PdP : Master 1 Informatique

Parallélisation de boucles en OpenMP

Moussa YADE
Kenji FONTAINE

13 février 2018

Table des matières

1 Cas d'un cas régulier **3**
1.1 Adaptation du code 3
1.2 Expériences à mener 3

2 Cas d'un calcul déséquilibré **6**
2.1 Adaptation du code 6
2.2 Expériences à mener 7

3 Conclusion **10**

1 Cas d'un cas régulier

1.1 Adaptation du code

Version parallèle

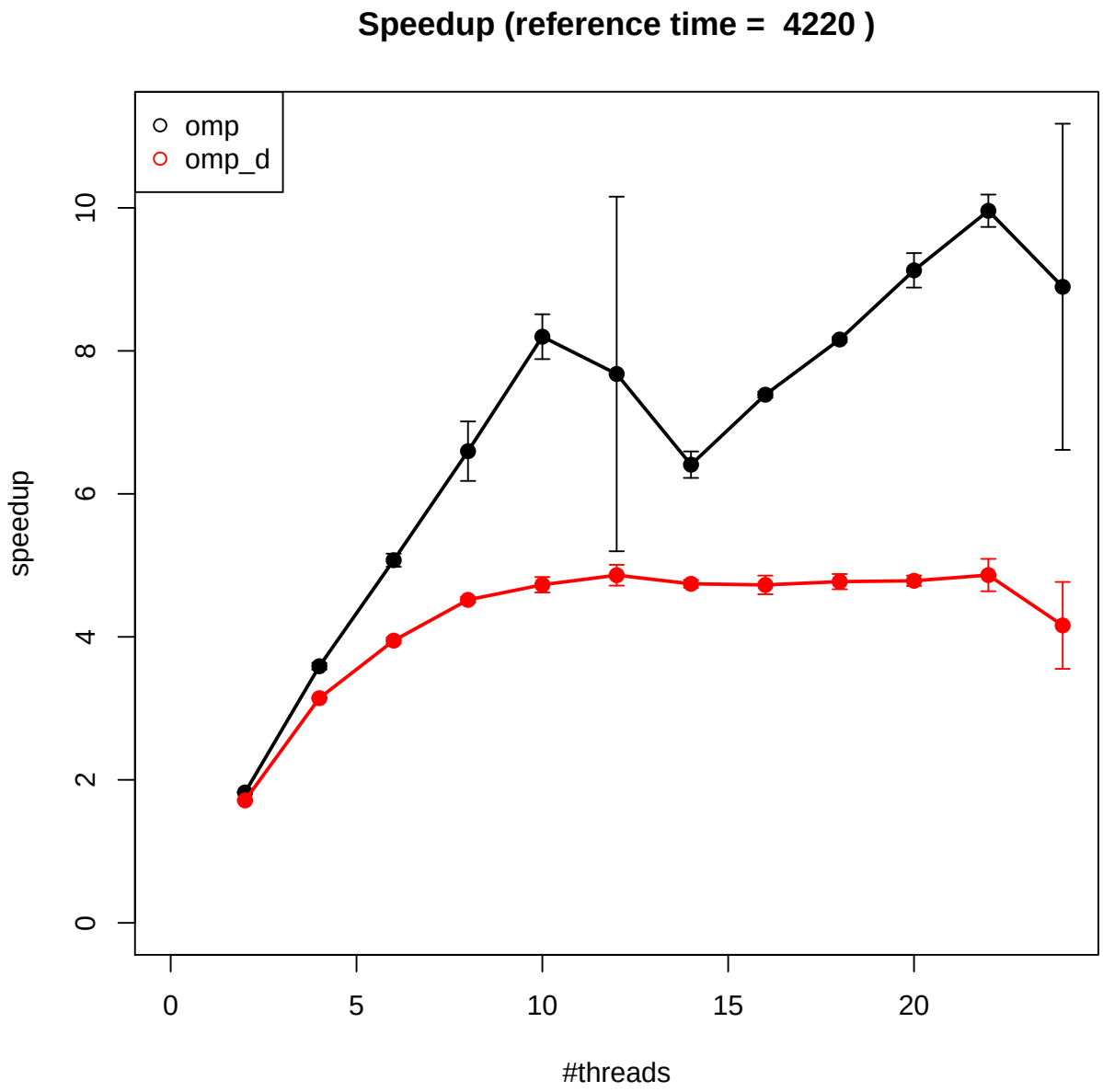
```
unsigned scrollup_compute_omp(unsigned nb_iter) {
    #pragma omp parallel // Création équipe threads
    for (unsigned it = 1; it < nb_iter; it++) {
        #pragma omp for schedule(static)
        for (int i = 0; i < DIM; i++)
            for (int j = 0; j < DIM; j++)
                // Calcule
        swap_images();
    }
    return 0;
}
```

