation de l'exécution de différents programmes standards, pour analyser les raisons de cette séquentialisation et pour faire ressortir les moyens nécessaires à la parallélisation, nous avons comparé deux modèles d'exécution.
Le modèle *séquentiel* est caractéristique des cœurs actuels. Le modèle *parallèle* est un modèle idéal où les dépendances inutiles sont décrétées inexistantes.
Notre but est de montrer ce qui séquentialise une exécution et d'ouvrir des perspectives à la parallélisation automatique par le matériel.

2.1. Le modèle séquentiel

En supposant qu'un prédicteur de saut soit parfait, cela ne suffirait pas à s'affranchir de la séquentialisation de l'extraction du code. Le modèle architectural des cœurs actuels, basé sur un pointeur d'instructions unique, impose une extraction bloc après bloc. Au mieux, chaque bloc de base est extrait un cycle après son prédécesseur. Dans l'exemple de la somme, on n'extrait le second appel récursif qu'une fois que le premier l'a été en totalité.
Le modèle séquentiel simulé correspond à l'état ultime que pourrait atteindre un cœur actuel : le prédicteur est parfait et en plus, on peut renommer sans limite (aujourd'hui, la limite est celle du nombre de registres de renommages, soit une centaine). L'impact des dépendances est réduit au minimum : toute instruction peut s'exécuter au cycle qui suit la production du dernier octet qu'elle prend en source et elle produit elle-même tous ses résultats en un seul cycle.
Parmi les dépendances de flot, les dépendances de contrôle sont toutes éliminées grâce au prédicteur parfait. Les fausses dépendances de registre sont toutes éliminées grâce au renommage illimité. En revanche, les dépendances de mémoire sont toutes conservées, ainsi que toutes les vraies dépendances de registre.
Dans ce modèle, l'extraction se fait par blocs de base entiers, d'un saut pris au suivant (les sauts non pris ne terminent pas les blocs de base). La hiérarchie mémoire est supposée parfaite et capable de délivrer un bloc de base en un cycle, quelle que soit sa longueur. Le tampon d'extraction a une capacité illimitée. Dans le même cycle, les instructions extraites sont renommées : on crée toutes les copies nécessaires pour tous les registres de destination. Une fois extraite et renommée, une instruction attend ses sources. Elle s'exécute dès qu'elle est prête. Le nombre d'instructions en attente est illimité. On peut exécuter au même cycle toutes les instructions prêtes, ce qui suppose un nombre illimité d'unités de calculs. L'exécution dure un cycle, quelle que soit l'instruction (les opérateurs, dont l'accès à la mémoire, ont une latence d'un cycle).
Concernant les accès à la mémoire (chargements et rangements), l'adresse $r + k$, où r est un registre et k est une constante, est calculée au mieux pendant le cycle de renommage mais après que r ait été produit. Pour un chargement, l'accès a lieu au mieux dans le même cycle que l'extraction et le renommage mais après que tous les accès (chargements et rangements) antérieurs à la même adresse aient été effectués. Des accès à une même adresse se font dans l'ordre. Des accès à deux adresses distinctes sont indépendants.

```
    Code en C :        for (i=0;i<100;i++) ...

    Code en X86        C                  Renommage      Renommage          Renommage
                                          itération 0    itération 1  ...   itération 99

    movq $0,%rcx       i=0                RR0=0
.L1:
    ...
    addq $1,%rcx       i++                RR1=RR0+1      RR2=RR0+2          RR100=RR0+100
    cmpq %rcx,$100     i<100              RR1<100        RR2<100            RR100<100
    jne .L1            si !égal vers .L1
```

FIGURE 3 – Renommage de constante dans un contrôle de boucle

Pour résumer, si une instruction i est extraite au cycle c_{fetch}, toutes les instructions du même bloc de base (jusqu'au premier saut pris) sont aussi extraites au cycle c_{fetch}. Elles sont toutes renommées au cycle c_{fetch} ce qui suppose que l'extraction, le décodage et le renommage sont fusionnés dans le même étage de pipeline. L'instruction i lit ses sources $s_1...s_n$ au cycle $c_{exe} = \max(c_1...c_n) + 1$, où c_i est le cycle de production de s_i ($i = 1...n$). Elle s'exécute en un cycle et produit tous ses resultats à la fin de c_{exe}. Enfin, si un bloc de base est extrait au cycle c_{fetch}, son successeur l'est au cycle $c_{fetch} + 1$.
L'intérêt de ce modèle est de montrer que les cœurs actuels, même améliorés à l'extrême, produiront des exécutions qui resteront séquentielles.

