Reproductibilité des expériences de l'article "Analyse et réduction du chemin critique dans l'exécution d'une application"

Katarzyna Porada, David Parello, Bernard Goossens

▶ **To cite this version:**

Katarzyna Porada, David Parello, Bernard Goossens. Reproductibilité des expériences de l'article "Analyse et réduction du chemin critique dans l'exécution d'une application". 2017. lirmm-01600249

HAL Id: lirmm-01600249
https://hal-lirmm.ccsd.cnrs.fr/lirmm-01600249v1

Preprint submitted on 2 Oct 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Reproductibilité des expériences de l'article "Analyse et réduction du chemin critique dans l'exécution d'une application"

Katarzyna Porada and David Parello and Bernard Goossens

Univ. Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques, F-66860, Perpignan.
Univ. Montpellier II, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier, UMR 5506, F-34095, Montpellier.
CNRS, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier, UMR 5506, F-34095, Montpellier.
prenom.nom@univ-perp.fr

Résumé
Nous décrivons les étapes qui permettent de reproduire les résultats présentés dans l'article compagnon *Analyse et réduction du chemin critique dans l'exécution d'une application*, des mêmes auteurs et soumis à Compas'2014.

---

1. Introduction.

Nous décrivons les principaux éléments de l'installation, tout étant accessible à l'URL suivant, dans la partie Realis08 :

`http://perso.univ-perp.fr/david.parello/software/realis-2014/index.html`

Il faut récupérer les trois fichiers (*PerPI.tar.gz*, *cbench.tar.gz* et *realis-08.pdf*) dans un dossier *Realis-08*.
Pour reproduire les résultats de l'article, il faut disposer d'une machine x86_64. Un environnement linux est recommandé, en mode 64 bits (indispensable). Des détails supplémentaires sur certains logiciels sont ajoutés dans la suite.

2. Installer Pin.

Le logiciel Pin (outil gratuit Intel) est indispensable. Il est accessible à l'URL :

`http://software.intel.com/en-us/articles/pintool-downloads`

Il faut utiliser la version Linux 56759 du 20 janvier 2013.
L'installation de Pin se fait en deux commandes shell :

```
//On suppose que le dossier courant est celui où Pin est placé
$ cd source/tools/SimpleExamples
$ make dir obj-intel64/opcodemix.so
```

Selon la version de noyau Linux utilisée, il peut être nécessaire (c'est le cas avec les distributions Ubuntu) pour utiliser pin, de modifier une variable du système. Cela se fait ainsi :

```
$ echo 0 > zero
$ sudo cp zero /proc/sys/kernel/yama/ptrace_scope
```

On vérifie que l'installation de Pin est réussie en exécutant un exemple de Pintool qui écrit dans le fichier *opcodemix.out* le nombre d'instructions exécutées pour chaque *opcode*.

```
//On suppose que le dossier courant est
//pin-2.12-56759-gcc.4.4.7-linux (où Pin est placé).
//On effectue le test à partir de la commande /bin/ls appliquée
//au dossier courant.
$ ../../../pin -t obj-intel64/opcodemix.so -- /bin/ls
...
//Le fichier opcodemix.out contient le nombre d'instructions
//machines exécutées par /bin/ls, classées par catégories.
$ cat opcodemix.out
...
```

En cas d'erreur à l'exécution de Pintool :

```
$ ../../../pin -t obj-intel64/opcodemix.so -- /bin/ls
bash: ../../../pin: Aucun fichier ou dossier de ce type
```

Dans ce cas, il faut installer *ia32-libs*. Pour Ubuntu jusqu'à la version 13.04, cela se fait simplement en exécutant :

```
sudo apt-get install ia32-libs
```

Pour Ubuntu > 13.04, il faut éditer le fichier /etc/apt/sources.list (avec les droits d'administrateur) et rajouter en fin de fichier la ligne (ajout d'un dépot) :

```
deb http://archive.ubuntu.com/ubuntu/ raring main restricted universe multiverse
```

Puis il faut mettre à jour la liste des fichiers disponibles dans les dépôts APT (sudo apt-get update), et ensuite installer le paquetage (sudo apt-get install ia32-libs).

