Tutoriel sur la génération de graphiques et l'analyse des performances

1 Introduction

Une fois un noyau EASYPAP mis au point, il est légitime de souhaiter faire des expériences pour analyser le problème sous l'angle de la performance avec pour objectif de comprendre le comportement parallèle de l'implémentation et de trouver des pistes d'optimisation. Souvent on procède manuellement car on a hâte de voir ce que cela donne. Cependant, dès que l'on veut observer l'infuence de deux ou trois paramètres, il est préférable de les explorer à l'aide de *scripts* pour générer des données de façon systèmatique, les traiter et les visualiser afin de comprendre l'influence des différents paramètres.

Ces magnifiques graphiques pourront avantageusement agrémenter des rapports et présentations de projet en illustrant les résultats obtenus ou encore mettre en valeur des phénomènes remarquables que chacun sera bougrement [1] fier d'expliquer en s'aidant, par exemple, de traces d'exécution.

2 Un premier graphique vite fait

L'option `-n` d'EASYPAP permet d'enregistrer le temps de calcul pris par l'exécution du programme. L'outil EASYPLOT permet de tracer des graphiques tel celui présenté en Figure 1 à partir des données enregistrées. Voici comment fabriquer un tel graphique de simple façon :

```
rm -f plots/data/perf_data.csv # fichier utilisé par défaut pour stocker les mesures
for i in $(seq 1 12); do OMP_NUM_THREADS=$i ./run -k mandel -v omp_tiled -ts 32 -n -i 5; done
./plots/easyplot.py
```

En itérant plusieurs fois la boucle for dans le terminal et en relançant le logiciel EASYPLOT on obtient un second graphique (Figure 2) permettant d'apprécier la stabilité des performances.

Dans la suite du document on montre comment utiliser EASYPLOT dans une démarche expérimentale.

Tip

Au CREMI ajouter `source /net/ens/python/python3/python3.8.env` au fichier `~/.bashrc` pour utiliser un interprète python3 récent.

1. On peut dire aussi "*fier comme Artaban*"

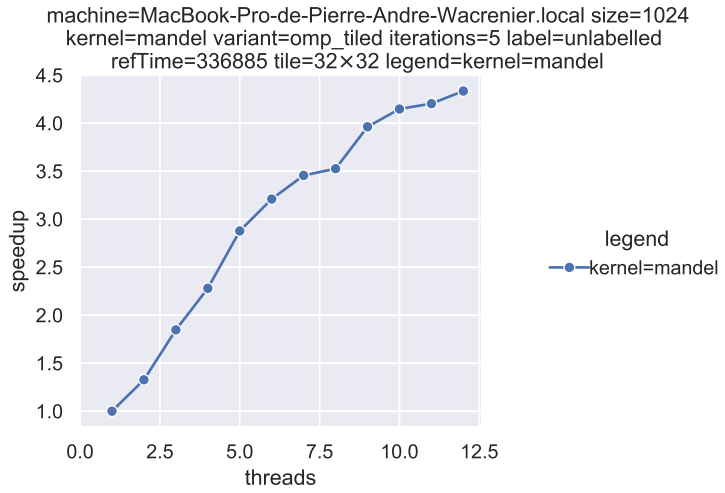


FIGURE 1 – Courbe de speedup `plot.pdf`. Les paramètres constants utilisés pour produire le graphique sont listés au dessus de celui-ci. En particulier on voit que le temps de rétérence utilisé pour calculer le speedup est de $336,885$ ms.

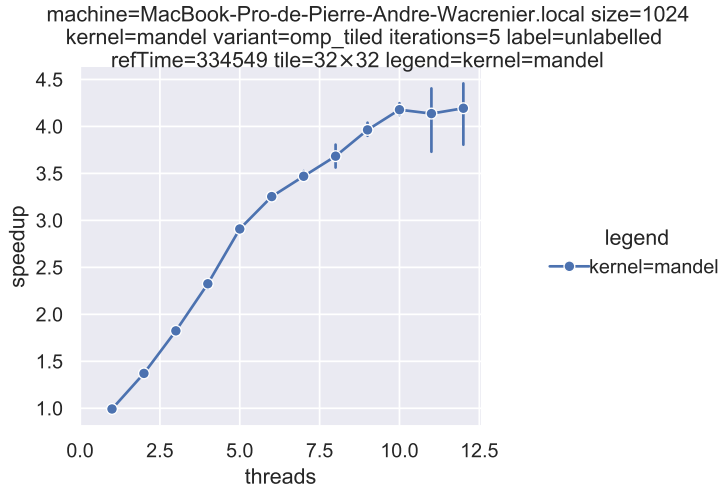


FIGURE 2 – Courbe de speedup avec barres de précision

-

3 Phase expérimentale

Il s'agit d'explorer le comportement d'un noyau de calcul en balayant un spectre de paramètres plus ou moins large pour repérer leur influence sur le temps d'exécution du programme. Pour cela on produit une base de données et on utilise des représentations graphiques comme outil de navigation dans ces données.