2.2. Le modèle parallèle

Le modèle parallèle se différencie du modèle séquentiel en ce que :

- tous les blocs de base de la trace sont pré-extraits et pré-renommés au premier cycle ;
- la gestion du pointeur de pile est négligée ;
- le renommage est étendu aux constantes, à la mémoire et aux mouvements.

La pré-extraction du code permet de s'affranchir de la séquentialisation de l'extraction. Le pré-renommage construit une destination unique pour chaque résultat de chaque instruction.

Le renommage des constantes

Les constantes employées dans les instructions d'accumulation (i++, i- -, i+=2...) sont aussi pré-renommées. Au renommage, la constante est adaptée pour faire dépendre l'instruction d'accumulation à l'initialisation la plus récente. La figure 3 montre que le renommage des constantes permet en particulier d'éliminer la chaîne de dépendances induite par l'incrémentation de la variable de contrôle d'une boucle. Plutôt que de dépendre du contrôle de l'itération $i - 1$, le contrôle de l'itération i est rendu dépendant de l'initialisation de la variable contrôlée (dans l'exemple, placée dans le registre rcx) avant l'entrée dans la boucle.
Le renommage des constantes a été proposé par Fahs [3].

Le renommage des mouvements

Le renommage de la source s d'un mouvement $d = s$ (où d et s sont des registres ou des variables en mémoire) lui substitue son dernier nom, soit n_s. La destination d reçoit la référence n_s si la valeur de s n'est pas établie. Une référence ultérieure à d est renommée n_s. Ainsi, le renommage d'une chaîne de mouvements en lie tous les points intermédiaires et tous les calculs qui en dépendent à la valeur initiale. La figure 4 illustre le renommage des mouvements. Le calcul de $b = a$; $c = b$; $d = c$; $e = d + x$; ne se fait pas en quatre étapes dont trois copies de valeurs mais en deux étapes, avec d'une part le renommage qui fait pointer b, c et d sur a et d'autre part les calculs de a et e, le second dépendant du premier. Une chaîne de dépendance

```
Code en X86        C                  Renommage
...                                                 //c est %rcx renommé RR0 et a est %rax renommé RR1
addq %rax,%rcx     c=c+a              RR10=RR0+RR1  //nouveau c dans RR10
...
movq %rcx,%rbx     b=c                RR11=RR10     //nouveau b dans RR11 et c produit par RR10
...                                                 //RR11 contient d'abord un pointeur sur RR10
...                                                 //puis la valeur de RR10 dès qu'elle est connue
...
...                                                 //x est %r12 renommé RR2
addq %rbx,%r12     x=x+b              RR12=RR2+RR10 //nouvel x dans RR12 et b remplacé par c
...
movq %rbx,%rdx     d=b                RR13=RR10     //nouveau d dans RR13 et b produit par RR10
...                                                 //RR13 pointe sur RR10
...
addq %rdx,%r12     x=x+d              RR14=RR12+RR10 //nouvel x dans RR14 et d remplacé par c
```

FIGURE 4 – Renommage d'un enchaînement de mouvements

de longueur 4 a été remplacée par une chaîne de longueur 2.

Le renommage étendu à la mémoire

En étendant le renommage à la mémoire et en le rendant pré-existant, on fait pré-exister les emplacements nécessaires au stockage de toutes les valeurs calculées par l'exécution. Par exemple, le second appel de la somme peut s'exécuter en parallèle avec le premier parce que la zone de pile commune qu'ils emploient est renommée, donc dédoublée.
Dans le modèle parallèle, toutes les instructions sont disponibles d'emblée et chacune dispose d'un emplacement pour son résultat. À chaque cycle on exécute toutes les instructions prêtes, où qu'elles se situent dans la trace. Le modèle parallèle permet de mesurer le parallélisme initial de l'algorithme. Il donne en quelque sorte la réduction optimale du chemin critique.

2.3. La simulation des modèles

Pour simuler les deux modèles, nous avons utilisé PerPI [5]. PerPI est un outil PIN [7] développé par notre équipe de recherche. PIN permet d'analyser un code pendant son exécution, un peu comme on le fait avec un débogueur. L'outil PerPI compte les instructions exécutées et détermine pour chacune les différents cycles des étapes de son traitement pipeliné. À la fin, le plus grand cycle donne le temps d'exécution simulé. Le rapport du nombre d'instructions sur le temps donne l'ILP (Instruction-Level Parallelism) de l'exécution.
Pour déterminer le cycle d'exécution d'une instruction, PerPI se fonde sur les dépendances. Chaque instruction productrice marque ses destinations de son cycle d'exécution. Chaque instruction consommatrice lit l'ensemble des marques de ses sources. La plus grande marque lue fixe le cycle à la fin duquel toutes les sources sont prêtes.
PerPI calcule le graphe de dépendances (DDG) des instructions et en tire une plus longue chaîne (LDC). Si une exécution s'effectue en c cycles, une LDC se compose d'instructions dépendantes exécutées du cycle 1 au cycle c. Pour une exécution, il peut exister plusieurs LDC. PerPI n'en donne qu'une, la première qu'il trouve.

