M2 E3A - SETI

TP OpenMP CPU

SAŠA RADOSAVLJEVIC

13 FÉVRIER 2023

Table des matières

Table des matières **1**

1 TP OpenMP **2**
1.1 Vérification des capacités systèmes 2
1.2 Opérations sur les éléments d'un tableau 3
1.3 Somme des éléments d'un tableau 4
1.4 Détection des angles par la méthode de Shi-Tomasi 8
1.4.1 Analyse du code 8
1.4.2 Analyse de l'influence des paramètres 8
1.4.3 CPP 8
1.5 Conclusion 11

1 | TP OpenMP

Le TP est réalisé sur un ordinateur portable personnel Prestige 15 A10SC ayant les caractéristiques suivantes :

- Intel Core i7-10710U (Hexa-Core 1.1 GHz / 4.7 GHz Turbo - 12 Threads - Cache 12 Mo)

Pendant l'exécution d'un programme, l'horloge se stabilise à 4 GHz.

1.1 - Vérification des capacités systèmes

En utilisant les fonctions OpenMP pour vérifier le nombre de processeurs et de threads max suivant le code :

```c
#include "stdio.h"
#include <stdlib.h>
#include <stdint.h>

#include <omp.h>

int main(int argc, char **argv)
{
    int nb_thread = omp_get_max_threads();
    int nb_proc = omp_get_num_procs();

    printf("Nb thread max = %d\n\r", nb_thread);
    printf("Nb procs = %d\n\r", nb_proc);
    return 0;
}
```

On obtient :

```
Nb thread max = 12
Nb procs = 12
```

Ce qui correspond bien au nombre de coeur logique et une limite de un thread par coeur logique.

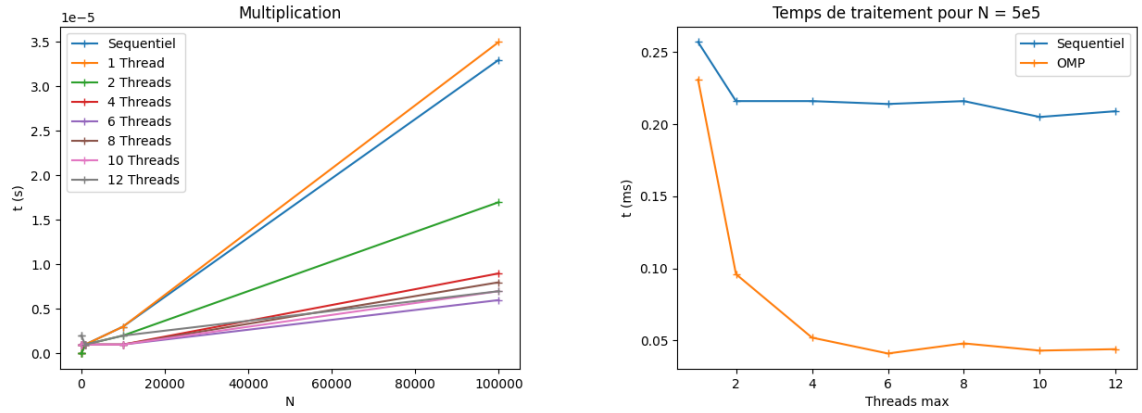
En utilisant le changement de variable environnement `OMP_NUM_THREADS=6; export OMP_NUM_THREADS`, le programme renvoie pour Nb thread max, 6.

1.2 - Opérations sur les éléments d'un tableau

Dans cette partie, on travaille avec un tableau de dimension N alloué dynamiquement. Ce nombre et le nombre de threads utilisés seront passés en paramètres de la fonction via `./exo2 N T`. Le programme est accompagné d'un script bash et d'un script python pour générer les graphiques ci-dessous. On analysera le temps d'exécution des opérations sur des flottants en fonction du nombre de threads utilisés.

Le premier calcul est une multiplication figure 1.1 dont le temps d'exécution augmente linéairement en fonction de N. L'augmentation du nombre de threads devient inefficace à partir de 6 threads sur la rapidité d'exécution. Le petit saut entre 1 et 2 threads en séquentiel correspond à la montée en fréquence du CPU qui impacte les premières itérations.

La mesure du temps d'exécution est effectuée sur une moyenne d'une boucle de 10 000 itérations.