3.1 Génération des données

En début de campagne d'expérimentation, on ne cherche pas à obtenir des mesures statistiquement très précises, quelques mesures par points suffisent pour voir les grandes tendances. Un exemple de script pour le noyau `mandel` est donné Figure 3

```
#!/usr/bin/env python3
from expTools import *

# expériences avec des tuiles carrées
easypapOptions = {
    "-k ": ["mandel"],
    "-i ": [10],
    "-v ": ["omp_tiled"],
    "-s ": [512, 1024],
    "-ts ": [8, 16, 32],
    "--label ": ["square"]
}

# OMP Internal Control Variable
ompICV = {
    "OMP_SCHEDULE=": ["static", "static,1", "dynamic"],
    "OMP_NUM_THREADS=": [1] + list(range(2, 13, 2))
}

# Lancement des experiences
execute('./run ', ompICV, easypapOptions, nbrun=3, verbose=False, easyPath=".")

# expériences avec des lignes
del easypapOptions["-ts "]
easypapOptions["--label "] = ["line"]
easypapOptions["-th "] = [1]
easypapOptions["-tw "] = [32, 64, 128, 256, 512]

execute('./run ', ompICV, easypapOptions, nbrun=3, verbose=False, easyPath=".")
```

FIGURE 3 – Script pour générer des données expérimentales pour le noyau `mandel`

On lance ainsi le script :

```
machine;size;tilew;tileh;threads;kernel;variant;iterations;schedule;label;arg;time
MacBook;512;8;8;1;mandel;omp_tiled;5;static,1;square;none;90962
MacBook;512;16;16;1;mandel;omp_tiled;5;static,1;square;none;91548
MacBook;512;32;32;1;mandel;omp_tiled;5;static,1;square;none;89880
MacBook;1024;8;8;1;mandel;omp_tiled;5;static,1;square;none;334039
```

FIGURE 4 – Fichier texte `plots/data/perf_data.csv`

```
./plots/run-xp-mandel.py
```

On obtient un fichier texte au format CVS (`plots/data/perf_data.csv`) tel que présenté en Figure 4, c'est un banal fichier texte qu'on peut éditer à la main [2].

3.2 Production de graphiques pour analyser le problème

Le script python EASYPLOT permet de sélectionner des données et produire des courbes à partir d'un fichier CSV produit par EASYPAP. Ce script repose sur les bibliothèques de visualisation de données `Seaborn / Matplotlib` (http://seaborn.pydata.org) et sur la biblibiothèque d'analyse de données `pandas` (https://pandas.pydata.org). Les options du script reprennent celles d'EASYPAP pour la sélection des données et celles de `Seaborn / Matplotlib` pour la présentation graphique.

Ainsi pour sélectionner des lignes du fichier CSV, on utilise `--nom-du-champs` suivi de la liste des valeurs à conserver. Pour supprimer des colonnes du fichier, on utilise l'option `--delete liste-de-champs`. Par exemple :

```
./plots/easyplot.py  --kernel mandel --size 512 1024 --delete label
```

Par défaut EASYPLOT trace des courbes de speedup, mais on peut modifier la nature des axes à l'aide des paramètres `-x` et `-y`. L'axe des ordonnées sert aux performances ; au lieu du speedup, on peut utiliser la durée `-y time`, l'efficacité parallèle `-y efficiency` ou le débit, en terme de pixels par seconde, `-y throughput`. Pour l'axe des absisces on peut utiliser tous les autres attributs.

Tip

On peut choisir l'échelle logarithmique de base 2 pour les axes via les options `--xscale log2` et `--yscale log2`. Pour les autres options voir https://matplotlib.org/api/_as_gen/matplotlib.pyplot.xscale.html.

Notons que pour calculer le speedup EASYPLOT se base sur la durée de l'exécution séquentielle la plus courte pour une configuration donnée (taille du problème, nombre d'itérations,...). Pour tracer des courbes de speedup, il est donc nécessaire d'avoir dans la base de données quelques mesures où le nombre de threads utilisés vaut 1.

2. À condition de ne pas s'écarter de la nécessaire éthique scientifique. On ne truande pas les mesures quoi :)

Attention

On utilise la variable `OMP_NUM_THREADS` pour communiquer le nombre de thread à EASYPAP. Lorsque cette variable n'est pas positionnée, le nombre de cœurs de la machine est utilisé par défaut. C'est pourquoi il est nécessaire de positionner `OMP_NUM_THREADS=1` pour toute exécution séquentielle, même celles qui n'utilisent pas OpenMP. Voici un exemple d'appel à une variante `seq` :

```
  easypapOptions = {
    "-k ": ["mandel"],
    "-i ": [10],
    "-v ": ["seq"],
    "-s ": [512, 1024]
}
ompICV = {"OMP_NUM_THREADS=":[1]}
execute('./run ', ompICV, easypapOptions, nbrun=3)
```

Pour mesurer l'accélération d'une version séquentielle par rapport à une autre, on peut limiter la liste des variantes à considérer pour le calculer le temps de référence via l'option `--RefTimeVariants nom-de-la-variante`.

Tip

On peut être amené à mesurer le temps d'exécution d'un noyau jusqu'à sa terminaison car un même cas pourra nécessiter plus ou moins d'itérations selon la variante utilisée. Il s'agit alors d'oublier la colonne `iterations` pour tracer des courbes :

```
./plots/easyplot.py --kernel max --delete iterations
```

Mais l'utilisateur doit maintenant s'assurer que la base de données ne contiennent que des données cohérentes (eg. aucune mesure produite pour le kernel `max` avec l'option `-i` dans la base de données).

Le graphe produit, présenté en Figure 5, est illisible. Pour y voir plus clair on va utiliser des options EASYPLOT pour éliminer des informations et mettre en forme le graphique. Tout d'abord on crée des tableaux de graphes à l'aide des paramètres `--row` et `--col` pour séparer les différentes politiques de distribution et de tailles de l'image. Ensuite on supprime de la légende l'impression du champs `refTime` qui indique la durée utilisée pour calculer le speedup pour chaque courbe : cela nous permet de diviser par deux le nombre de courbes. On obtient ainsi la Figure 6a.

Sur le graphe 6a, on observe que le paramètre size ne semble pas prépondérant : les graphiques des cas 512 et 1024 se ressemblent. On va donc se concentrer sur le cas `size=512` pour deux raisons : il coûte moins cher à calculer et les écarts entre les courbes semblent plus marqués.