2.4. Un exemple

La figure 5 présente un code assembleur X86 de la réduction de somme. Il a été écrit à la main. L'exemple est représentatif de la parallélisation de l'extraction et de la gestion de la pile. En revanche, le programme étudié ne contient pas de boucle ni de chaînes de mouvements.
La figure 6 présente l'exécution de la fonction *somme* sur un vecteur de huit éléments. Il y a 93 instructions exécutées, découpées en trois tiers. Chaque instruction est identifiée par son numéro (première colonne de chaque tiers) conformément à la figure 5 et par son texte en as-

```
1  somme: pushq   %rbx                 //sauver t
2         pushq   %rbp                 //sauver n
3         subq    $8, %rsp             //allouer temp
4         cmpq    $2, %rbp             //n==2
5         jne     .L2                  //si (n!=2) vers .L2
6         movq    8(%rbx), %rax        //rax=t[1]
7         addq    (%rbx), %rax         //rax+=t[0]
8         jmp     .L3                  //vers .L3
9  .L2:   shrq    %rbp                 //rbp=n/2
10        call    somme                //somme(t,n/2)
11        movq    %rax, 0(%rsp)        //temp=somme(t,n/2)
12        leaq    (%rbx,%rbp,8), %rbx  //rbx=&t[n/2]
13        call    somme                //somme(&t[n/2],n/2)
14        addq    0(%rsp), %rax        //rax+=temp
15 .L3:   addq    $8, %rsp             //libérer temp
16        popq    %rbp                 //restaurer n
17        popq    %rbx                 //restaurer t
18        ret                          //somme(t,n) dans rax
```

FIGURE 5 – Une réduction de somme en assembleur X86

	instruction	f	xs	xp
1	pushq rbx	1	1	–
2	pushq rbp	1	2	–
3	subq $8,rsp	1	3	–
4	cmpq $2,rbp	1	1	1
5	jne .L2	1	2	2
9	shrq rbp	2	2f	1
10	call somme	2	4	1
1	pushq rbx	3	5	–
2	pushq rbp	3	6	–
3	subq $8,rsp	3	7	–
4	cmpq $2,rbp	3	3	2
5	jne .L2	3	4	3
9	shrq rbp	4	4f	2
10	call somme	4	8	1
1	pushq rbx	5	9	–
2	pushq rbp	5	10	–
3	subq $8,rsp	5	11	–
4	cmpq $2,rbp	5	5f	3
5	jne .L2	5	6	4
6	movq 8(rbx),rax	5	5f	1
7	addq (rbx),rax	5	6	2
8	jmp .L3	5	5f	1
15	addq $8,rsp	6	12	–
16	popq rbp	6	13	–
17	popq rbx	6	14	–
18	ret	6	15	–
11	movq rax,0(rsp)	7	16	3
12	leaq (rbx,rbp,8),rbx	7	15	3
13	call somme	7	16	1
1	pushq rbx	8	17	–
2	pushq rbp	8	18	–
3	subq $8,rsp	8	19	–
4	cmpq $2,rbp	8	14	3
5	jne .L2	8	15	4
6	movq 8(rbx),rax	8	16	4
7	addq (rbx),rax	8	17	5
8	jmp .L3	8	8f	1
15	addq $8,rsp	9	20	–
16	popq rbp	9	21	–
17	popq rbx	9	22	–
18	ret	9	23	–
14	addq 0(rsp),rax	10	24	6
15	addq $8,rsp	10	24	–
16	popq rbp	10	25	–
17	popq rbx	10	26	–
18	ret	10	27	–
11	movq rax,0(rsp)	11	28	7
12	leaq (rbx,rbp,8),rbx	11	27	3
13	call somme	11	28	1
1	pushq rbx	12	29	–
2	pushq rbp	12	30	–
3	subq $8,rsp	12	31	–
4	cmpq $2,rbp	12	26	3
5	jne .L2	12	27	4
9	shrq rbp	13	26	3
10	call somme	13	32	1
1	pushq rbx	14	33	–
2	pushq rbp	14	34	–
3	subq $8,rsp	14	35	–
4	cmpq $2,rbp	14	27	4
5	jne .L2	14	28	5
6	movq 8(rbx),rax	14	28	4
7	addq (rbx),rax	14	29	5
8	jmp .L3	14	14f	1
15	addq $8,rsp	15	36	–
16	popq rbp	15	37	–
17	popq rbx	15	38	–
18	ret	15	39	–
11	movq rax,0(rsp)	16	40	6
12	leaq (rbx,rbp,8),rbx	16	39	4
13	call somme	16	40	1
1	pushq rbx	17	41	–
2	pushq rbp	17	42	–
3	subq $8,rsp	17	43	–
4	cmpq $2,rbp	17	38	4
5	jne .L2	17	39	5
6	movq 8(rbx),rax	17	40	5
7	addq (rbx),rax	17	41	6
8	jmp .L3	17	17f	1
15	addq $8,rsp	18	44	–
16	popq rbp	18	45	–
17	popq rbx	18	46	–
18	ret	18	47	–
14	addq 0(rsp),rax	19	48	7
15	addq $8,rsp	19	48	–
16	popq rbp	19	49	–
17	popq rbx	19	50	–
18	ret	19	51	–
14	addq 0(rsp),rax	20	52	8
15	addq $8,rsp	20	52	–
16	popq rbp	20	53	–
17	popq rbx	20	54	–
18	ret	20	55	–

