

TP5 Mandelbrot Set Computation with OpenMP

Michel Donnet

December 11, 2023

Contents

1	Introduction	2
2	Méthodologie	2
3	Résultats	4
4	Discussion	5
5	Conclusion	6

1 Introduction

L'objectif de ce travail pratique est de réutiliser le code parallélisé permettant de calculer l'ensemble de Mandelbrot, créé lors du travail pratique précédent, de sélectionner 2 régions de l'ensemble de Mandelbrot, de faire varier l'ordonnanceur entre statique et dynamique, ainsi que sa granularité pour les 2 régions choisies, et d'analyser les résultats obtenus.

Un ordonnanceur permet d'indiquer comment distribuer les itérations d'une boucle aux threads actifs. Les itérations sont regroupées en blocs avant d'être distribuées aux différents threads. La granularité de l'ordonnanceur donne le nombre d'itérations par bloc.

Dans le travail pratique, on va étudier les avantages et désavantages des ordonnanceurs statiques et dynamiques. Un ordonnanceur statique attribue les blocs entre les threads au moment de la compilation. Un ordonnanceur dynamique attribue les blocs entre les threads au moment de l'exécution.

2 Méthodologie

Voici la structure de mon code:

```
├── graph.py
├── Makefile
├── rapport.pdf
├── run.sh
└── src
    ├── dynamic.cpp
    ├── global.hpp
    ├── static.cpp
    ├── writer.cpp
    └── writer.hpp
```

Il y a un code qui permet de faire uniquement de l'ordonnancement statique (car cela se fait à la compilation) et un autre code qui permet de faire uniquement de l'ordonnancement dynamique (qui se fait à l'exécution).

J'ai défini la granularité de l'ordonnanceur dans le header `global.hpp`. C'est une variable constante afin que le compilateur puisse faire des optimisations.

Lorsque l'on compile le code, on a `tp6` qui est notre exécutable, et le résultat de notre fractal est enregistré dans un fichier `.bmp`, qui possède dans son nom le nombre d'itérations utilisé. J'ai réutilisé le `writer.cpp` du travail pratique 3 afin de créer le fichier `.bmp` pour visualiser le résultat obtenu.

Mon code est presque le même que le code rendu au travail pratique précédent, dupliqué en 2 fichiers permettant une exécution avec ordonnanceur statique (`static.cpp`), et une exécution avec ordonnanceur dynamique (`dynamic.cpp`).

C'est pourquoi, je vais uniquement expliquer les changements, et les grandes parties de mon code.

Mon code est divisible en 3 parties. Une première partie permettant de gérer les options données par l'utilisateur, une seconde partie permettant de calculer l'ensemble de Mandelbrot en section parallèle, et une dernière partie consistant à enregistrer dans un fichier `.bmp` les résultats obtenus, et à afficher le temps d'exécution, et d'autres informations, comme le nombre de threads utilisé, le type d'ordonnanceur utilisé...

On peut donner comme option le nombre d'itérations.

```
./tp6 -i 256
```

On peut donner comme option le nombre de threads actifs en section parallèle.

```
./tp6 -n 16
```

On peut donner comme option les coordonnées du `top left (tl)` et du `bottom right (br)` de la vue de notre fractal.

```
./tp6 -c -1 -1 1 1
```

Enfin, on peut afficher une aide:

```
./tp6 -h
Usage: ./tp6
        -i [iterations]
        -n [number of threads]
        -c [tl_x tl_y br_x br_y]
```

Si on donne aucun paramètre, le nombre d'itérations par défaut est 256, et la vue par défaut de notre fractal est `tl(0.3475, 0.3475)` et `br(0.36, 0.36)`. Le nombre par défaut de thread est donné par notre machine. La granularité de l'ordonnanceur est donnée dans le fichier `global.hpp`

On peut combiner les paramètres entre eux... Voici le résultat de cette exécution:

```
./tp6_static -n 4 -i 100
```

```
Execution time: 2.25746
Number of threads: 4
Number of iterations: 100
Schedule: static
Chunk size: 4
```

Voici comment l'ordonnanceur est traité:

```
#pragma omp for collapse(2) schedule(static, chunk_size)
```

Ici, j'indique que je désire avoir un ordonnanceur statique, avec une granularité donnée par `chunk_size`, définie dans `global.hpp`.

À la fin de mon code j'affiche également les informations concernant l'ordonnanceur.

