PdP : Master 1 Informatique

Tas de sable abéliens

Moussa YADE
Kenji FONTAINE

27 mars 2018

Table des matières

1 Optimisation de la version séquentielle

1.1 Déroulement de boucles

Déroulement en 2

```
/* Déroulement de boucles en 2 */
static void traiter_tuile2 (int i_d, int j_d, int i_f, int j_f) {

    PRINT_DEBUG ('c', "tuile [%d-%d][%d-%d] traitée\n", i_d, i_f, j_d, j_f);

    int modj = (j_f - (j_d + 1)) % 2;

    for (int i = i_d; i <= i_f; i++) {
        for (int j = j_d; j <= j_f - modj; j+=2) {
            compute_new_state (i, j);
            compute_new_state (i, j+1);
        }
        if (modj ==1 ) compute_new_state (i, j_f);
    }
}
```

Déroulement en 3

```
/* Déroulement de boucles en 3 */
static void traiter_tuile3 (int i_d, int j_d, int i_f, int j_f) {

    PRINT_DEBUG ('c', "tuile [%d-%d][%d-%d] traitée\n", i_d, i_f, j_d, j_f);

    int mod = (j_f - j_d + 1) % 3;
    for (int i = i_d; i <= i_f; i++) {
        for (int j = j_d; j <= j_f - mod; j+=3) {
            compute_new_state (i, j);
            compute_new_state (i, j+1);
            compute_new_state (i, j+2);
        }
        if (mod != 0) {
            compute_new_state (i, j_f);
            compute_new_state (i, j_f - 1);
        }
    }
}
```

Déroulement en 4

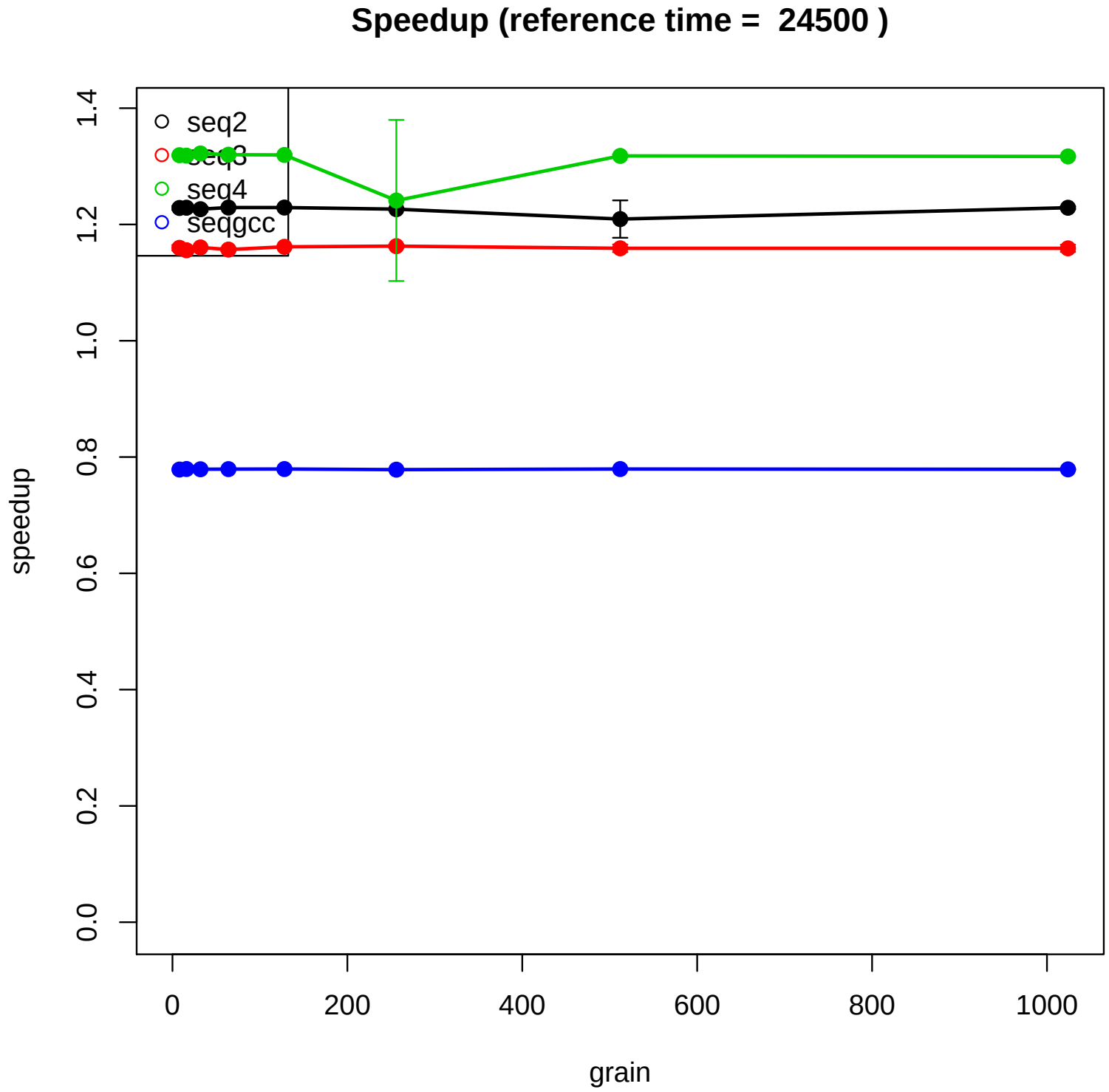
```
/* Déroulement de boucles en 4 */
static void traiter_tuile4 (int i_d, int j_d, int i_f, int j_f) {

    PRINT_DEBUG ('c', "tuile [%d-%d][%d-%d] traitée\n", i_d, i_f, j_d, j_f);

    int mod = (j_f - j_d + 1) % 4;
    for (int i = i_d; i <= i_f; i++) {
        for (int j = j_d; j <= j_f - mod; j+=4) {
            compute_new_state (i, j);
            compute_new_state (i, j+1);
            compute_new_state (i, j+2);
            compute_new_state (i, j+3);
        }
        if (mod != 0) {
            compute_new_state (i, j_f);
            compute_new_state (i, j_f - 1);
            compute_new_state (i, j_f - 2);
        }
    }
}
```