FIGURE 6 – Exécution de somme(t,8)

sembleur. La colonne f contient le cycle d'extraction de l'instruction dans le modèle séquentiel. Dans le modèle parallèle, toutes les instructions sont pré-extraites au cycle 0. Les deux colonnes xs et xp sont les cycles d'exécutions de l'instruction dans les modèles séquentiel et parallèle.
Par exemple, la 64ème instruction de la trace (la seconde à partir du haut de la colonne de droite) est l'instruction *jmp* apparaissant au rang 8 du *listing* de la figure 5. Cette instruction est extraite et exécutée au cycle 14 dans le modèle séquentiel. Le suffixe f indique que la dépendance critique vient de l'extraction : l'instruction s'exécute au cycle 14 parce qu'elle est extraite à ce cycle, alors que, ne dépendant d'aucune source, elle pourrait s'exécuter dès le cycle 1. Un tiret en colonne xp indique que l'instruction n'est pas traitée dans le modèle parallèle. Dans ce modèle, puisque les zones de stockage sont pré-allouées, il n'est pas nécessaire de gérer la pile. Les instructions de mises à jour du registre rsp (pointeur sur le sommet de pile) ainsi que les instructions d'empilements et de dépilements et de retours ne sont pas exécutées.
Le modèle séquentiel extrait la trace en 20 cycles pour 93 instructions, soit à peine un peu plus de quatre nouvelles instructions qui entrent à chaque cycle. Si on doublait la taille du vecteur à sommer, le nombre d'instructions exécutées doublerait et le nombre de cycles d'extractions doublerait aussi. Puisque moins de cinq nouvelles instructions sont fournies à chaque cycle, l'ILP ne peut être supérieur à 5.