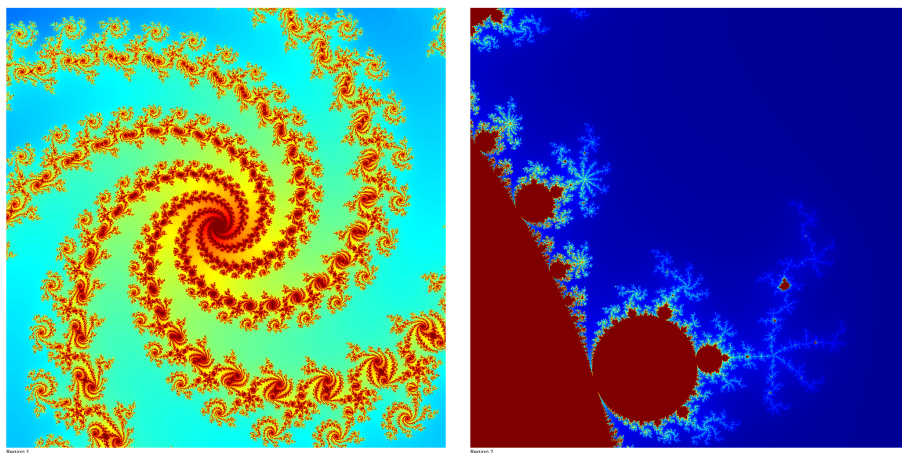
3 Résultats

J'ai exécuté mon programme en variant la région où on calcule notre fractal (en changeant les coordonnées de `top left` et de `bottom right`), et en variant le type et la granularité de l'ordonnanceur.

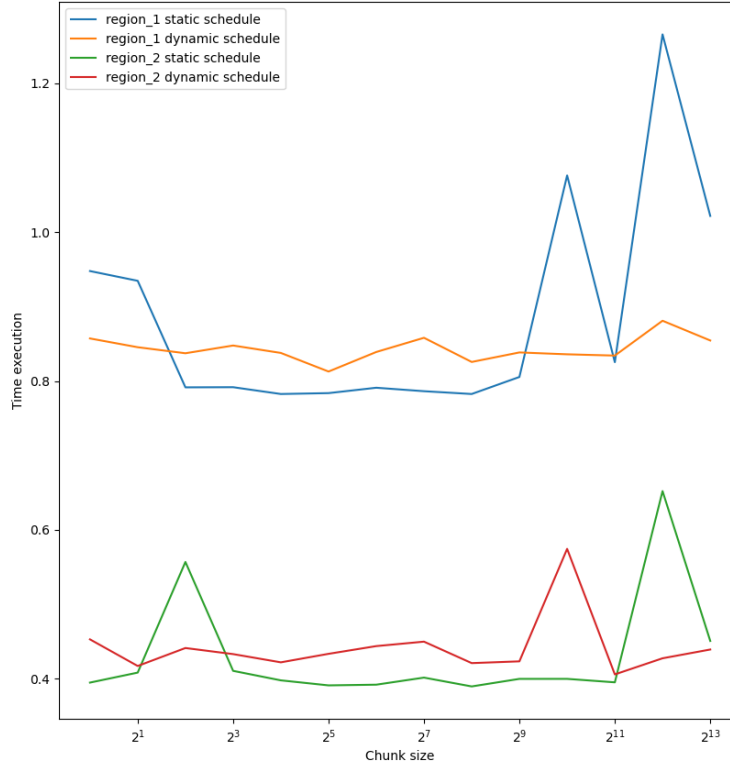
La première région est donnée par `tl(0.3602, 0.35656)` et `br(0.3612, 0.35756)`. Je l'ai choisie car le nombre d'itérations est relativement bien réparti: nous avons une charge de travail bien réparti sur toute l'image.

La deuxième région est donnée par `tl(0.3, 0.3)` et `br(0.5, 0.5)`. Je l'ai choisie car le nombre d'itérations n'est pas très bien réparti. En effet, on peut constater en regardant l'image que à gauche, on a un grand nombre d'itérations, et à droite de l'image, on a un faible nombre d'itérations...

Voici les deux régions que j'ai choisi pour exécuter mon code:



Et voici les résultats obtenus pour des exécutions avec 14 threads sur baobab, car un noeud possède 14 cpu dans baobab:



4 Discussion

Afin de pouvoir discuter plus facilement du comportement et des performances du programme, j'ai calculé un speed up empirique, permettant de déterminer l'efficacité de la parallélisation, que j'ai défini de la sorte:

$$S = \frac{T_{seq}}{T_{par}} \approx \frac{\text{Time execution with one thread}}{\text{Time execution (with one or more threads)}}$$

Cela me donne le graphique suivant:

Nous pouvons observer des choses intéressantes grâce aux deux graphiques.

5 Conclusion

Pour conclure, pour chaque problème donné, on a plusieurs défis à relever. Tout d'abord, on doit essayer de paralléliser le plus possible le code. Ensuite, lorsque le code est parallélisé, on doit trouver le nombre de threads maximisant l'efficacité de la parallélisation.

Il faut donc trouver le juste équilibre pour chaque programme entre nombre de threads utilisé et gain de performance.