On produit alors la Figure 6b. Pour réduire plus encore l'information présentée, on sélectionne quelques courbes dont les meilleures et, pourquoi pas, les moins bonnes. On obtient alors un gra-

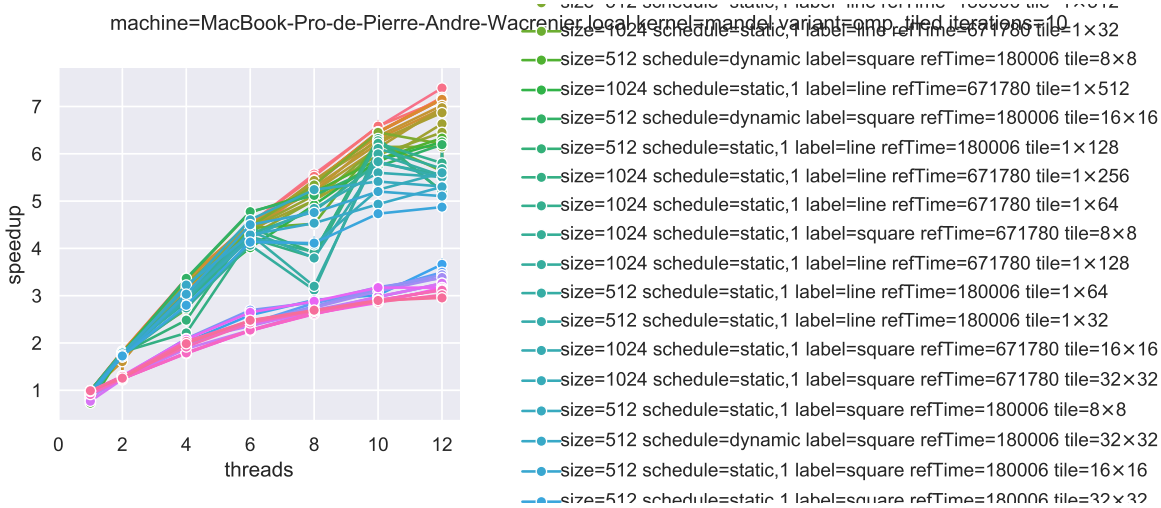


FIGURE 5 – Un beau bazar qui représente 1000 expériences réalisées en 4 minutes.

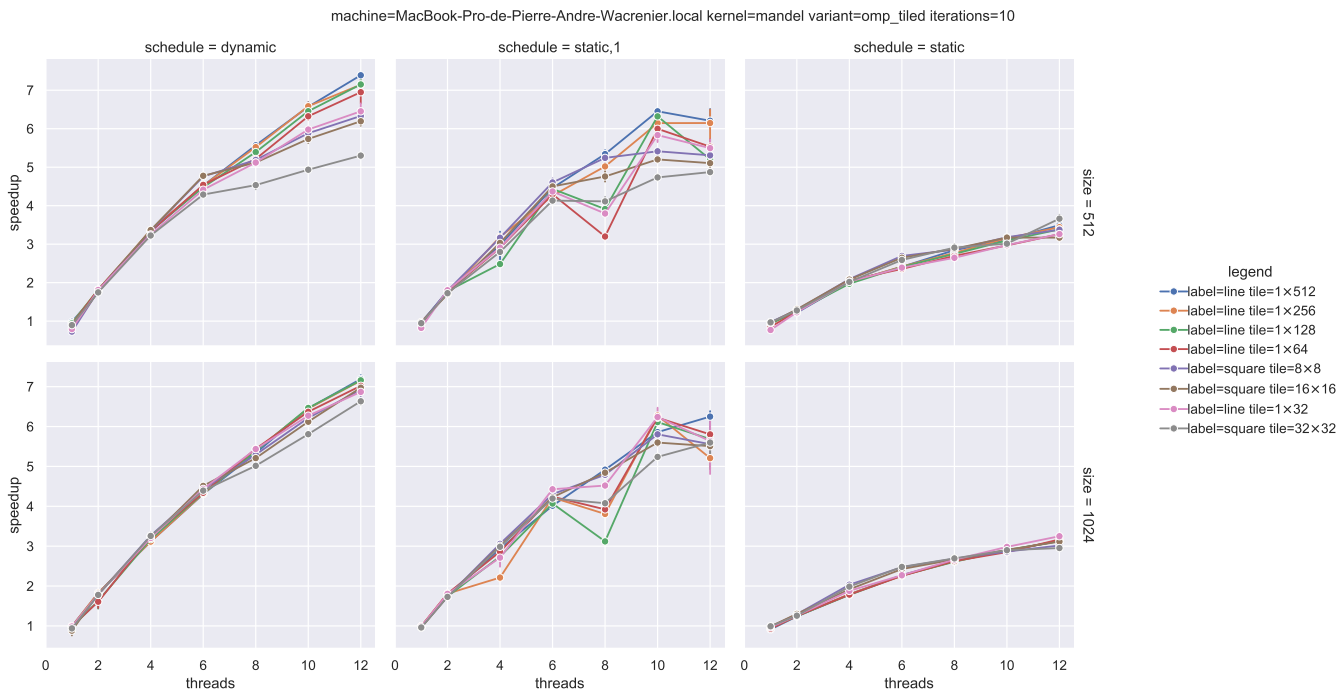
phique (Figure 7a) qui assez lisible. On pourrait même commenter ce graphique s'il était statistiquement solide. On va donc produire de nouvelles données avant de passer à la phase d'analyse. Pour mettre au point cette expérience on affine encore notre analyse des données. On remarque que les courbes `tile 1 x 32` et `tile 1 x 64` sont assez similaires, on sélectionne donc une des deux courbes. On observe des mesures parfois erratiques lorsqu'il y a plus de plus de 6 threads, on va donc ajouter des mesures pour 7, 9 et 11 threads. On relance maintenant les expériences sélectionnées de façon à avoir suffisamment de données pour faire des statistiques (30 données par exemple). On obtient la Figure 7b qui présente de nettes différences avec la figure précédente : des courbes sont plus irrégulières et trois stratégies obtiennent les meilleures performances au lieu d'une seule. Pour améliorer la lisibilité, on peut procéder à des retouches cosmétiques en augmentant la hauteur du graphe (figure 7c).

