Devoir de programmation multi-cœurs

Étudiants
Hervé BLOCIER
Damien GOMBAULT

18 mars 2009

Sommaire

1 Description de l'archive

- `doc/` : ce document et ses sources au format LaTeX
- `doxygen/` : la documentation Doxygen du projet
- `src/` : les sources C++ du projet
- `doxygen.conf` : le fichier de configuration pour la génération de la documentation Doxygen
- `Makefile` : le `Makefile` qui permet de compiler le projet
- `test.sh` : un script shell qui permet de tester le programme `strassen` avec les outils fournis avec le devoir

2 Compilation

Pour compiler le projet, il suffit simplement de taper la commande suivante à la racine du projet :

```
$ make
```

Par défaut, le support OpenMP est activé. Si vous souhaitez le désactiver, vous devez modifier la première ligne du fichier `Makefile` et remplacer `OPENMP=1` par `OPENMP=0`. Nettoyez ensuite les sources avec la commande `make clean` puis lancez de nouveau la compilation avec la commande `make`.

Après compilation, trois nouveaux exécutables sont créés.

`generate` Ce programme permet de générer de façon aléatoire une matrice et de l'enregistrer dans un fichier. Il est possible de choisir la taille de la matrice.

```
Usage: ./generate size outputfile
```

`benchmark` Ce programme permet de tester les performances de l'algorithme implémenté. Il génère des matrices de façon aléatoire et effectue leur multiplication en utilisant l'algorithme de Strassen. Il est possible de choisir la taille des matrices générées et le nombre de multiplications à effectuer.

```
Usage: ./benchmark size iteration
```

Il est recommandé d'utiliser la commande `time` interne au shell pour mesurer le temps d'exécution du programme :

```
$ time ./benchmark 256 1
./benchmark 256 1  4,87s user 0,03s system 96% cpu 5,104 total
```

`strassen` Ce programme effectue la multiplication de Strassen de deux matrices lues à partir de deux fichiers en entrée et enregistre le résultat dans un autre fichier.

```
Usage: ./strassen inputfile1 inputfile2 outputfile
```

3 Test

Pour tester le programme, il faut d'abord copier à la racine du projet les exécutables `mult` et `compare` fournis avec le devoir. Il suffit ensuite d'utiliser le script shell `test.sh` et lui spécifier une taille de matrice :

```
$./test.sh 256
Generating 256-sized matrix 1... Ok
Generating 256-sized matrix 2... Ok
Computing strassen algorithm... Ok
Computing normal algorithm... Ok
Comparing results... Ok
```

4 Implémentation

Nous avons développé plusieurs implémentations de l'algorithme de Strassen mais nous présentons uniquement notre version la plus performante qui est développée en C++ avec OpenMP. Nous avons également essayé de dérécursiver l'algorithme de Strassen mais nous ne sommes pas arrivé à des résultats satisfaisants. Nous avons également essayé d'implémenter une version en utilisant les *threads* POSIX.

Nous avons choisi le C++ afin d'obtenir une implémentation de haut niveau proche de l'algorithme original. Nous utilisons par exemple la surcharge des opérateurs `*`, `+` et `-`. Ceci nous permet d'avoir un code plus concis et facile à lire.

Le principal inconvénient des implémentations haut niveau est qu'ils ne sont généralement pas très performants. Nous avons donc cherché à obtenir un compromis intéressant entre haut niveau et performances. Nous avons donc cherché à optimiser cet algorithme en utilisant certaines fonctionnalités avancées du C++ comme les `rvalue references`[1]. Cette technique permet d'éviter un nombre important de recopies et ainsi augmenter de façon conséquente les performances. Une version récente du compilateur GCC est ainsi requise. Le projet a uniquement été testé avec la version 4.3 de ce compilateur.

Concernant la parallélisation, nous avons préféré utiliser OpenMP. Il nous permet de garder un code concis et facile à lire.

Nous avons tout d'abord tenté de paralléliser les petites opérations sur les matrices (les additions et les soustractions) mais cette version était beaucoup plus lente à cause de l'*overhead* provoqué par la création des *threads*. Le système passait beaucoup plus de temps à gérer les *threads* qu'à faire les calculs.

Nous avons donc changé de méthode et tenté de paralléliser l'exécution récursive de l'algorithme de Strassen. Ceci est permis grâce à l'utilisation du *nested parallelism* d'OpenMP. Nous avons limité ce parallélisme au niveau de la récursion par la formule $\frac{\log_2 \text{taillematrice}}{2}$. Cette technique nous permet d'obtenir un bon compromis entre le nombre de calculs effectués et le nombre de *threads* créés. Ceci permet ainsi d'augmenter les performances de l'algorithme.