3. Installer PerPI.

Le logiciel PerPI est dans PerPI.tar.gz.
Après avoir déballé le dossier PerPI, il faut adapter le fichier PATH_TO_PERPI/makefile.inc en fixant les chemins d'accès à PerPI (PATH_TO_PERPI) et à Pin (PATH_TO_PIN).
La construction de PerPI se fait ainsi :

```
//On suppose que le dossier courant est PerPI
$ cd perpi_tools/ilptool-1.0
$ scons
```

Le dossier *obj-intel64* doit contenir quatre outils : ilptool-1.0_ilp_inscount, ilptool-1.0_ilp_nolimit, ilptool-1.0_ilp_seq, ilptool-1.0_ilp_speculative_fork.

4. Récupérer les *benchmarks* cBench

Les *benchmarks* cBench utilisés dans les expériences sont dans l'archive compressée cbench.tar.gz. Cette archive contient également les jeux de données et les scripts pour les expériences.

4.1. Configurer l'installation

Le script bash *configurer_variables_env* configure les chemin d'accès à PerPI, à pin et à cBench. Il doit être edité et adapté (étape indispensable pour la suite).
Le script bash *configurer* rend tous les scripts bash exécutables. Il doit lui-même être rendu exécutable :

```
//On suppose que le dossier courant est celui
//où est installé cBench (où se trouve le script configurer)
$ chmod u+x configurer
$ ./configurer
```

4.2. Compilation des *benchmarks* cbench et exécution de PerPI

Les mesures de PerPI sont basées sur le code machine exécuté. Le compilateur utilisé, la version, les options, le niveau d'optimisation impactent le code produit, donc le nombre d'instructions machines exécutées et leurs dépendances, c'est-à-dire le graphe d'ordonnancement partiel des exécutions.
Dans les expériences effectués pour l'article compagnon, les *benchmarks* ont été compilés avec option O1. Le script *compiler_tout* compile tous les benchmarks cbench :

```
//On suppose que le dossier courant est celui où est installé cBench
$ ./compiler_tout
```

A la fin de son exécution on devrait avoir une ligne indiquant *"26 benchs sur 26 ont été compilé(s) correctement"*. Dans le cas contraire, la suite peut être effectuée, mais la figure produite ne contiendra que le nombre de benchmarks indiqué.
La simulation des benchmarks dans le modèle séquentiel se fait en exécutant le script *lancer_seq_tout* :

```
//On suppose que le dossier courant est celui où est installé cBench
$ ./lancer_seq_tout
```

Pour leur simulation dans le modèle parallèle, on exécute *lancer_nolimit_tout* :

```
//On suppose que le dossier courant est celui où est installé cBench
$ ./lancer_nolimit_tout
```

Les deux scripts prennent environ 30 minutes (on peut les exécuter en même temps).
A la fin de cette étape, on peut vérifier que tous les résultats ont été produit en lancant le script *verif_fig2* :

```
//On suppose que le dossier courant est celui où est installé cBench
$ ./verif_fig2
//on doit avoir en sortie '52 fichiers trouvés'
//sinon, il faut attendre
```

Ensuite il faut exécuter le script *lancer_trace_benchs_article* qui construit les LDC des deux modèles (parallèle et séquentiel) pour les benchmarks patricia et crc32 :

```
//On suppose que le dossier courant est celui où est installé cBench
$ ./lancer_trace_benchs_article
```

Ce script produit des fichiers de traces d'exécution volumineux qui nécessitent environ 40Go d'espace disque. Il s'exécute en plusieurs heures (3 heures environ).
La vérification peut s'effectuer en lancant le script *verif_ldc* :

```
//On suppose que le dossier courant est celui où est installé cBench
$ ./verif_ldc
//on doit avoir en sortie '8 fichiers trouvés'
//sinon, il faut attendre
```

5. Obtention des figures

5.1. Figure 2 page 3 et Table 1 page 9

Pour obtenir la figure 2 page 3 (*Les LDC des modèles séquentiel et parallèle pour les benchmarks de la suite cBench*) et la table 1 page 9 (*Les benchmarks de la suite cbench*), il faut d'abord exécuter :

```
//On suppose que le dossier courant est celui où est installé cBench
$ ./recup_resultats
```