(a) Fonction de N

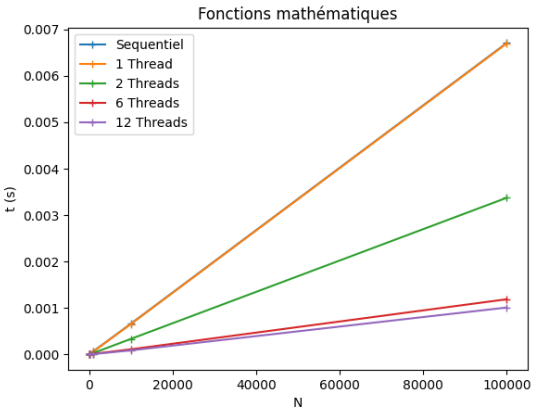
(b) Fonction du nombre de threads pour N = 5e5

FIGURE 1.1 – Temps d'exécutions pour la multiplication

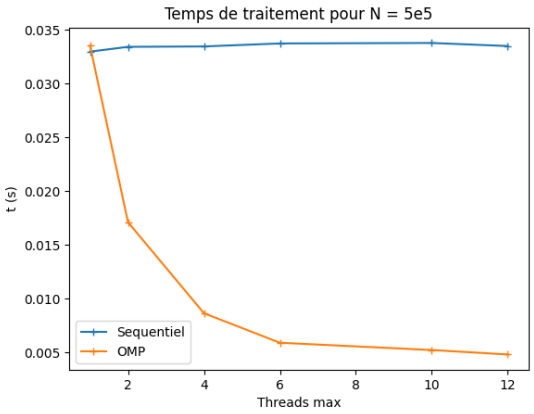
On remarque la même tendance pour l'exécution des fonctions mathématiques figure 1.2.

Les fonctions sont données par les codes suivants :

```
1 for(int i=0; i<N; i++){
2     y[i] = 2.27*x[i];
3 }
```



(a) Fonction de N



(b) Fonction du nombre de threads pour N = 5e5

FIGURE 1.2 – Temps d'exécutions pour les fonctions mathématiques

```
for(int i=0; i<N; i++){
    y[i] = 2.27*log(x[i])*cos(x[i]);
}
```

On obtient une accélération de 4,75 pour la multiplication et 6,94 pour les fonctions mathématiques.

1.3 - Somme des éléments d'un tableau

On souhaite analyser un programme effectuant la somme en parallèle des éléments d'un tableau x de taille N rempli de flottants aléatoires correspondant au code suivant :

```
double y = 0;
for(int i=0; i<N; i++){
    y = y + x[N];
}
```

La réduction utilisée est `#pragma omp parallel for reduction(+:y)`. Les résultats sont présents sur la figure 1.3. On observe une augmentation du temps d'exécution linéaire en fonction de N. On pourra par la suite étudier seulement le cas $N = 500000$.

Pour le scheduling, la parallélisation de base correspond à `schedule(static,1)`. A partir de cette configuration de base, on peut comparer avec plusieurs cas :