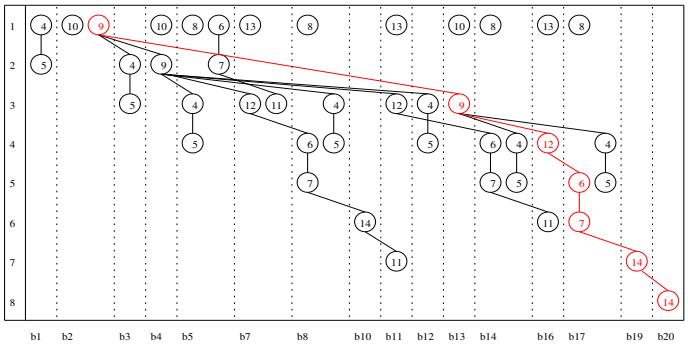

FIGURE 7 – Le graphe d'ordonnancement de somme(t,8) dans le modèle parallèle

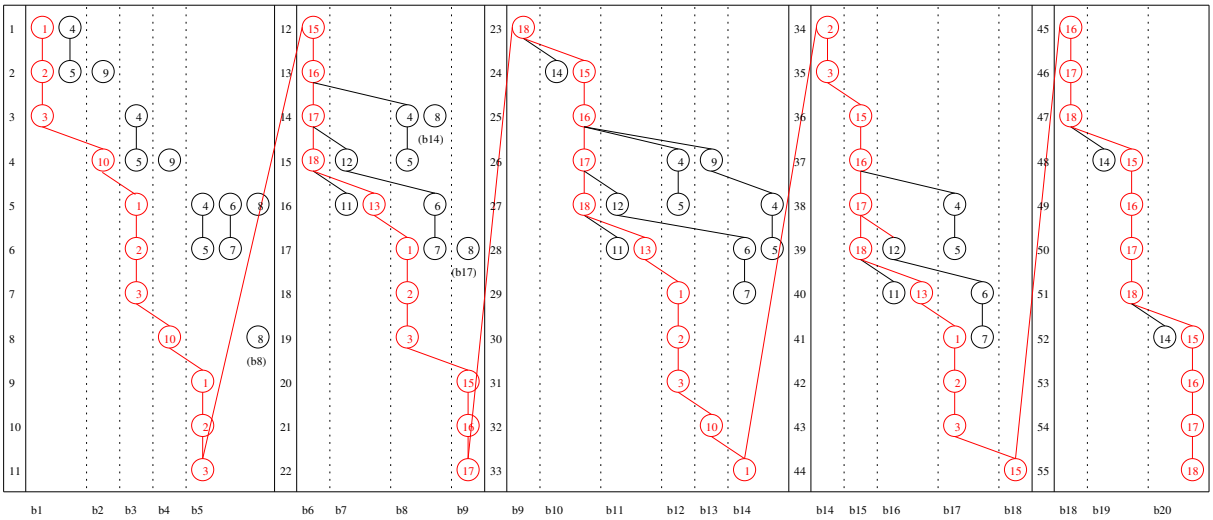

FIGURE 8 – Le graphe d'ordonnancement de somme(t,8) dans le modèle séquentiel

Le modèle séquentiel exécute le code en 55 cycles contre 8 pour le modèle parallèle. L'exécution dans le modèle séquentiel a une complexité linéaire par rapport au nombre d'éléments à sommer alors qu'elle est logarithmique pour le modèle parallèle.
Les raisons du meilleur comportement du modèle parallèle apparaissent quand on construit le chemin critique, c'est-à-dire la plus longue chaîne de dépendances (LDC).
La figure 8 montre le graphe d'ordonnancement des instructions pour leur exécution dans le modèle séquentiel. Le graphe est représenté par un tableau dont chaque ligne représente un cycle. Il y a 55 cycles répartis en cinq colonnes (colonnes des cycles 1 à 11, 12 à 22, 23 à 33, 34 à 44 et 45 à 55). Chaque colonne est subdivisée par des lignes pointillées. Les instructions d'une colonne pointillée forment un bloc de base (instructions contiguës ; les instructions 8 n'ont pas pu être placées dans leur bloc de base ; leur bloc est indiqué entre parenthèses). Les blocs de bases sont identifiés en bas de la figure par des numéros de b_1 à b_{20}. Par exemple, le bloc b_1 se compose des instructions 1 à 5 exécutées du cycle 1 au cycle 3. Le bloc b_8 regroupe les instructions 1 à 8, exécutées en désordre des cycles 8 à 19. L'instruction 8 de ce bloc n'est pas celle qui est exécutée au cycle 14 mais celle qui est exécutée au cycle 8.

En couleur (rouge) est représenté le chemin critique de cette exécution, formé par la plus longue chaîne de dépendances. Cette chaîne est entièrement constituée d'instructions manipulant implicitement ou explicitement le registre sommet de pile rsp.
La figure 7 présente le graphe d'ordonnancement des instructions dans le modèle parallèle.
Les blocs b_6, b_9, b_{15} et b_{18} n'ont pas d'instructions exécutées dans le modèle parallèle (gestion du sommet de pile). Le chemin critique se compose du calcul de $n = 2$ (instructions 9 de b_2 et b_{13} ; descente récursive), du calcul de l'adresse de t[6] (instruction 12 de b_{16}) et de son accès (instruction 6 de b_{17}), de la somme t[6]+t[7] (instruction 7 de b_{17}) et de sa propagation (instructions 14 de b_{19} et b_{20}) ; remontée). Cela coïncide avec la méthode diviser-pour-régner de l'algorithme initial.

3. Les simulations

3.1. Les benchmarks

Les deux modèles ont été appliqués à la suite *cBench* [4]. Des jeux de données réduits ont été utilisés pour garder des LDC de tailles raisonnables. La suite *cBench* est composée de codes séquentiels. Nos mesures montrent que malgré cela, certains codes sont parallélisables, étant issus d'algorithmes présentant un certain degré de parallélisme. D'autres en revanche ne le sont pas, parce que leur LDC se compose essentiellement de données et de résultats intermédiaires de l'algorithme. Pour paralléliser ceux-là, il faut utiliser un autre algorithme.