Tip

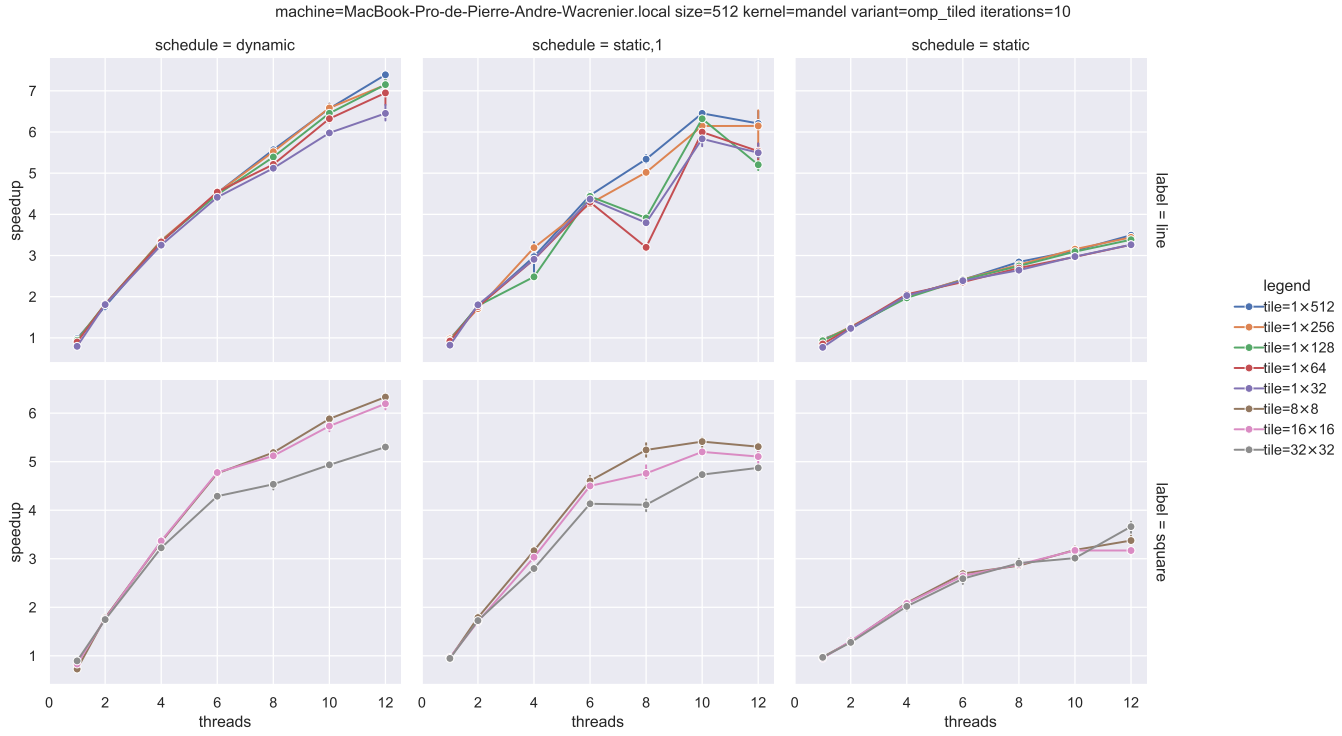
Les paramètres `--col`, `--row`, `--height` et `--aspect` reprennent ceux de la classe FacetGrid du module `seaborn`. La taille de la police est modifiable à l'aide du paramètre `--font_scale` et `--adjustTop` permet d'ajuster l'espace verticale pour entre titre le graphique. Le paramètre `legend_out` a été renomé `--legendInside`.

On peut ensuite vérifier Il y a quelques points étranges qui méritent une analyse :

— la courbe `1 x 64` pour 8 threads avec une distribution cyclique : pour en avoir le cœur net on peut zoomer sur ce point via l'option `--plottype catplot` qui permet de présenter les données sous forme d'histogrammes ou de nuages de points, comme dans la Figure 8. Comme le résultat est très sable, on peut utiliser une trace, présentée en Figure 9, pour observer le phénomène : sur la trace on constate que la version exploitant 8 threads souffre d'un mauvais équilibrage de charge qui n'apparait pas avec 7 threads.