- `schedule(static,1)`
- `schedule(static,4)`

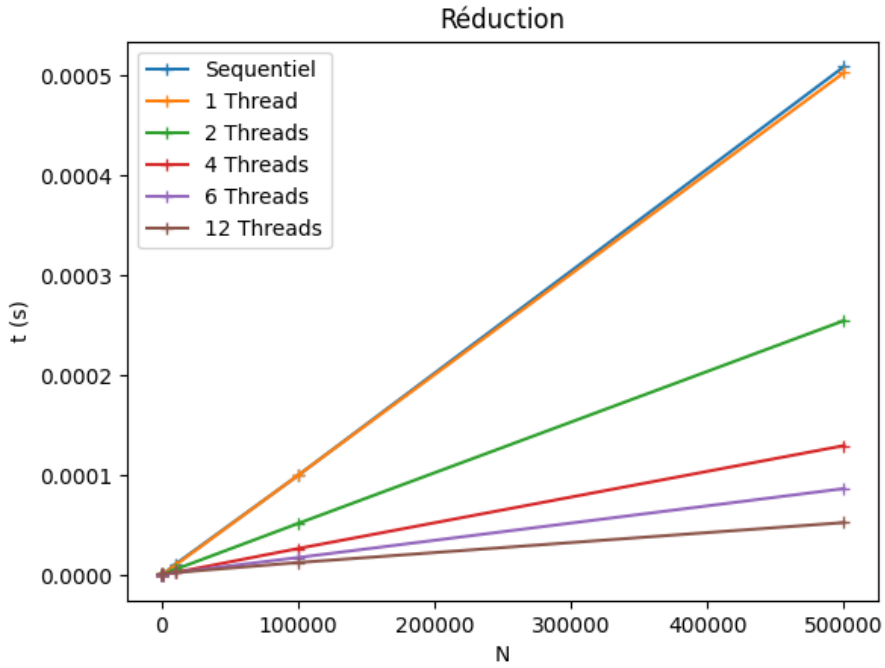


FIGURE 1.3 – Parallélisation par réduction

- `schedule(static,12)`
- `schedule(static,100)`
- `schedule(dynamic,1)`
- `schedule(dynamic,4)`
- `schedule(dynamic,12)`
- `schedule(dynamic,100)`
- `schedule(guided,1)`
- `schedule(guided,4)`
- `schedule(guided,12)`
- `schedule(guided,100)`

La première analyse figure 1.4 est effectuée sans réduction pour observer l'impact des paramètres du scheduling.

On peut observer que l'ordonnancement dynamique est plus coûteux qu'il n'apporte d'avantage dans ce cas d'application car l'opération a un temps d'exécution constant entre "chunks". Les meilleures performances sont obtenues pour une allocation de plus gros paquets à chaque chunk statiquement. C'est dans cette configuration que l'on perd le moins de temps d'ordonnancement du travail entre les threads.

On peut alors appliquer la clause schedule à la réduction avec les chunks les plus gros (100 et 1000 pour l'ordonnancement statique). On notera les abréviations D, G et S pour Dynamic, Guided et Static respectivement et SEQ pour séquentiel. On montre à l'aide de la figure 1.5 que des chunks > 100 n'apportent pas plus de gain pour un système composé de 12 threads maximum.