Benchmark	Nb séquentiel	Nb parallèle	Benchmark	Nb séquentiel	Nb parallèle	Benchmark	Nb séquentiel	Nb parallèle
automotive bitcount	67711586	65933642	consumer tiff2bw	65598581	65526353	security blowfish e	22903941	22701423
automotive qsort1	49925431	49925395	consumer tiff2rgba	59716238	59638722	security rijndael d	54595752	54243880
automotive susan c	59655992	59655922	consumer tiffdither	69142995	66224339	security rijndael e	52894747	52542875
automotive susan e	57930174	57930092	consumer tiffmedian	60418211	60362663	security sha	76609860	76217818
automotive susan s	64795722	64795638	network dijkstra	75713764	75463168	telecom adpcm c	67566059	67548239
bzip2d	63128426	63125570	network patricia	4697279	4690209	telecom adpcm d	67157842	67139830
bzip2e	56667030	56666160	office rsynth	42713214	42415228	telecom CRC32	13688337	13688309
consumer jpeg c	67603480	65690008	office stringsearch1	38880869	38768113	telecom gsm	57697117	57583295
consumer jpeg d	65142187	62343481	security blowfish d	41737062	41373216			

TABLE 1 – Les benchmarks de la suite *cBench*

La table 1 montre la liste des benchmarks retenus (certains ont été écartés parce qu'ils ne compilaient pas ou ne s'exécutaient pas). Pour chacun, la taille de l'exécution est donnée, qui comptabilise les instructions exécutées du code principal. Le code système est enlevé de la mesure pour ne pas la parasiter. Deux tailles sont mentionnées : dans le modèle parallèle, les instructions d'empilement et de dépilement ne sont pas comptabilisées.
Les *benchmarks* ont été compilés en optimisation -O1 pour éviter la vectorisation et la dérécursification de gcc. Les exécutions ont été simulées jusqu'à leurs termes.

3.2. Comparaison des LDC des deux modèles

3.2.1. Le cas d'un *benchmark* non parallélisable : crc32

Le *benchmark* crc32 effectue un calcul de somme de contrôle (*checksum*) pour la détection d'erreurs. La figure 9 montre un extrait de l'implémentation en C de crc32.

Le calcul dans la boucle *while* est séquentialisé par la récurrence sur la variable *oldcrc32*. La trace est composée majoritairement des instructions du corps de boucle (voir la figure 10).
L'exécution de crc32 compte 13.7M d'instructions, exécutées en 5M de cycles (ILP 2.8) dans le

```
#define UPDC32(octet,crc) (crc_32_tab[((crc)^((BYTE)octet)) \& 0xff] ^ ((crc) >> 8))
...
Boolean_T crc32file(char *name, DWORD *crc, long *charcnt) {
  register FILE *fin; ...
  while ((c=getc(fin))!=EOF){
    ++*charcnt;
    oldcrc32 = UPDC32(c, oldcrc32);
  } ...
} ...
int main(int argc, char **argv, int print){ ...
  errors |= crc32file(*++argv, &crc, &charcnt); ...
}
```

FIGURE 9 – Extrait de code C du *benchmark* CRC32

```
1 — 4007eb   add qword ptr [rbp], 0x1
2 — 4007f0   mov ecx, ebx
3 — 4007f2   xor ecx, eax
4 — 4007f4   movzx ecx, cl
5 — 4007f7   shr rbx, 0x8
6 — 4007fb   xor rbx, qword ptr [rcx*8+0x400a80]
7 — 400803   jmp 0x40080a
```

```
 8 — 40080a   mov rdi, r12
 9 — 40080d   call 0x400640
10 — 400812   cmp eax, 0xffffffff
11 — 400815   jnz 0x4007eb
12 — 22 = 1 — 11
23 — 33 = 1 — 11
34 — 45 = 1 — 12
```

FIGURE 10 – Extrait du code de CRC32 (syntaxe Intel)

modèle séquentiel. La LDC est composée d'un prologue, des instructions 2, 3, 4 et 6 répétées en boucle et d'un épilogue (la chaîne de couleur rouge sur la figure 11).
Dans le modèle parallèle la trace est la même mais les instructions de gestion de la pile ne sont pas comptées (*push*, *pop* et *rsp+/-=k*). Elle est exécutée en 2.5M de cycles (ILP 5.5). La LDC est une répétition des instructions 5 et 6 (voir la figure 12). Par rapport au modèle séquentiel, les instructions 2 et 4 (*mov*) sortent de la LDC, par renommage des mouvements. Il ne reste que deux instructions dépendantes par itération, soit 5 et 6. C'est cette amélioration qui réduit à elle seule le chemin critique de 50%. Néanmoins, la dépendance de donnée entre 5 et 6, qui provient de la méthode de calcul de la somme, lie toutes les itérations et empêche la parallélisation.