Déroulement via GCC

```
/* Déroulage de boucle via GCC*/

#pragma GCC push_options
#pragma GCC optimize("unroll-all-loops")
static void traiter_tuilegcc (int i_d, int j_d, int i_f, int j_f) {

    PRINT_DEBUG ('c', "tuile [%d-%d][%d-%d] traitée\n", i_d, i_f, j_d, j_f);

    for (int i = i_d; i <= i_f; i++) {
        for (int j = j_d; j <= j_f; j++)
            compute_new_state (i, j);
    }
}
#pragma GCC pop_options
```

1.1.1 Comparaison des vitesses d'exécution

Les expériences ont été menées en utilisant une table de taille 4096 avec une configuration aléatoire. Le temps utilisé comme référence est le temps d'exécution de la version séquentielle fournie.


Speedup (reference time = 24500)
seq2
seq3
seq4
seqgcc
speedup
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
200
400
600
800
1000
grain

Nous pouvons voir que le déroulement de boucles permet de gagner en performance. En séquentielle, nous obtenons les meilleures performances avec un déroulage de boucle en 4. Nous avons essayé d'utiliser les différentes versions de traitement de tuiles dans nos algorithmes en parallèle et il s'avère que la version déroulée en 2 (seq2) s'est révélée être la plus performante. Nous pouvons imaginer que le déroulement en 4 pourrait surcharger les ALU en parallèle et que le déroulement en 2 offre un bon compromis. Par la suite, nous avons opté pour la version seq2 dans nos algorithmes en parallèle.