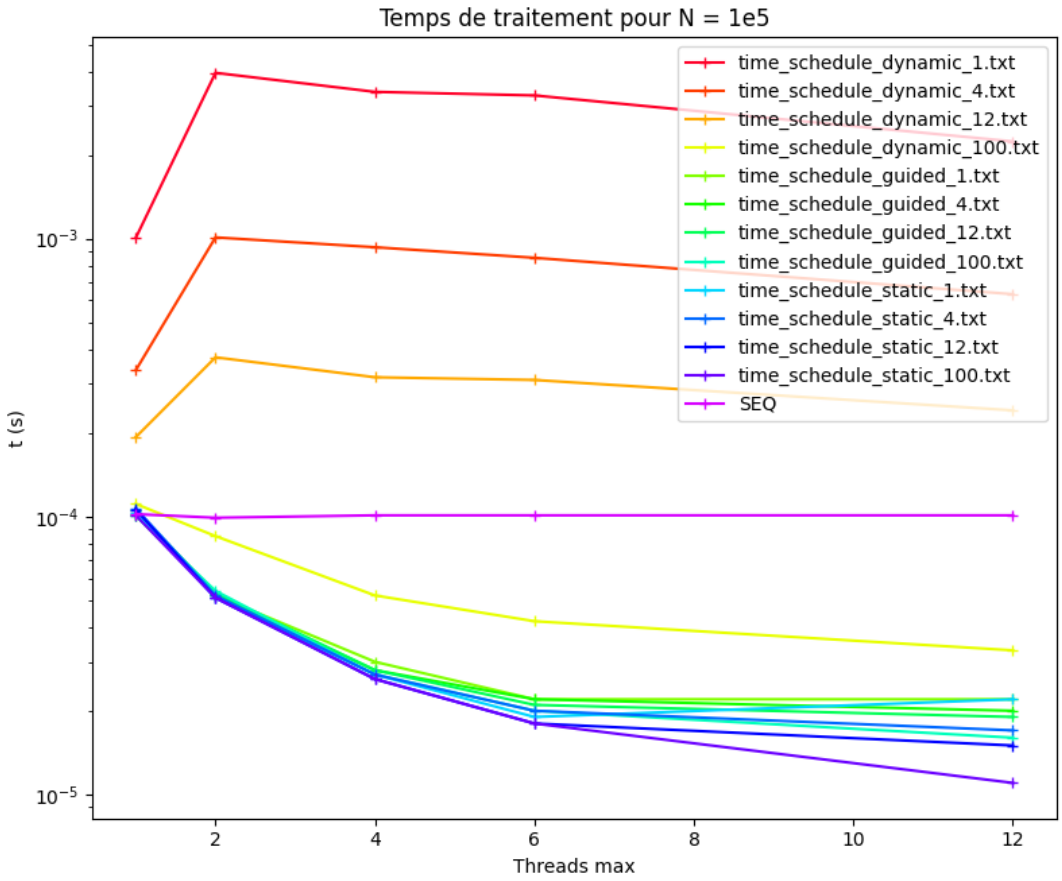


FIGURE 1.4 – Parallélisation par scheduling

Finalement, on peut analyser l'impact de l'ajout de zones critiques ou atomiques dans une partie de code afin de protéger l'accès aux données. On observe sur la figure 1.6, que la protection des zones entraîne une augmentation du temps de traitement. Ce résultat est attendu car l'atomicité empêche la parallélisation du code et consomme beaucoup de nombre de cycles (une centaine). Les abréviations A et C correspondent à Atomic et Critical respectivement.

Pour la meilleure clause schedule, on obtient une accélération de 9,18. En ajoutant la réduction, on obtient une accélération de 7,7. Dans ce cas d'utilisation, il vaut mieux ne pas mettre les deux optimisations en même temps.