Version dynamique

```
unsigned scrollup_compute_omp(unsigned nb_iter) {
    #pragma omp parallel
    for (unsigned it = 1; it < nb_iter; it++) {
        #pragma omp for schedule(dynamic)
        for (int i = 0; i < DIM; i++)
            for (int j = 0; j < DIM; j++)
                // Calcule
        swap_images();
    }
    return 0;
}
```

1.2 Expériences à mener

Voici les résultats obtenus en utilisant le script *lancer-expe.sh* fourni. Nous avons effectué les tests avec 500 itérations. Voici les résultats obtenus :


Speedup (reference time = 4220)
omp
omp_d
speedup
0
2
4
6
8
10
0
5
10
15
20
#threads

Nous pouvons remarquer que la version parallèle est bien plus effiace que la version dynamique. Dans cette situation, la charge de travail est la même pour chaque itération de la boucle. Il n'y a donc que peu d'intérêt à répartir le travail de façon dynamique. De plus la partie calculatoire étant rapide, la répartition dynamique des tâches peut représenter un temps non négligeable. Au-delà d'une dizaine de threads, nous pouvons remarquer qu'augmenter le nombre de threads ne permet pas un gain en performance. Nous pouvons penser que les threads perdent beaucoup de temps à attendre de recevoir un calcul à effectuer. De plus, nous remarquons une baisse de performance avec 24 threads. Ceci coincidant avec le nombre de threads de la machine, on pourrait penser que la répartition dynamique perd en efficacité lorsque l'on a plus de threads que de coeurs.

2 Cas d'un calcul déséquilibré

2.1 Adaptation du code

Distribution statique

```
unsigned mandel_compute_omps (unsigned nb_iter) {
    for (unsigned it = 1; it <= nb_iter; it ++) {
        #pragma omp parallel for schedule(static, 10)
        for (int i = 0; i < DIM; i++)
            for (int j = 0; j < DIM; j++)
                cur_img (i, j) = iteration_to_color (compute_one_pixel (i, j));
        zoom ();
    }
    return 0;
}
```

Distribution dynamique

```
unsigned mandel_compute_ompd (unsigned nb_iter) {
    #pragma omp parallel
    for (unsigned it = 1; it <= nb_iter; it ++) {
        #pragma omp for schedule(dynamic)
        for (int i = 0; i < DIM; i++)
            for (int j = 0; j < DIM; j++)
               cur_img (i, j) = iteration_to_color (compute_one_pixel (i, j));
        zoom ();
    }
    return 0;
}
```