1.2 Version récursive tuilée

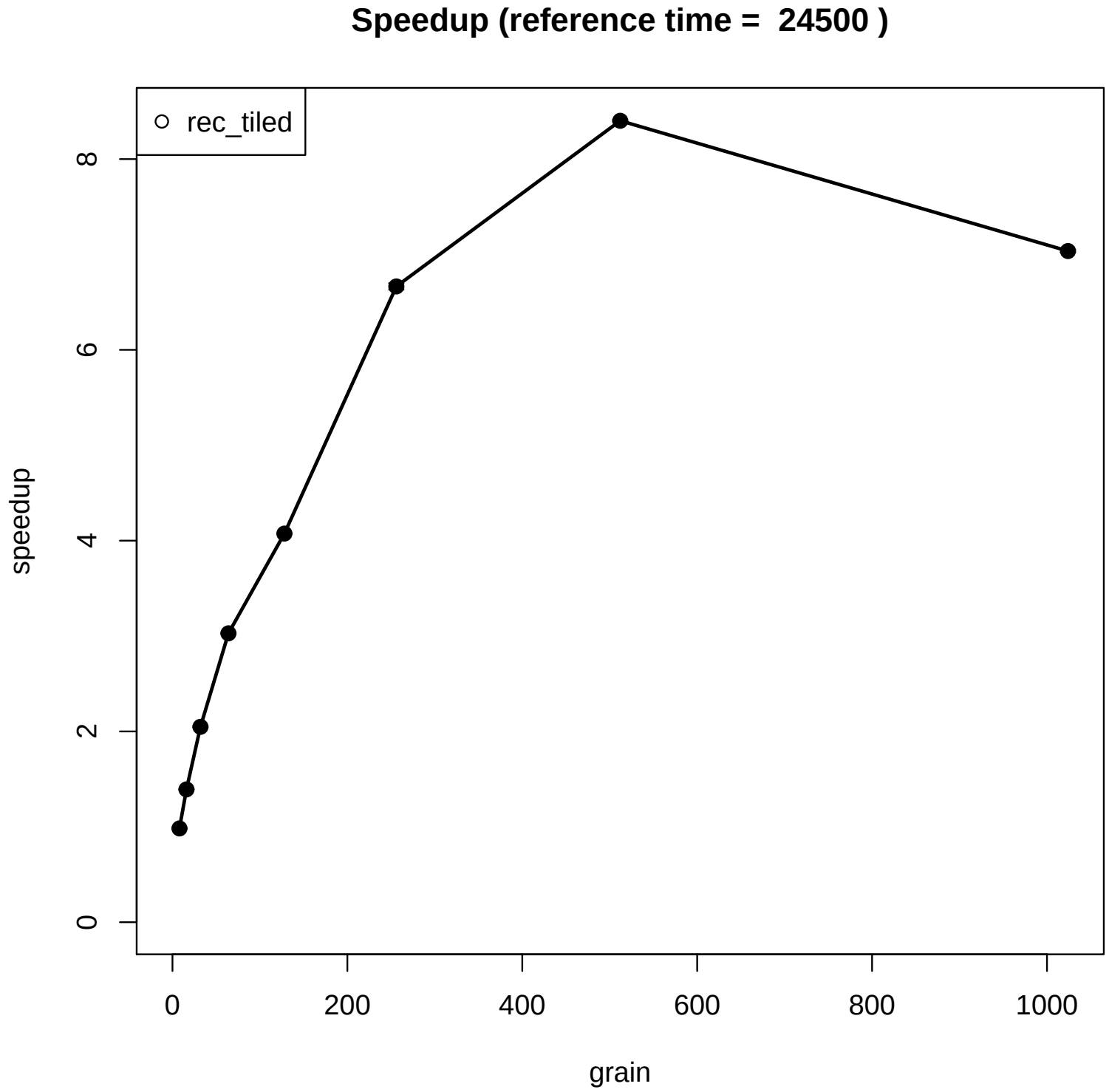
```
/* Profondeur maximale arbre d'appels récursifs */
unsigned int MAX_DEPTH = 32;

/* Traitement d'une tuile */
void traiter_tuile_rec(int i, int j, int tranche, int depth) {
    if (i<0 || i>=GRAIN || j<0 || j>=GRAIN) return;

    int tmp = 0;
    int i_d = i == 0 ? 1 : (i * tranche); /* i debut */
    int i_f = (i + 1) * tranche - 1 - (i == GRAIN-1); /* i fin */
    int j_d = j == 0 ? 1 : (j * tranche); /* j debut */
    int j_f = (j + 1) * tranche - 1 - (j == GRAIN-1);/* j fin */
    PRINT_DEBUG ('c', "tuile [%d-%d][%d-%d] traitée\n", i_d, i_f, j_d, j_f);

    do {
        changement = 0;
        traiter_tuile2(i_d, j_d, i_f, j_f);
        if (changement == 1) tmp = 1;
    } while (changement == 1);

    if(tmp == 1 && depth < MAX_DEPTH) {
        traiter_tuile_rec(i, j+1, tranche, ++depth);
        traiter_tuile_rec(i, j-1, tranche, ++depth);
        traiter_tuile_rec(i-1, j, tranche, ++depth);
        traiter_tuile_rec(i+1, j, tranche, ++depth);
    }
}

/* Version récursive tuilée */
unsigned sable_compute_rec_tiled(unsigned nb_iter) {
    tranche = DIM / GRAIN;
    int depth = 0;

    for (unsigned it = 1; it <= nb_iter; it ++) {
        changement_tile = 0; /* Modifiée par compute_new_state */
        for (int i=0; i < GRAIN; i++) {
            for (int j=0; j < GRAIN; j++) {
                traiter_tuile_rec(i, j, tranche, depth);
            }
        }
        if(changement_tile == 0) return it;
    }
    return 0;
}
```