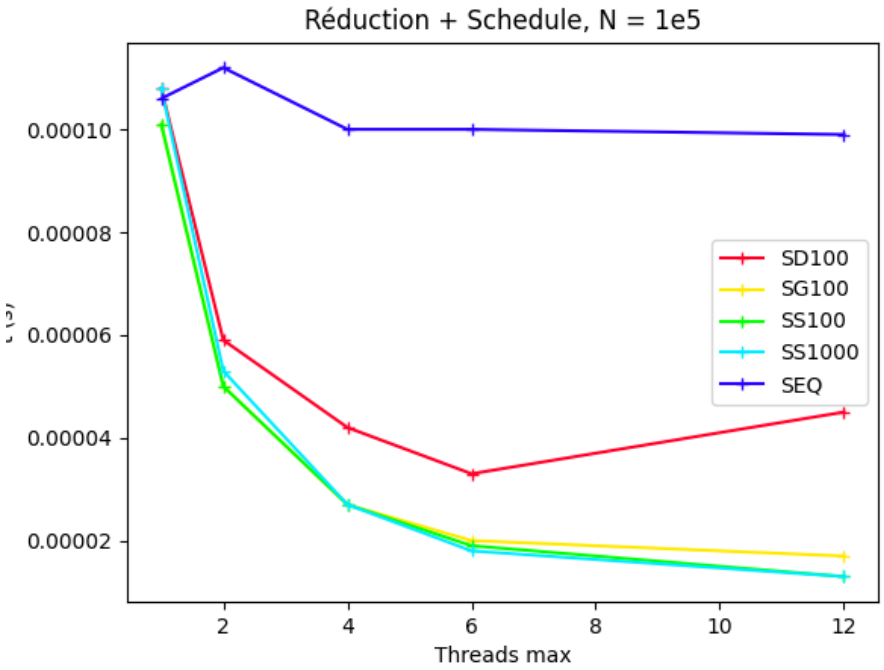


FIGURE 1.5 – Parallélisation par réduction et clause schedule

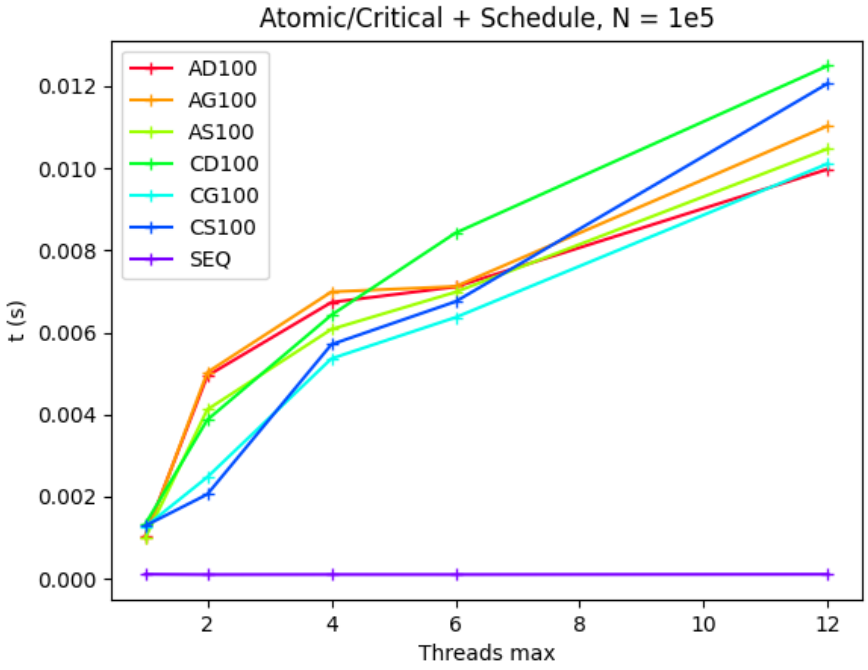


FIGURE 1.6 – Parallélisation par clause schedule avec instruction atomique et zone critique

1.4 - Détection des angles par la méthode de Shi-Tomasi

Le but de cette partie est d'évaluer les performances entre une première version fournie "non-optimisée" et une version que l'on va optimiser à l'aide d'OpenMP et des analyses réalisées précédemment.

1.4.1 - Analyse du code

Le fichier Makefile comporte 3 modes : Un mode debug sans optimisation affichant des éléments de debug via -g. Un mode release qui compile le projet avec une optimisation -O2. Un mode benchmark.

Les mesures de temps sont effectuées par les fonctions `omp_get_wtime()` et la fonction `timedifference_msec()` retournant le temps écoulé en milli-secondes.

Dans le programme `ShiTomasi.c`, les portions pouvant êtres optimisées sont les boucles for qui se trouvent dans les fonctions `compute_eigenvalues()`, `convolve()`.

Les 2 pragmas rajoutés pour optimiser les boucles sont :

convolve : #pragma omp parallel for schedule(static,100)
eigen : #pragma omp parallel for collapse(2) schedule(static,100)

1.4.2 - Analyse de l'influence des paramètres

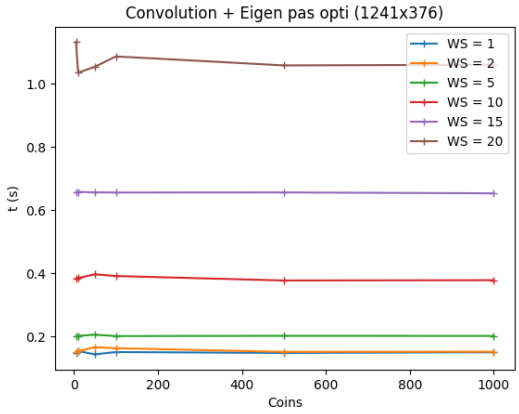
Les accélérations obtenues pour des valeurs faibles (fenêtre = 5 et coins = 100), on obtient un peu d'accélération. Pour ces paramètres, on obtient au mieux une accélération de 1,22 pour la convolution et 1,83 pour le eigen. Plus l'on augmente ces valeurs et plus l'accélération est importante.

Pour des valeurs plus élevées, on arrive à une accélération de 5 pour le calcul eigen, et de 2 pour la convolution figure 1.7. On peut observer que le nombre de coins à détecter n'impacte que très peu le temps de calcul. De plus, la parallélisation pose problème pour une fenêtre de 1 et le gain n'est visible qu'au dessus de 2.

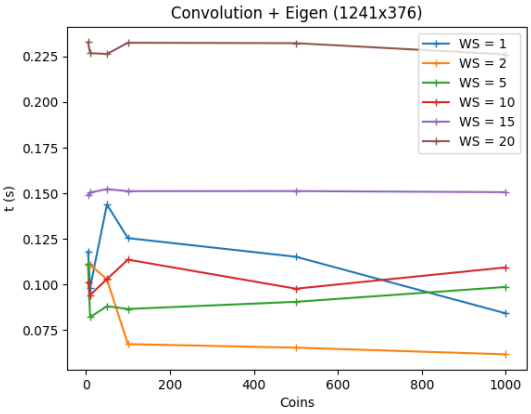
A l'aide de la figure 1.8, on peut constater que la forme de l'image n'impacte pas ou peu le temps de calcul, ce qui est important est le nombre de pixels total.