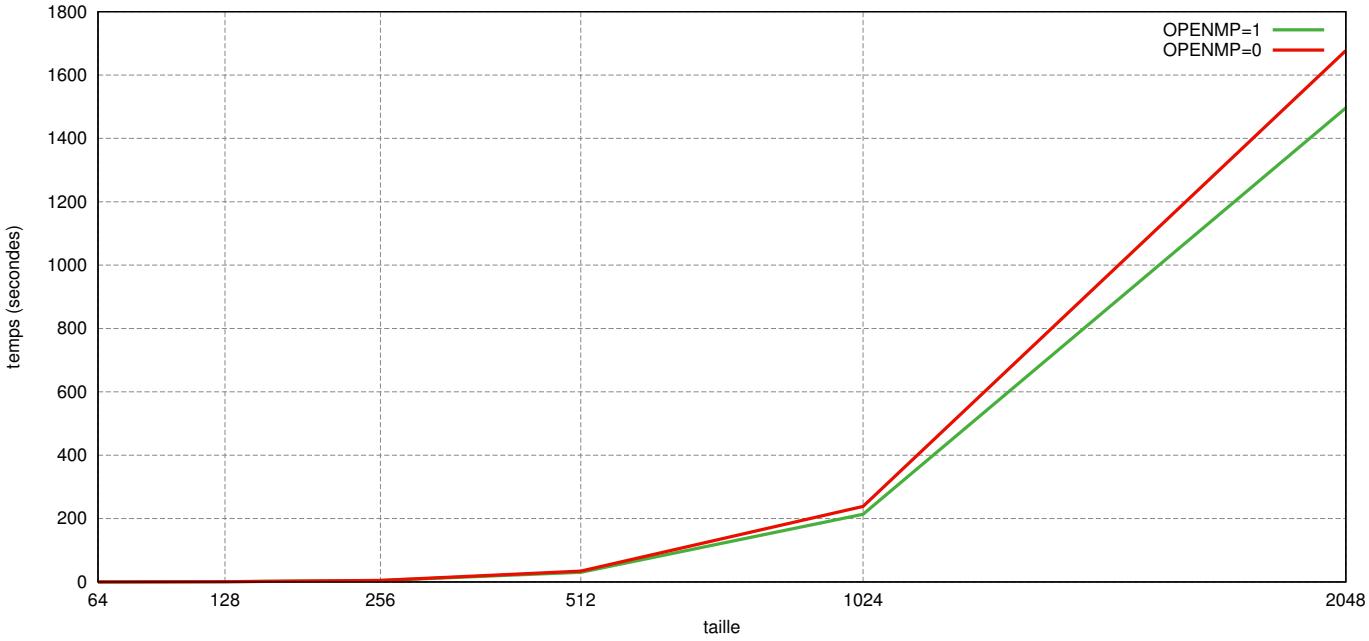
Nous pourrons dans un futur proche utiliser les tâches (*tasking*) d'OpenMP 3.0 à la place du *nested parallelism* d'OpenMP 2.5. Il s'agit d'une autre méthode plus performante pour exploiter le parallélisme imbriqué. Actuellement, cette fonctionnalité n'est pas implémentée dans GCC 4.3.

[1] http://www.artima.com/cppsource/rvalue.html

5 Performances

Nous avons effectué quelques tests de performance de notre projet. Nous avons lancé 5 exécutions du programme avec OpenMP puis sans OpenMP avec des tailles de matrices différentes et nous avons ensuite fait la moyenne des temps obtenus. Voici les résultats obtenus (en secondes) :

Taille	**64**	**128**	**256**	**512**	**1 024**	**2 048**
`OPENMP=0`	0,102	0,697	4,872	34,086	238,524	1 677,14
`OPENMP=1`	0,104	0,641	4,384	30,586	213,578	1 496,15
	-1,74%	8,67%	11,13%	11,44%	11,68%	12,10%



On peut donc remarquer que notre version OpenMP apporte un gain d'environ 11% par rapport à la version sans OpenMP. Nous pouvons également remarquer que le gain semble légèrement augmenter avec la taille de la matrice.

Nous avons effectué nos tests avec un Pentium Dual-Core T2080 (1,73 GHz / 1 Mio de Cache L2). Nous n'avons pas pu tester notre application avec plus de cœurs car nous n'avions pas de machine mieux équipées à disposition.