Dans cette version, l'idée est de parcourir la table une première fois et de détecter une tuile instable puis de la traiter de façon séquentielle. Une fois cette tuile traitée, nous appelons de façon récursive cette même fonction sur les 4 tuiles voisines. De cette façon, nous pouvons éviter d'itérer sur des zones stables de la table et ainsi gagner en temps de calcul. Cette version est cependant très lente sur une configuration 4partout. Ne comportant aucune zone stable, le surcoût de la récursion pour éviter ces zones stables ne fait que nous faire perdre en performance.

Un des problèmes que nous avons rencontré avec cette version était une erreur de segmentation lorsque l'on utilisait cette version pour traiter une table assez grande (taille 512+) avec une configuration 4partout. L'erreur provenait d'un trop grand nombre d'appels récursifs. Pour remédier à ce problème nous avons ajouter la variable **depth** qui nous permet de limiter la profondeur maximale de l'arbre d'appels récursifs.

1.2.1 Vitesse d'exécution et meilleur grain

Les expériences ont été menées en utilisant une table de taille 4096 avec une configuration aléatoire. Le temps utilisé comme référence est le temps d'exécution de la version séquentielle fournie. Différents nombres de grain ont été testés : 8 16 32 64 128 256 512 1024.


Speedup (reference time = 24500)
rec_tiled
speedup
0
2
4
6
8
0
200
400
600
800
1000
grain

Nous obtenons les meilleurs performances avec un grain de 512, qui semblerait offrir le meilleur compromis entre nombre et taille des tuiles. La version tuilée récursive est notre algorithme séquentiel le plus performant, il servira donc de référence pour le speed-up de nos algorithmes en parallèle. Le temps moyen d'exécution pour un grain de 512 est de 2915 ms.