3.2.2. Le cas d'un *benchmark* parallélisable : patricia

Patricia est un parcours d'arbre utilisé en réseau. La figure 13 montre un extrait du code C.
Les itérations de la boucle *while* peuvent se faire en parallèle si le fichier est lu en parallèle parce qu'il n'y a pas de dépendances inter-itérations. La simulation ne trace pas les appels systèmes, si bien que les séquentialisations éventuelles de la fonction *fgets* ne sont pas considérées.
L'exécution dans le modèle séquentiel se fait en 500K cycles, pour 4.7M d'instructions (ILP 9.7), tandis que dans le modèle parallèle il ne faut que 48 cycles (ILP 100K).
La LDC en modèle séquentiel vient de l'extraction des instructions par bloc de base (à cause des innombrables sauts) et en partie de la gestion de pile (*call*, *ret*, *push* et *pop*, soit 10% de la LDC). Dans le modèle parallèle, la LDC vient de vraies dépendances entre registres.
La figure 14 compare les cycles d'exécutions de la dernière itération de la boucle *while*. La colonne f est le cycle d'extraction de l'instruction dans le modèle séquentiel (toutes celles d'un même bloc de base sont extraites en même temps). La colonne x est le cycle d'exécution dans le modèle séquentiel. La colonne xp est le cycle d'exécution dans le modèle parallèle.

4. Conclusion

L'exécution parallèle des programme se heurte à d'innombrables dépendances entre les instructions. La plupart sont d'origine architecturale. Quand on fait abstraction des dépendances

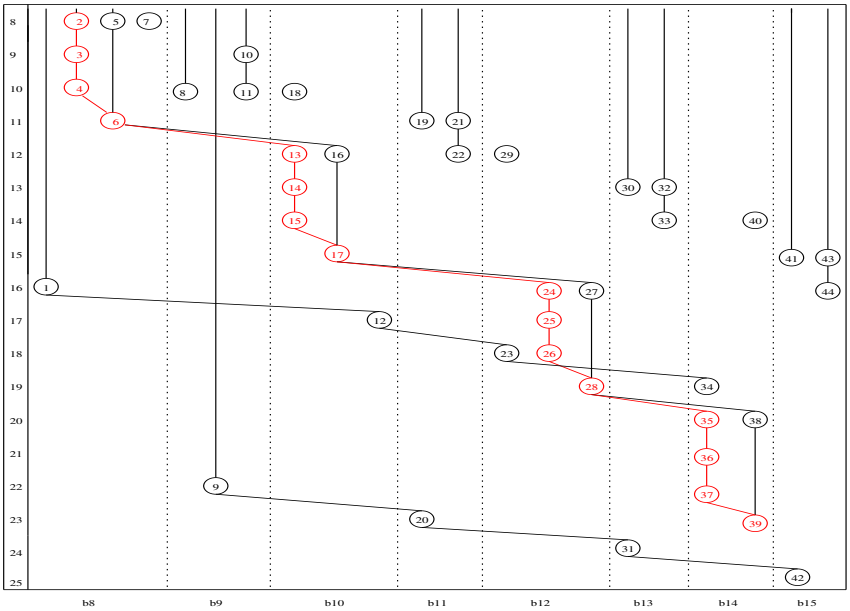

FIGURE 11 – Graphe d'ordonnancement des instructions pour l'exécution séquentielle de crc32

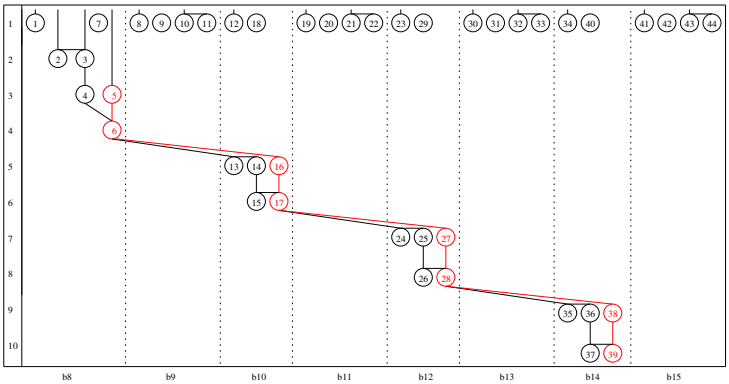

FIGURE 12 – Graphe d'ordonnancement des instructions pour l'exécution parallèle de crc32

```
int main1(int argc, char **argv, int print) {
  FILE *fp; ...
  while (fgets(line, 128, fp)) { ...
    pfind=pat_search(addr.s_addr,phead); ...
  } ...
}
```