1.4.3 - CPP

Pour une fenêtre de 10 et de 1000 coins à détecter, on trace la courbe de CPP pour les versions non optimisée et optimisée figure 1.9. En absence d'un moyennage du temps d'exécution sur plusieurs itérations, la fréquence du CPU a été bloquée à 1,1 GHz.

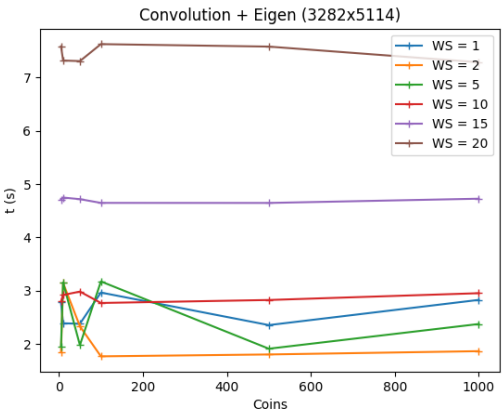


(a) ShiTomasi pas optimisé

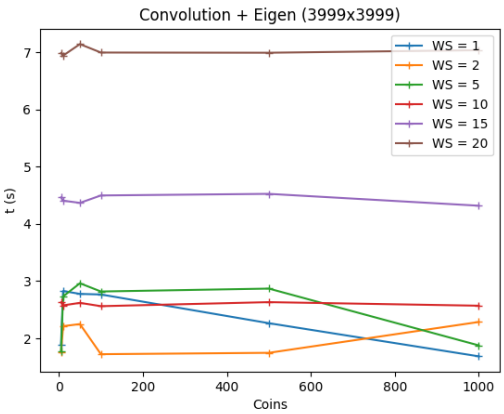


(b) ShiTomasi optimisé

FIGURE 1.7 – Image 0



(a) ShiTomasi optimisé image 1



(b) ShiTomasi optimisé image 4

FIGURE 1.8 – Différence entre les tailles des images

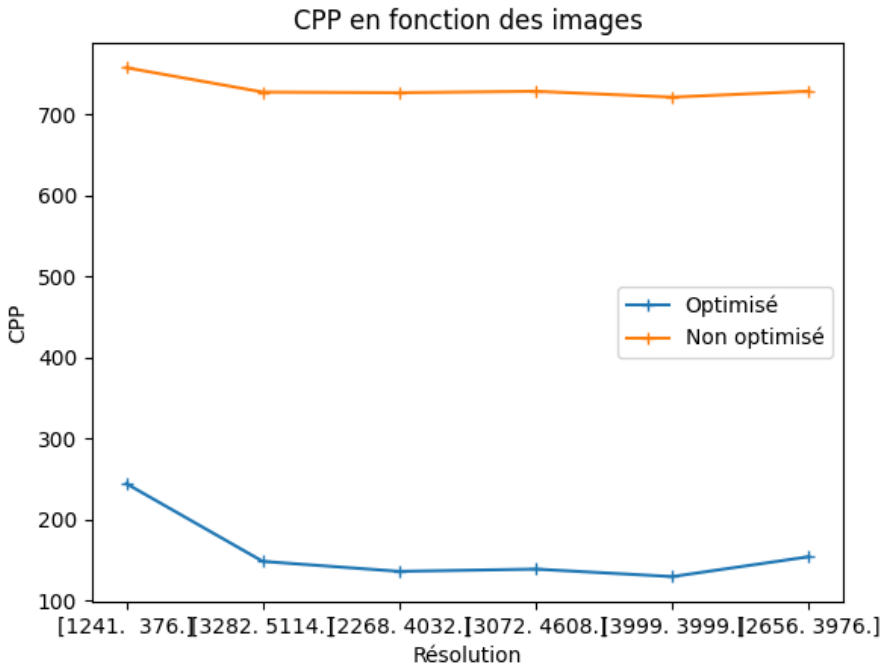


FIGURE 1.9 – CPP fonction de la résolution

1.5 - Conclusion

Au cours de ce TP, nous avons mis en oeuvre plusieurs directives d'OpenMP vues en cours. Nous avons pu, pour celles utilisées, comprendre l'impacte qu'elles ont sur la parallélisation d'un programme et notamment des gains que l'on peut obtenir. Ainsi, à l'aide des analyses dans les premières parties, nous avons pu ajouter des directives à un programme de détection de coins pour accélérer son exécution et ainsi obtenir une accélération de 5.