2 Versions parallèle

2.1 Parallélisation en ligne OpenMP for

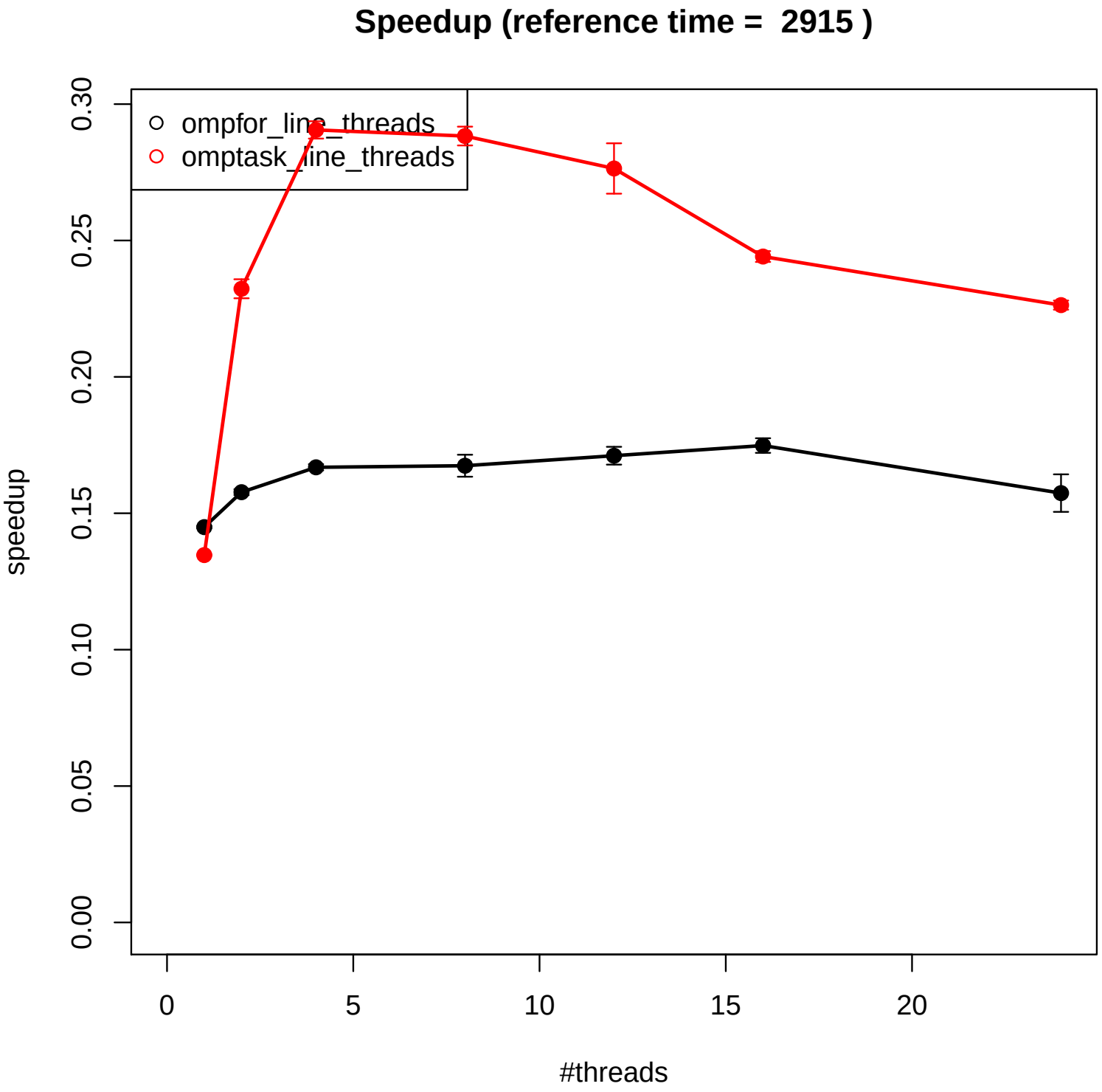
```
/* Version parallele en ligne omp for
 * Découpage en ligne :
 * 111111
 * 222222
 * 333333
 * 111111
 * 222222..
 * Chaque ligne est traitée par un seul thread.
 * Les lignes numérotées du même chiffre peuvent être traitées en parallèle.
 */
unsigned sable_compute_ompfor_line (unsigned nb_iter) {
    for (unsigned it = 1; it <= nb_iter; it++) {
        changement = 0;

        #pragma omp parallel
        #pragma omp for
            for (int j = 1; j <= DIM - 2; j+=3) {
                traiter_tuile2(j, 1, j, DIM - 2);
            }

        #pragma omp for
            for (int j = 2; j <= DIM - 2; j+=3) {
                traiter_tuile2(j, 1, j, DIM - 2);
            }

        #pragma omp for
            for (int j = 3; j <= DIM - 2; j+=3) {
                traiter_tuile2(j, 1, j, DIM - 2);
            }

        if (changement == 0) return it;
        }
    return 0;
}
```

2.2 Parallélisation en ligne OpenMP task

```
/* Version parallèle lignes omp task */
unsigned sable_compute_omptask_line(unsigned nb_iter) {
    unsigned tmp = 0;
    tranche = DIM / GRAIN;
    int i = 0;
    #pragma omp parallel
    #pragma omp single
    for (unsigned it = 1; it <= nb_iter && tmp == 0; it++) {
        changement = 0;
        for (i = 0; i < GRAIN; i += 3) {
            #pragma omp task firstprivate(i, changement) depend(out : table(i, 0))
            {
                for (int j = 0; j < GRAIN; j++) {
                    traiter_tuile2(i == 0 ? 1 : (i * tranche) /* i debut */,
                                   j == 0 ? 1 : (j * tranche) /* j debut */,
                                   (i + 1) * tranche - 1 - (i == GRAIN - 1) /* i fin */,
                                   (j + 1) * tranche - 1 - (j == GRAIN - 1) /* j fin */);
                }
            }
        }
        for (int k = 1; k < 3; k++) {
            for (i = k; i < GRAIN; i += 3) {
                #pragma omp task firstprivate(i, changement) depend(in : table(i - 1, 0), table(i
                {
                    for (int j = 0; j < GRAIN; j++) {
                        traiter_tuile2(i == 0 ? 1 : (i * tranche) /* i debut */,
                                       j == 0 ? 1 : (j * tranche) /* j debut */,
                                       (i + 1) * tranche - 1 - (i == GRAIN - 1) /* i fin */,
                                       (j + 1) * tranche - 1 - (j == GRAIN - 1) /* j fin */);
                    }
                }
            }
        }
        #pragma omp taskwait
        if (changement == 0) tmp = it;
    }
    return tmp;
}
```

2.3 Vitesse d'exécution

Les expériences ont été menées en utilisant une table de taille 4096 avec une configuration aléatoire. Le temps utilisé comme référence est le temps d'éxecution de la version récursive tuilée avec un grain de 512. Différents nombres de threads ont été testés : 1 2 4 8 12 16 24