Ce script rassemble dans un fichier *table1_resultats* le nombre d'instructions exécutées par benchmark pour les deux modèles. Il range dans le fichier *resultats_bench* les longueurs des LDC.
La figure 2 est produite ensuite en exécutant (prérequis : gnuplot 4.6, qui doit disposer d'un terminal eps) :

```
//On suppose que le dossier courant est celui où est installé cBench
$ gnuplot 'gnuplot_figure2'
```

La figure est dans le fichier *figure_2.eps*. Pour passer du format eps au format pdf on peut utiliser epstopdf :

```
//Pour convertir le fichier "fichier.eps" au format pdf
$ epstopdf --outfile=fichier.pdf fichier.eps
```

5.2. Figures 5 à 8

Les figures 5 à 8 de l'article ont été produites à la main. Elle ne font donc pas partie de cette expérience de reproductibilité.

5.3. Figures 10 à 14

Les figures 10 et 14 présentent des codes assembleurs issus de la compilation des benchmarks CRC32 et patricia. Le code assembleur produit par gcc n'est pas portable, de sorte qu'il n'est pas possible de reproduire à l'identique les figures 10 à 14. Néanmoins, on peut retrouver les résultats de l'article à partir de codes présentant le même type de dépendances. La suite décrit comment analyser les résultats.
Le but de la figure 10 est de montrer que le benchmark CRC32 n'est pas parallélisable. La figure 10 contient la boucle qui forme le chemin critique de l'exécution de CRC32. Il n'est pas nécessaire de produire l'extrait de code pour observer ce qui est énoncé dans l'article :

```
L'exécution de crc32 compte 13.7M d'instructions, exécutées en 5M
de cycles (ILP 2.8) dans le modèle séquentiel.
```

Cette information (nombre d'instructions et nombre de cycles) est affichée par le script *crc32* :

```
//On suppose que le dossier courant est celui où est installé cBench
$ ./crc32
```

On doit constater que la LDC du modèle parallèle (nombre de cycles d'exécution) est du même ordre de grandeur que celle du modèle séquentiel (dans nos expériences, elle est deux fois plus courte), ce qui n'augmente pas sensiblement l'ILP (dans nos expériences, les ILP sont 2.8 dans le modèle séquentiel et 5.5 dans le modèle parallèle).
Les figures 11 et 12 comparent les LDC obtenues en modèle séquentiel et en modèle parallèle pour le benchmark *patricia*. Elles ont été construites à la main à partir d'une analyse du code assembleur de la figure 14. Il est difficile de construire des figures comparables pour le code x86 issu d'un autre compilateur sans connaître finement le langage machine et les dépendances entre instructions x86.
La figure 14 montre que les instructions critiques sont exécutées très précocement dans le modèle parallèle et très tardivement dans le modèle séquentiel, ce qui explique la très grande différence des ILP (9.7 contre 100K). Le script *figure14* construit un équivalent de la figure 14. Les instructions sont probablement différentes, ainsi que les cycles, mais la figure obtenue doit faire apparaître que l'exécution séquentielle est tardive à cause du *fetch* (les cycles d'exécution et les cycles de *fetch*, colonnes `x` et `f`, sont élevés) et l'exécution parallèle est précoce (colonne `xp`).
Pour obtenir la figure 14, il faut exécuter :

```
//On suppose que le dossier courant est celui où est installé cBench
$ ./figure14
//l'exécution nécessite environ 30 minutes
```

Le résultat est écrit dans le fichier *figure14_resultat* ainsi que sur la sortie standard.

6. Conclusion

La figure 2 page 3 doit montrer que le modèle séquentiel produit toujours des LDC très grandes (entre 500K et 2G instructions dans notre expérience). Par contre le modèle parallèle produit, pour les benchmarks les plus à droite sur la figure, des LDC courtes (inférieures à 10K).
La table 1 permet de se rendre compte de la taille des exécutions. Les exécutions en mode séquentiel ont un peu plus d'instructions qu'en mode parallèle (push et pop non comptés en mode parallèle).

La figure 10 doit faire ressortir que CRC32 n'est pas parallélisable. La LDC en mode séquentiel est longue et celle en mode parallèle est plus courte mais néanmoins longue aussi, ce qui résulte d'un ILP faible (inférieur à 10) dans les deux cas.
La figure 14 montre que l'exécution du morceau critique de code est très tardif dans le modèle séquentiel (cycle 480000 dans notre expérience) et très précoce dans le modèle parallèle (cycles 1 à 3 dans notre expérience).