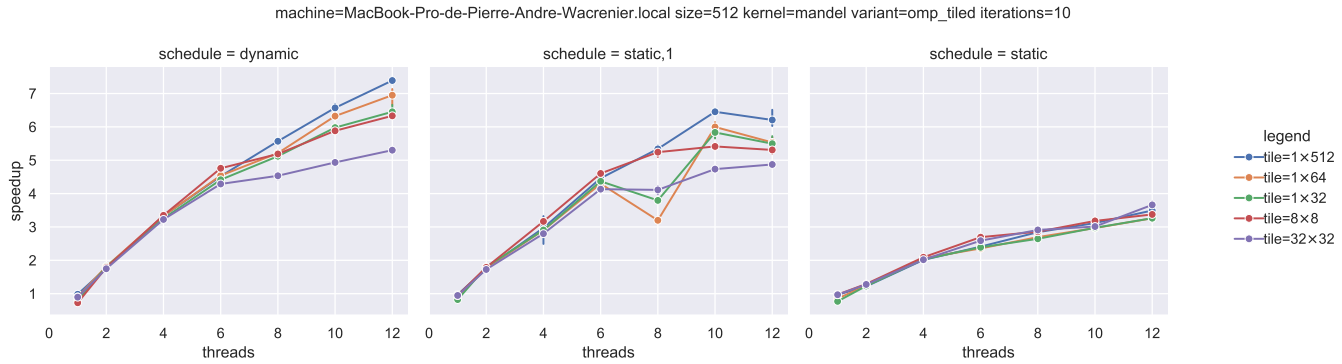


(a) `./easyplot --kernel mandel --col schedule --row size --noRefTime`

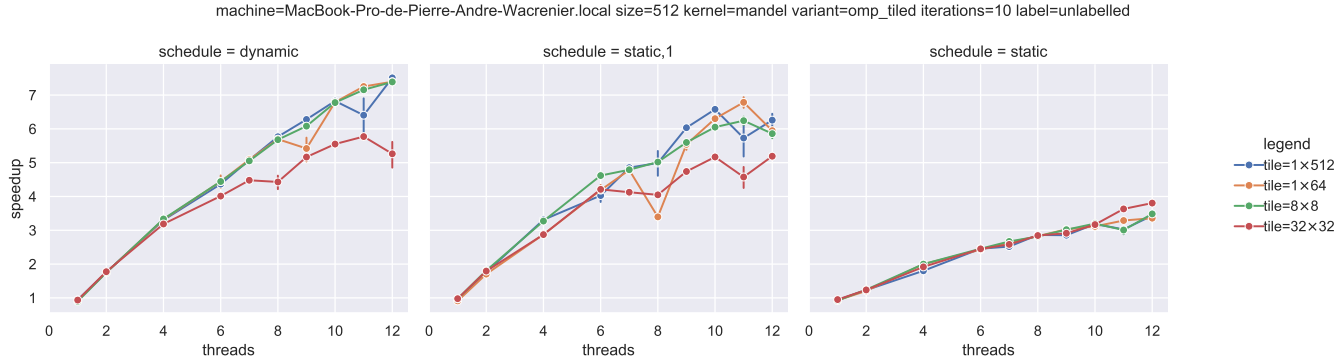


(b) `./easyplot --col schedule --row label --size 512`

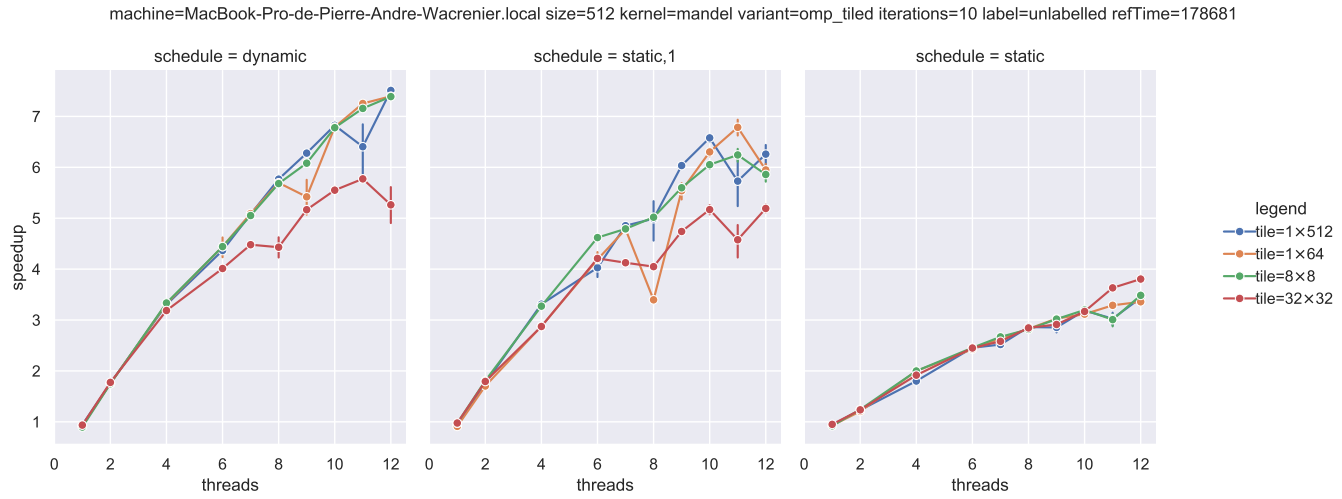
FIGURE 6 – Productions successives de graphiques pour démêler les courbes.



(a) `./easyplot --col schedule --delete label --size 512 -tw 8 32 64 512`



(b) Graphique produit à partir de 30 expériences par point, 7 minutes de calcul.



(c) `./easyplot.py --kernel mandel --col schedule --height 5 --aspect 0.8`

FIGURE 7 – Production d'un graphique statistiquement significatif à partir d'une nouvelle base de données.

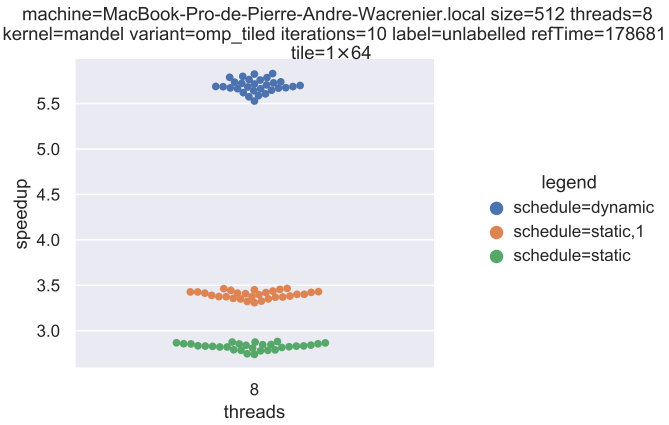


FIGURE 8 – `easyplot --kernel mandel --plottype catplot --kind swarm -tw 64 --threads`