Speedup (reference time = 2915)
ompfor_line_threads
omptask_line_threads
speedup
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0
5
10
15
20
#threads

Nous pouvons voir qu'augmenter le nombre de threads ne permet pas de gagner en performance de façon significative. Ceci pourrait s'expliquer par la faible parallélisation, nous ne pouvons traiter qu'un tier de la table à la fois. Nous pouvons également remarquer que la version task permet d'obtenir de meilleures performances. Ceci pourrait s'expliquer par le fait qu'avec une configuration aléatoire, chaque ligne ne représente par la même charge de calcul.

3 Version parallèle tuilée

3.1 Version parallèle 3 par 3

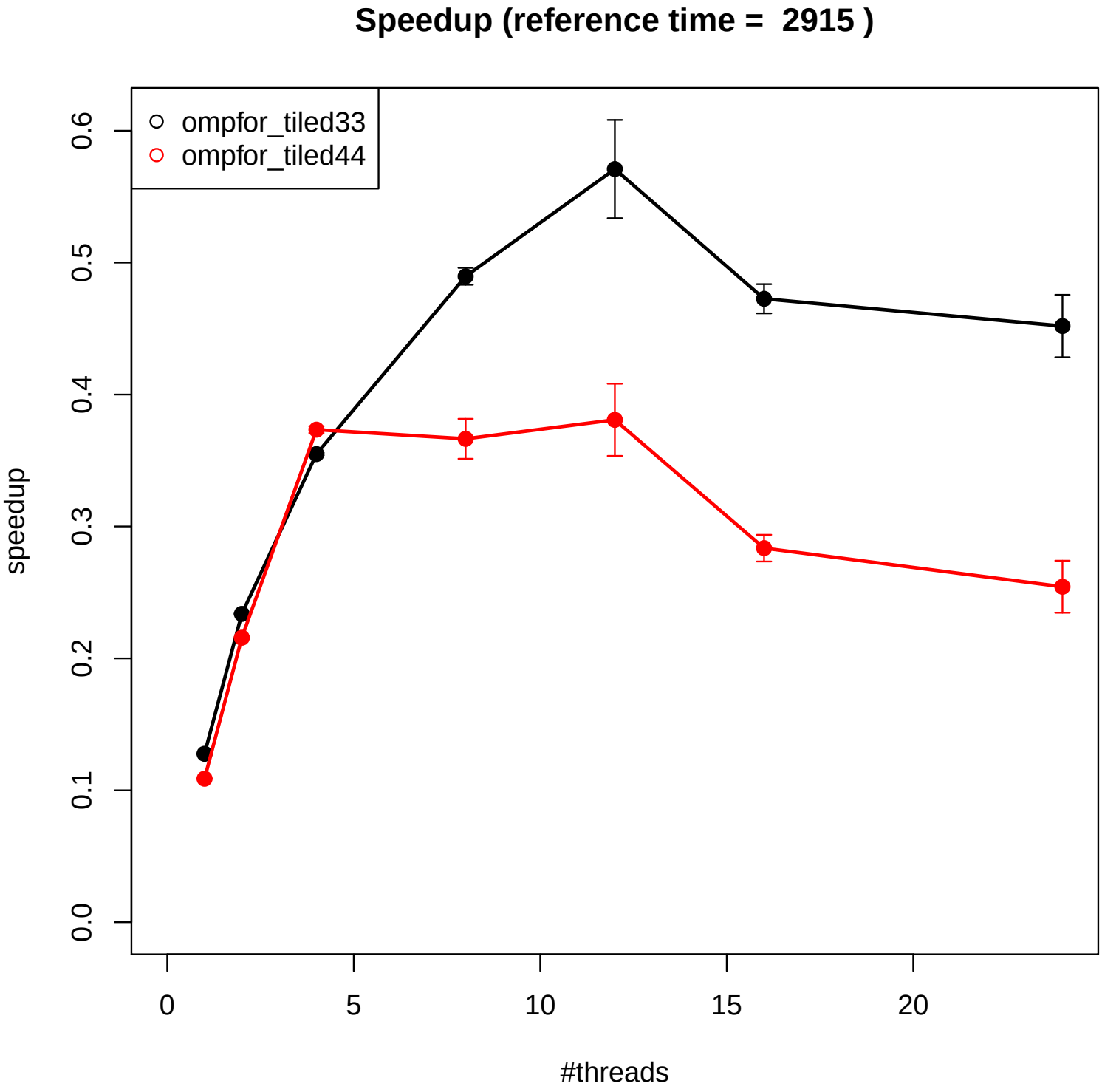
```
/* Version parallele tuilee omp for
 * Découpage en :
 * 123123
 * 456456
 * 789789...
 * 123
 * 456
 * 789...
 * Chaque tuile est traitée par un seul thread.
 * Les tuiles numérotées du même chiffre peuvent être traitées en parallèle.
 */
unsigned sable_compute_ompfor_tiled33 (unsigned nb_iter) {
    tranche = DIM / GRAIN;
    unsigned tmp = 0; //Contourne GCC
    #pragma omp parallel
    for (unsigned it = 1; it <= nb_iter && tmp == 0; it++) {
        changement = 0;
        for (int modi = 0; modi < 3; modi++) {
            for (int modj = 0; modj < 3; modj++) {
            #pragma omp for collapse(2)
                for (int i = modi; i < GRAIN; i+=3) {
                    for (int j = modj; j < GRAIN; j+=3) {
                        traiter_tuile2 (i == 0 ? 1 : (i * tranche) /* i debut */,
                                        j == 0 ? 1 : (j * tranche) /* j debut */,
                                        (i + 1) * tranche - 1 - (i == GRAIN-1)/* i fin */,
                                        (j + 1) * tranche - 1 - (j == GRAIN-1)/* j fin */);
                    }
                }
            }
        }
        if (changement == 0) tmp = it;
    }
    return tmp;
}
```