FIGURE 13 – Extrait du code C du *benchmark* patricia

instruction		f	x	xp
400fc2	call 0x400d29	483754	483756	1
400d29	test rsi, rsi	483755	483755	2
400d2c	jz 0x400d8f	483755	483756	3
400d2e	mov rax, rsi	483755	483755	2
400d31	mov esi, 0x0	483755	483755	1
400d36	mov r9d, 0x1f	483755	483755	1
400d3c	mov r8d, 0x1	483755	483755	1
400d42	mov rdx, qword ptr [rax+0x8]	483755	483756	2
400d46	mov rdx, qword ptr [rdx]	483755	483757	2
400d49	and rdx, rdi	483755	483758	3
400d4c	cmp qword ptr [rax], rdx	483755	483756	2
400d4f	cmovz rsi, rax	483755	483757	2
400d53	movzx edx, byte ptr [rax+0x11]	483755	483756	2

instruction		f	x	xp
400d57	movsx ecx, dl	483755	483757	2
400d5a	mov r10d, r9d	483755	483756	2
400d5d	sub r10d, ecx	483755	483758	2
400d60	mov ecx, r10d	483755	483759	3
400d63	mov r10d, r8d	483755	483756	2
400d66	shl r10d, cl	483755	483760	3
400d69	movsxd rcx, r10d	483755	483761	4
400d6c	test rcx, rdi	483755	483762	4
400d6f	jz 0x400d77	483755	483763	5
400d77	mov rax, qword ptr [rax+0x18]	483756	483756	2
400d7b	cmp dl, byte ptr [rax+0x11]	483756	483757	3
400d7e	jl 0x400d42	483756	483758	3
400d42	mov rdx, qword ptr [rax+0x8]	483757	483757	3

FIGURE 14 – Extrait de la trace de patricia - comparaison des cycles d'exécutions de la dernière itération de la boucle *while*

de contrôle (en imaginant qu'on puisse extraire le programme par tous ses blocs de base en même temps), des dépendances liées à la pile (en imaginant qu'on puisse allouer autant d'emplacements pour les variables locales qu'il y a d'appels) et qu'on assigne une destination unique à chaque écriture (par renommage), la trace des instructions ne présente plus que les vraies dépendances, c'est-à-dire celles qui viennent des données de l'algorithme. S'il est parallèle par nature, les chaînes de dépendances sont courtes et le programme est parallélisable, sans retouche ni sur le code source ni sur le code machine.
Il reste à appliquer ces remarques sur un modèle de processeur dessiné pour l'élimination des dépendances architecturales.

Bibliographie

1. Austin (T. M.) et Sohi (G. S.). – Dynamic dependency analysis of ordinary programs. *SIGARCH Comput. Archit. News*, vol. 20, n2, 1992, pp. 342–351.
2. Bernstein (A.). – Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, vol. EC–15, n5, 1966, pp. 757–763.
3. Fahs (B.), Rafacz (T.), Patel (S. J.) et Lumetta (S. S.). – Continuous optimization. *SIGARCH Comput. Archit. News*, vol. 33, n2, 2005, pp. 86–97.
4. Fursin (G.) et Temam (O.). – Collective optimization : A practical collaborative approach. *ACM Trans. Archit. Code Optim.*, vol. 7, n4, 2010, pp. 20 :1–20 :29.
5. Goossens (B.), Langlois (P.), Parello (D.) et Petit (E.). – Perpi : A tool to measure instruction level parallelism. *In : PARA'2010 (1)*, pp. 270–281.
6. Hempel (R.). – The MPI standard for message passing. *In : High–Performance Computing and Networking, International Conference and Exhibition, Proceedings, Volume II : Networking and Tools*, éd. par Gentzsch (W.) et Harms (U.), pp. 247–252.
7. Luk (C.-K.), Cohn (R.), Muth (R.), Patil (H.), Klauser (A.), Lowney (G.), Wallace (S.), Reddi (V. J.) et Hazelwood (K.). – Pin : Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, vol. 40, n6, 2005, pp. 190–200.
8. Moore (G. E.). – Cramming more components onto integrated circuits. *Electronics*, vol. 38-8, 1965, pp. 114—-117.
9. Mueller (F.). – A library implementation of posix threads under unix. *In : Proceedings of the USENIX Conference*, pp. 29–41.
10. Wall (D. W.). – Limits of instruction-level parallelism. *In : Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188.