Distribution dynamique collapse

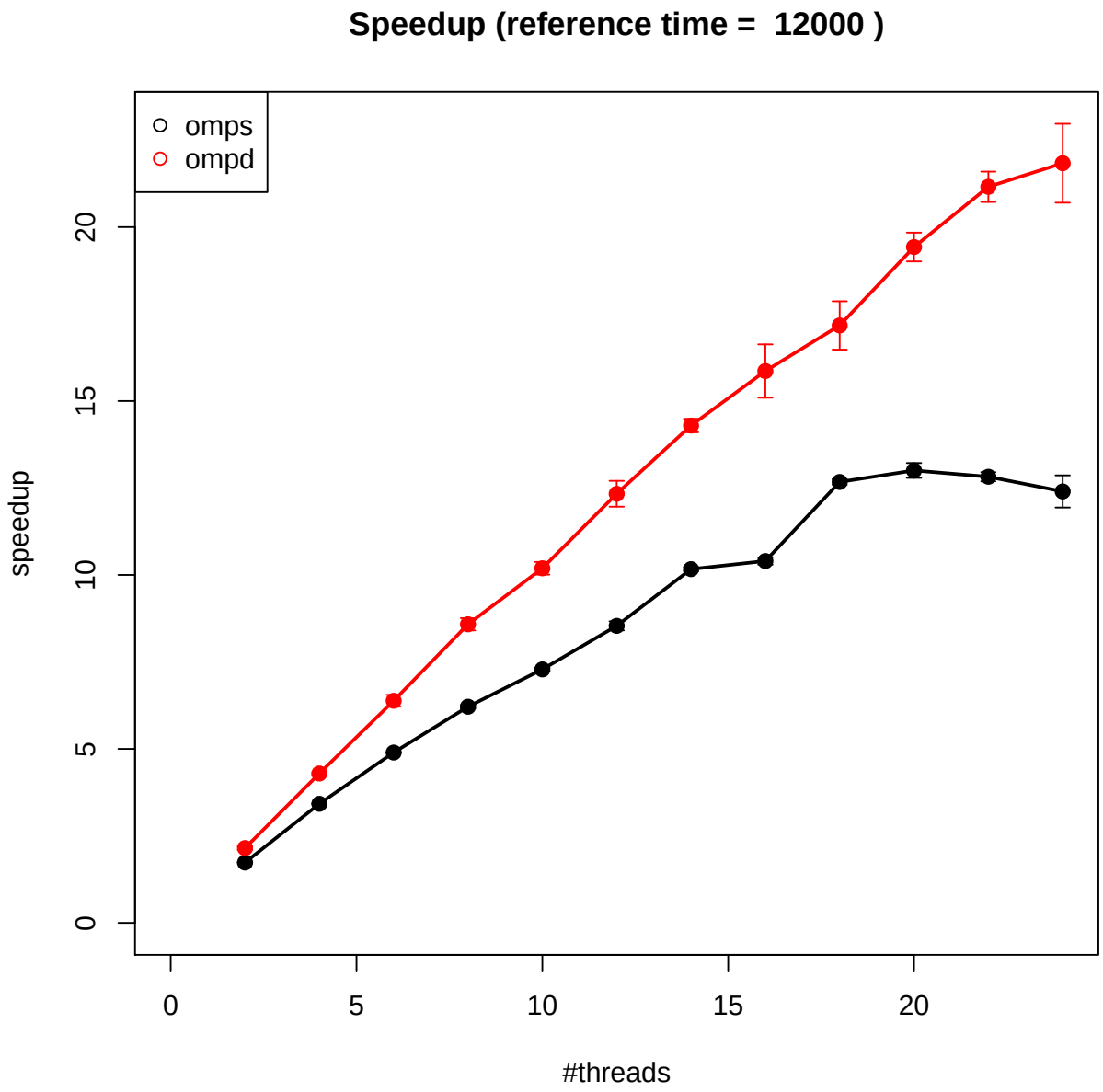
```
unsigned mandel_compute_omptiled (unsigned nb_iter) {
    tranche = DIM / GRAIN;
    #pragma omp parallel
    for (unsigned it = 1; it <= nb_iter; it ++) {
        #pragma omp for collapse(2) schedule(dynamic)
        for (int i=0; i < GRAIN; i++)
            for (int j=0; j < GRAIN; j++)
               // Calcule
    zoom ();
    }
    return 0;
}
```