3.2 Version parallèle 4 par 4

```
/*
 * Version parallèle tuilée omp for
 * 1234
 * 5678
 * 3412
 * 7856
 */

unsigned sable_compute_ompfor_tiled44(unsigned nb_iter) {
    tranche = DIM / GRAIN;
    unsigned tmp = 0;

    for (unsigned it = 1; it <= nb_iter; it ++) {
        changement = 0;
        #pragma omp parallel
        for (int modi = 0; modi < 2; modi++) {
            for (int modj = 0; modj < 4; modj++) {
            #pragma omp for
                for (int i = modi; i < GRAIN; i+=2) {
                    for (int j = (modj + i%4)%4; j < GRAIN; j+=4) {
                        traiter_tuile (i == 0 ? 1 : (i * tranche) /* i debut */,
                                       j == 0 ? 1 : (j * tranche) /* j debut */,
                                      (i + 1) * tranche - 1 - (i == GRAIN-1)/* i fin */,
                                      (j + 1) * tranche - 1 - (j == GRAIN-1)/* j fin */);
                    }
                }
            }
        }
        if (changement == 0) return tmp = it;
    }
    return tmp;
}
```

3.3 Vitesses d'exécution

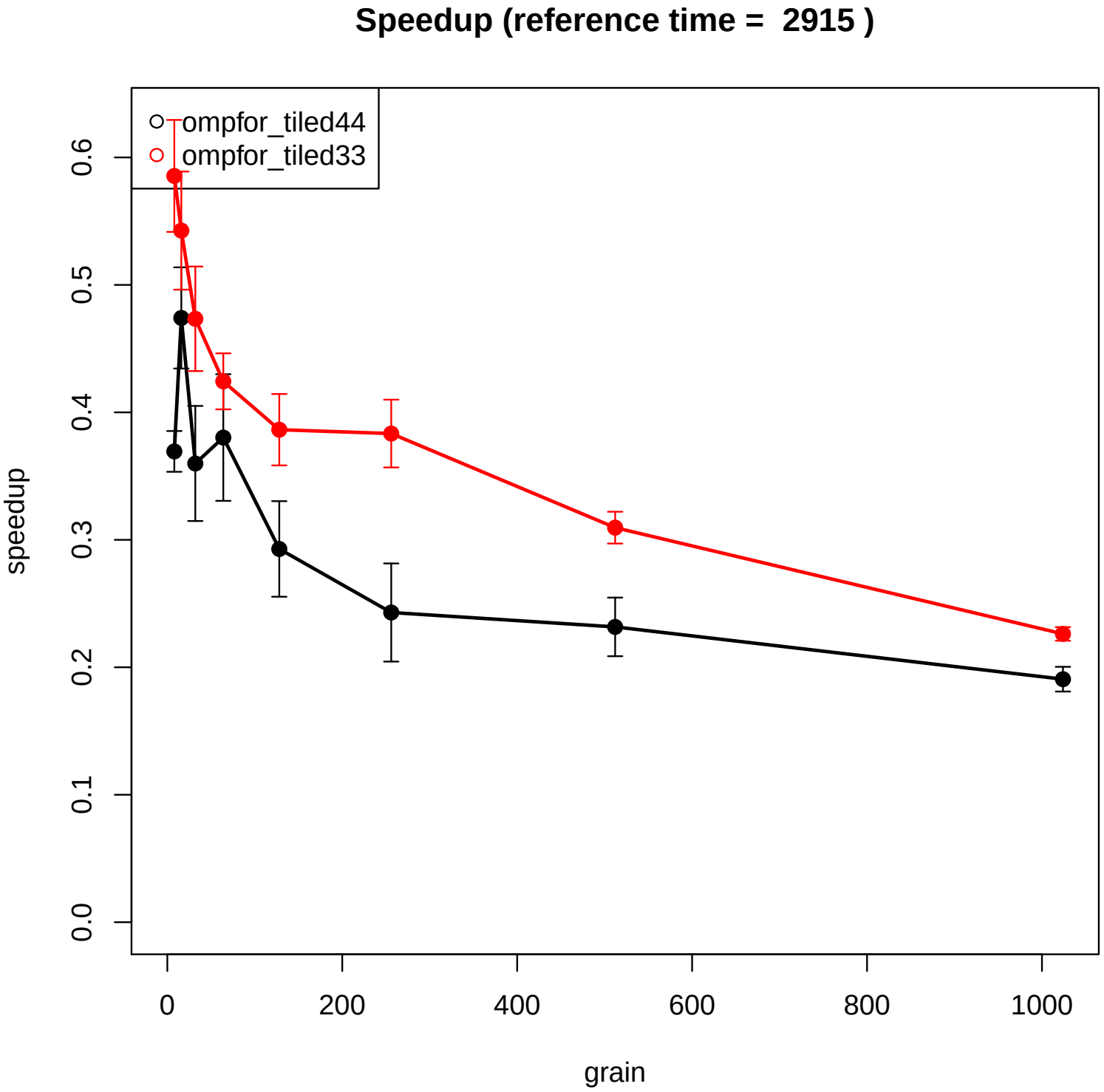
Les expériences ont été menées en utilisant une table de taille 4096 utilisant une configuration aléatoire. Le temps utilisé comme référence est le temps d'exécution de la version récursive tuilée avec un grain de 512. Différentes nombres de threads ont été testés : 1 2 4 8 12 16 24


Speedup (reference time = 2915)
ompfor_tiled33
ompfor_tiled44
speedup
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
5
10
15
20
#threads

Nous pouvons remarquer que nous perdons en performance au-delà de 12 threads. Il semblerait que l'hyperthreading ne fonctionne pas de façon efficace sur notre problème.

3.4 Optimisation du grain