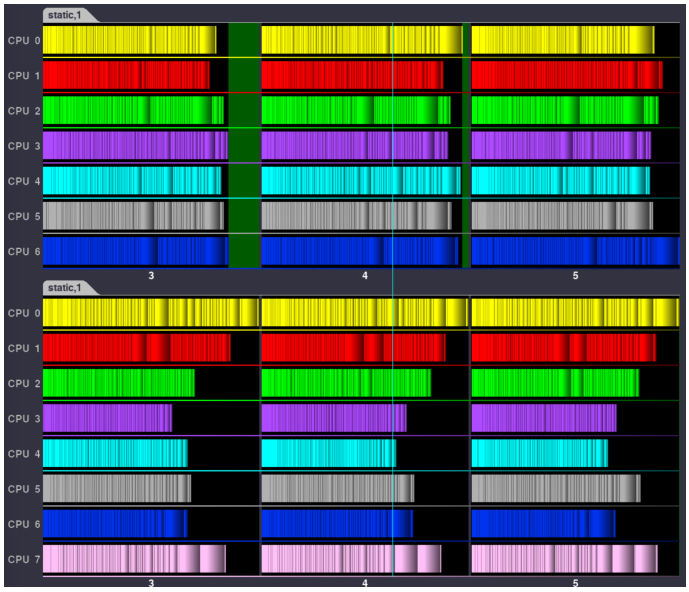


FIGURE 9 – Comparaison de traces pour le cas `OMP_SCHEDULE=static,1 ./run -k mandel -v omp_tiled -i 10` pour 7 et 8 threads.

— la courbe `1 x 512` pour 11 threads avec les distributions dynamique et cyclique présente des variations importantes. Sur la Figure 10, on voit des expériences dont le temps double. En réalisant, quelques expériences complémentaires à l'aide d'une boucle shell on arrive à reproduire le phénomène : les traces de la Figure 11 montre que la machine apparaît perturbée. Dans un rapport on mentionnera ce phénomène s'il est reproductible (comme ici) ou refera les expériences si le phénomène n'est pas reproductible. Bien entendu le mieux est d'utiliser une machine non chargée et d'avoir des courbes propres car il est inutile de commenter des courbes complétement brouillées.

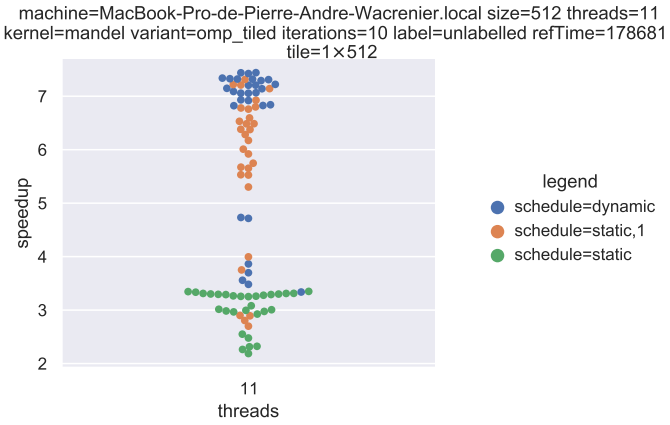


FIGURE 10 – `easyplot --kernel mandel --plottype catplot --kind swarm -tw 512 --threads`

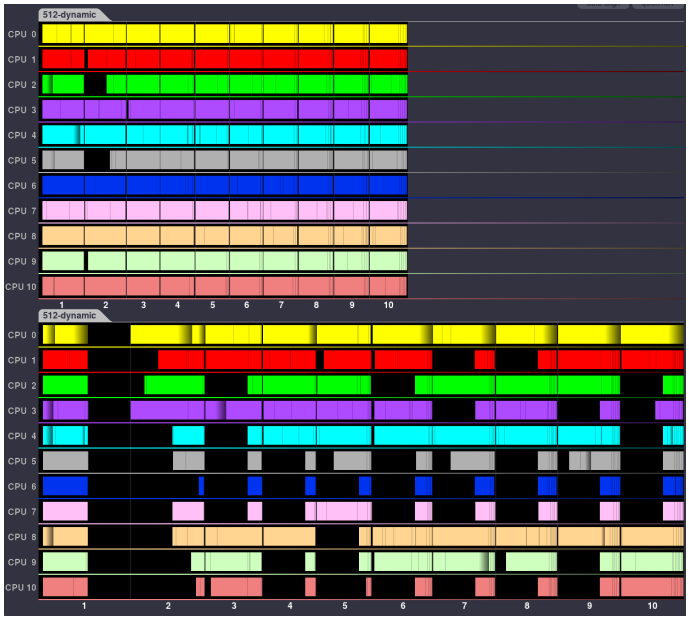


FIGURE 11 – Comparaison de deux traces pour le cas `OMP_SCHEDULE=dynamic ./run -k mandel -v omp_tiled -i 10` pour 11 threads.

Tip

Les valeurs du paramètre `--kind` associé au paramètre `--plottype catplot` sont décrites sous https://seaborn.pydata.org/generated/seaborn.catplot.html.

4 Présentation d'une expérience dans un rapport

4.1 Présentation du code

Dans le rapport il est nécessaire de présenter les parties importantes du code (e.g. figure 12) et d'apporter des éléments d'explication.

```
unsigned mandel_compute_omp_tiled (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {

    // On itére sur les coordonnées des do_tiles
#pragma omp parallel for collapse(2) schedule(runtime)
    for (int y = 0; y < DIM; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W)
        do_tile (x, y, TILE_W, TILE_H, omp_get_thread_num ());

    zoom ();
  }

  return 0;
}
```

FIGURE 12 – Variante du noyau `mandel`

4.2 Présentation des expériences

Il s'agit alors d'expliquer quelles expérimentations ont été menées et ce qu'elles ont apportées. Cela débouche sur quelques gaphiques. Chaque graphique doit être correctement légendé[3] et décrit de façon globale :

> Le graphique 13 présente les courbes de speedup obtenues par la variante `omp_tiled` du noyau `mandel` (cf. code en Figure 12) sur un processeur Core I9 à 6 cœurs (12 cœurs logiques). Trente mesures ont été faites par points. Les mesures sont stables jusque 6 threads mais sont parfois bruitées au delà en raison de préemptions de threads par le noyau.