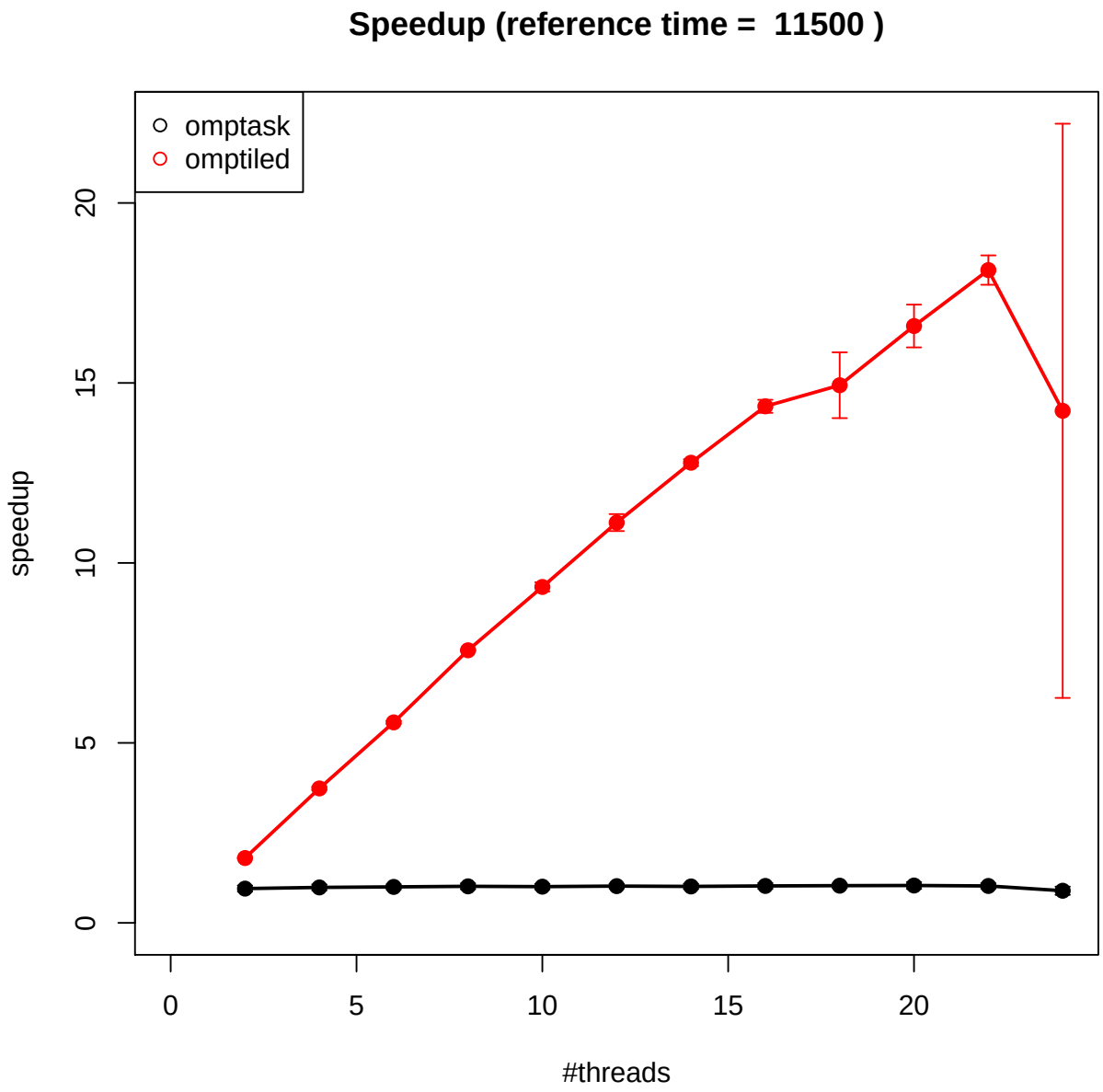
Distribution dynamique task

```
unsigned mandel_compute_omptask (unsigned nb_iter) {
    tranche = DIM / GRAIN;
    #pragma omp parallel
    for (unsigned it = 1; it <= nb_iter; it ++) {
        for (int i=0; i < GRAIN; i++)
            for (int j=0; j < GRAIN; j++)
                #pragma omp single
                #pragma omp task firstprivate(i,j)
                // Calcule
    zoom ();
    }
    return 0;
}
```

2.2 Expériences à mener

Voici les résultats obtenus en utilisant le script *lancer-expe.sh* fourni. Nous avons effectué les tests avec 50 itérations et pour une image de taille 512 par contrainte de temps. Voici les résultats obtenus :


Speedup (reference time = 12000)
omps
ompd
speedup
0
5
10
15
20
#threads


Speedup (reference time = 11500)
omptask
omptiled
speedup
0
5
10
15
20
#threads

Version non-tuilée

Nous pouvons remarquer que la version dynamique est plus performante que la version statique. Nous sommes cette fois dans une situation où chaque itération de boucle ne représente pas la même charge, il devient donc intéressant de répartir les calculs de façon dynamique afin d'équilibrer les charges. Ceci expliquant les meilleures performances de ompd face à omps. De plus, nous pouvons remarquer qu'au-delà de 18 threads, nous ne gagnons plus en performance pour la version statique. Nous pouvons penser qu'on doit attendre que les quelques threads ayant des tâches lourdes finissent et donc qu'ajouter des threads supplémentaires ne fait qu'accélérer la partie facile.

Version tuilée

Pour cette dernière partie, nous ne sommes pas parvenu à obtenir de meilleures performances avec la version omptask. Nous avons essayé de créer les tâches de façon différente et celle présentée dans le rapport est la meilleure que nous avons pu obtenir. De façon similaire à la versio non-tuilée, nous pouvons voir que la répartion dynamique permet d'obtenir de bonnes performances. Lorsque les calculs à effectuer sont inégaux, la répartition dynamique permet de lisser la charge de travail attribuée à chacun des threads.

3 Conclusion

Au travers de ce TP, nous avons pu voir qu'il n'y a pas de façon universelle de paralléliser un calcul. Certains méthodes sont plus efficaces que d'autres dans certaines situations. Nous avons également pu réviser l'utilisation de différentes directives OpenMP.