Les expériences ont été menées en utilisant une table de taille 4096 utilisant une configuration aléatoire. Le temps utilisé comme référence est le temps d'exécution de la version récursive tuilée avec un grain de 512. Différents nombres de grains ont été testés : 8 12 42 64 128 256 512 1024


Speedup (reference time = 2915)
ompfor_tiled44
ompfor_tiled33
speedup
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
200
400
600
800
1000
grain

Nous pouvons remarquer que la performance décroit en fonction du nombre de grains. Autrement dit, plus les tuiles sont grandes, plus on a de performance. Dans une configuration aléatoire, la charge de travail n'est pas la même pour chaque tuile. Avoir de grandes tuiles permettrait de lisser la charge et ainsi obtenir de meilleures performances.

4 Conclusion et persepectives d'améliorations

4.1 Persepectives d'améliorations

Plusieurs améliorations été envisagées mais n'ont pas pu être implémentées.

— Parallélisation de la version récursive tuilée pour obtenir de meilleures performances. L'utilisation de task parait être le plus adapté, avec un tableau annexe pour gérer les dépendances.

— Changement du fonctionnement de l'algorithme de base. Passer à une version où chaque case lit la valeur de ses voisins et modifie sa propre valeur. Cela permet d'éviter les accès concurrents lecture/écriture ou écriture/écriture et d'obtenir une meilleure parallélisation.

4.2 Conclusion

Au travers de ce projet, nous avons pu voir qu'il pouvait être difficile de paralléliser un algorithme présentant beaucoup de dépendances entre ses données ainsi que de nombreux accès concurrents. Nous ne sommes pas parvenu à aller plus vite que la version séquentielle récursive tuilée.