3. Nous vous conseillons d'afficher les paramètres constants sur la figure, comportement par défaut d'EASYPLOT.

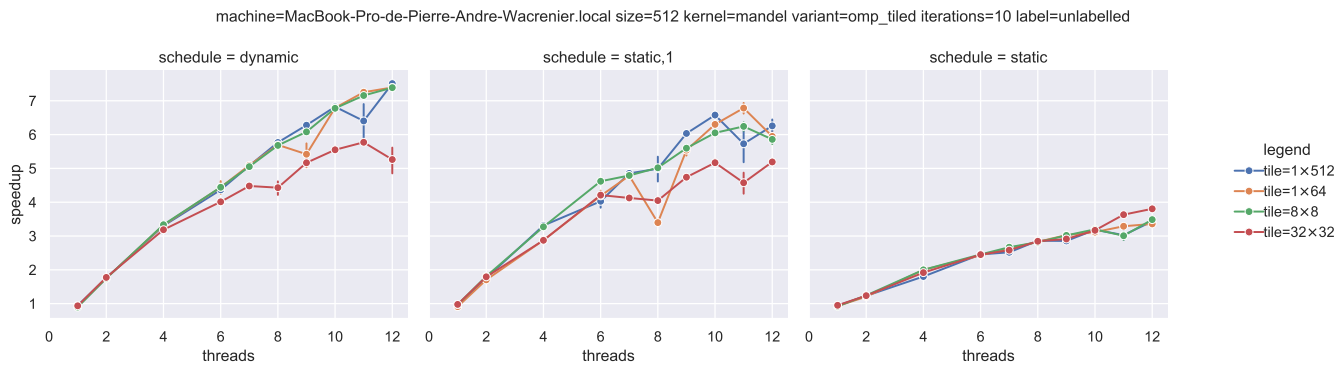


FIGURE 13 – Courbes de speedup d'une version tuilée du noyau mandel (processeur : 6-Core Intel Core i9)

4.3 Analyse et conclusion

Il s'agit ensuite de décrire les courbes et d'expliquer le comportement observé : pourquoi un speedup de 7 sur une machine à 6 cœurs ? Pourquoi un meilleur speedup avec un grain plus fin ? Pour cela on peut s'aider de traces (figure 14) que l'on peut également présenter et décrire. On peut s'attaquer à des cas particuliers comme nous l'avons fait en section précédente. Enfin on peut conclure :

> Nous avons observé que le temps nécessaire au calcul d'un pixel du nuage de Mandelbrot dépend de ses coordonnées et qu'il existe des zones de l'image bien plus coûteuses que d'autres à calculer (facteur 1000). Cela entraine un mauvais équilibrage de charge entre les threads lorsqu'on emploie une répartition statique. On constate cependant que les distributions bloc-cycliques atteignent de meilleures performances que les distributions par bloc car la charge est, en moyenne, bien mieux répartie. Au final c'est la politique de distribution dynamique qui apporte le plus de perfomance et le plus de stabilité dès que la granularité employée est suffisante pour équilibrer la charge.

4.4 Toujours plus d'expériences

Il est possible d'investiguer différents champs, par exemple nous avons rapidement évacué le paramètre taille de l'image. Le graphique en Figure 15, tiré du papier `https://hal.archives-ouvertes.fr/hal-03126887v1` montre que ce paramètre peut être intéressant à examiner. Vous pouvez consulter ce papier pour voir quelques traces et graphiques et comment nous essayons de mettre en valeur notre travail.

4.5 Un mot sur les traces

Nous vous recommandons de vous appuyer sur des traces d'exécution dans vos démarches d'optimisation et d'analyse. Par exemple, sur la trace de la figure 16 on observe des périodes d'inactivité entre les tâches exécutées par un thread au sein d'une itération. Il s'agit d'un goulet d'engorgement au niveau de la distribution des tâches. Dans la trace de la figure 17, deux traces

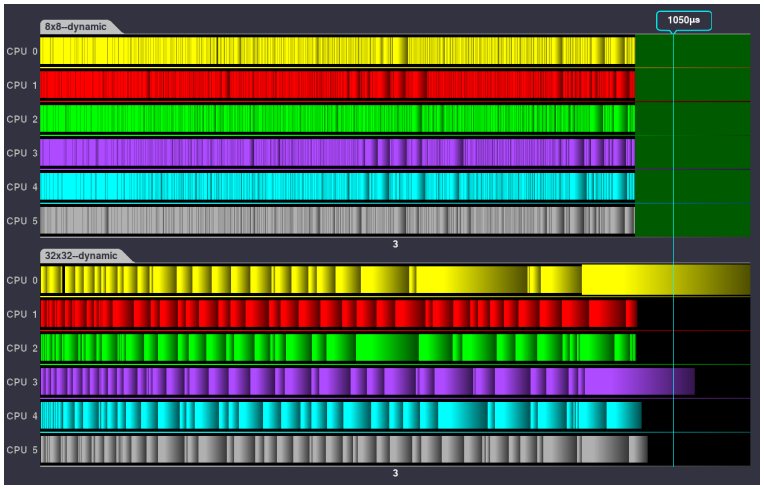


FIGURE 14 – Noyau `mandel` : comparaison de traces d'exécution (variante `omp_tiled`, tuiles 8×8 vs 32×32, distribution dynamique)

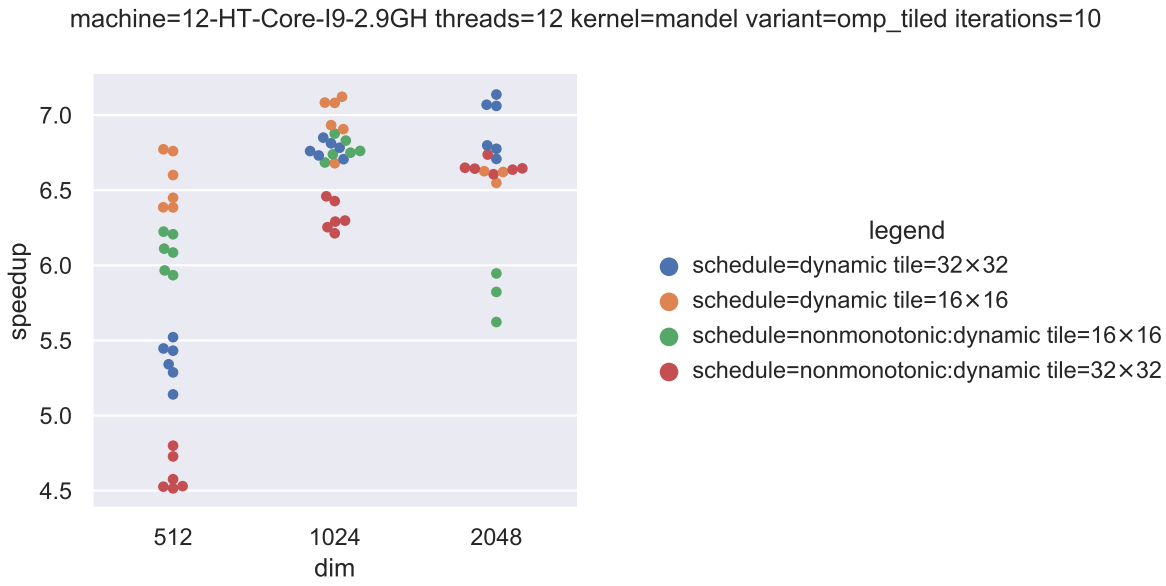
sont comparées : une trace séquentielle et une trace parallèle. La durée des tâches calculées en concurrence est nettement supérieure à à la durée des tâches traitées en solitaire sur la machine. Il s'agit très probablement d'un phénomène de contention au niveau du calcul (mutex, variable atomique, variable partagée, faux partage).

5 Personnalisation

Il est possible d'adapter le style de graphique en modifiant l'appel à `sns.set()` dans le script EASYPLOT (Figure 18). De même il est possible de modifier la base de données en sélectionnant certaines lignes avec des expressions complexes ou encore en y ajoutant des champs calculés. Pour cela il est peut être utilise consulter les pages suivantes :

— https://matplotlib.org/tutorials/introductory/customizing.html,
— https://pandas.pydata.org/docs/getting_started/intro_tutorials/03_subset_data.html.
— https://pandas.pydata.org/docs/getting_started/intro_tutorials/05_add_columns.html.

FIGURE 15 – Influence de la taille des tuiles selon la taille du problème .



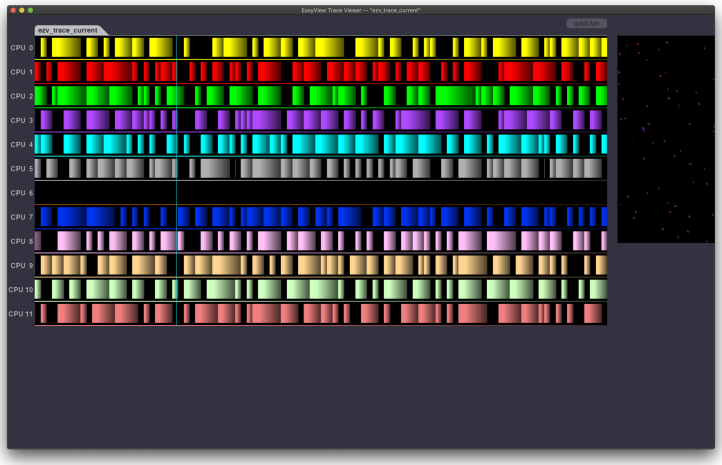


FIGURE 16 – Noyau mystère : congestion au niveau de la distribution des tâches

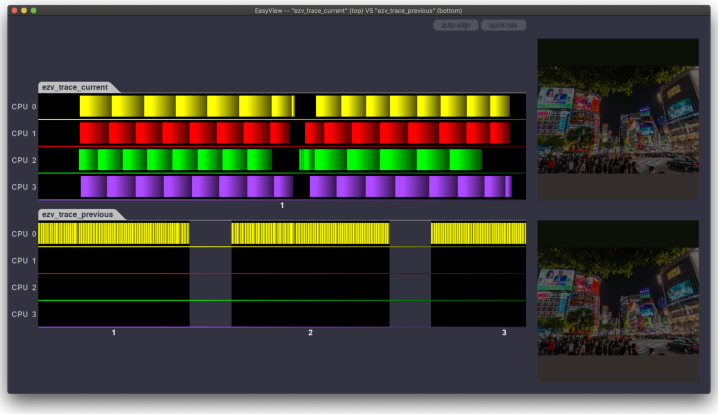


FIGURE 17 – Noyau Mystère : Congestion interne aux tâches.

```
  #!/usr/bin/env python3
from graphTools import *
import sys

args = parserArguments(sys.argv)
df = lireDataFrame(args)

# see customizing-with-matplotlibrc-files
# https: // matplotlib.org/tutorials/introductory/customizing.html
sns.set(style="darkgrid", rc={'text.usetex': False,
                              'legend.handletextpad': 0,
                              'figure.titlesize': 'medium'})

# Selection des lignes :
# df = df[(-df.threads.isin([8])) & (df.kernel.isin(['mandel']))].reset_index(drop = True)

# Creation du graphe :
fig = creerGraphique(df=df, args=args)
fig.tight_layout()
engeristrerGraphique(fig)
```

FIGURE 18 – Script EASYPLOT