Programmation Paralléle

Lucas Marques, Matis Duval

February 28, 2025

Abstract

Rapport Projet: Le jeu de la vie

1 Introduction

Ce projet s'inscrit dans le cadre de l'étude de la programmation parallèle appliquée à la simulation du Jeu de la Vie de Conway. L'objectif est d'optimiser l'exécution du programme en exploitant les capacités parallèles des architectures modernes.

2 Utilisation de EasyPaP

EasyPaP est un framework utilisé pour exécuter et analyser des simulations parallèles. Il nous permet d'observer l'impact de différentes stratégies de parallélisation sur la performance et d'obtenir des métriques précises sur le comportement du programme. Nous l'avons utilisé pour générer les traces et mesurer le temps d'exécution des différentes optimisations mises en place.

3 Premier Jalon

Pour ce premier jalon, nous travaillons sur les versions omp_taskfor ainsi que omp_taskloop s'appuyant respectivement sur la parallélisation des boucles for par openmp et l'utilisation des task pour permettre une meilleur distribution de la charge de travail entre les différents threads.
L'objectif de ces expérimentations est de trouver les paramètres optimaux permettant de tirer les meilleurs performances de notre code.

Parallélisation de base (salle 008) [Vendredi 28 février]

On ajoute les fonctions demandées:

```
unsigned life_compute_omp_tiled (unsigned nb_iter)
{
  unsigned res = 0;

  #pragma omp parallel
  {
      unsigned local_change = 0;

    for (unsigned it = 1; it <= nb_iter; it++) {
      local_change = do_tile(0, 0, DIM, DIM);

      #pragma omp for collapse(2) schedule(runtime) nowait
      for (int y = 0; y < DIM; y += TILE_H)
        for (int x = 0; x < DIM; x += TILE_W)
          local_change |= do_tile(x, y, TILE_W, TILE_H); // Combine changes from all tiles

      #pragma omp single
      {
        if (!local_change) { // If no changes, stop early
          res = it;
          it = nb_iter + 1; // Ensure all threads exit loop
        }
        swap_tables();
      }
    }
  }

```

```
  return res;
}
```

4 Modifications et Optimisations du Code

Nous avons apporté plusieurs modifications au code afin d'améliorer ses performances. Voici quelques versions mises à jour avec leurs justifications :

4.1 Optimisation avec OpenMP Task

```
unsigned  life_compute_omp_taskloop(unsigned nb_iter)
{
  int tuile[TILE_H][TILE_W + 1] __attribute__ ((unused));
  for (unsigned it = 1; it <= nb_iter; it++) {

#pragma omp master
    for (int y = 0; y < DIM; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W)
        #pragma omp task firstprivate(x,y) depend(out: tuile[x][y]) depend(in: tuile[x-1][y], tuile[x][y
        do_tile (x, y, TILE_W, TILE_H);

    swap_tables ();
  }
  return 0;
}
```

Cette optimisation remplace la parallélisation par boucle avec 'omp task', permettant une meilleure gestion des dépendances et une charge mieux équilibrée.

5 Expérimentations et Résultats

Nous allons tester différentes configurations de simulation pour analyser leurs impacts sur les performances.

5.1 Effet du Nombre de Threads

Nous alons dans un premier temps analyser l'impact du nombre de threads sur la performance, en faisant varier la variable d'environnement OMP_NUM_THREADS de 2 à 47 threads. Nous n'allons pas jusqu'au nombre maximal de threads de la machine afin d'en laisser au moins un au système d'exploitation.

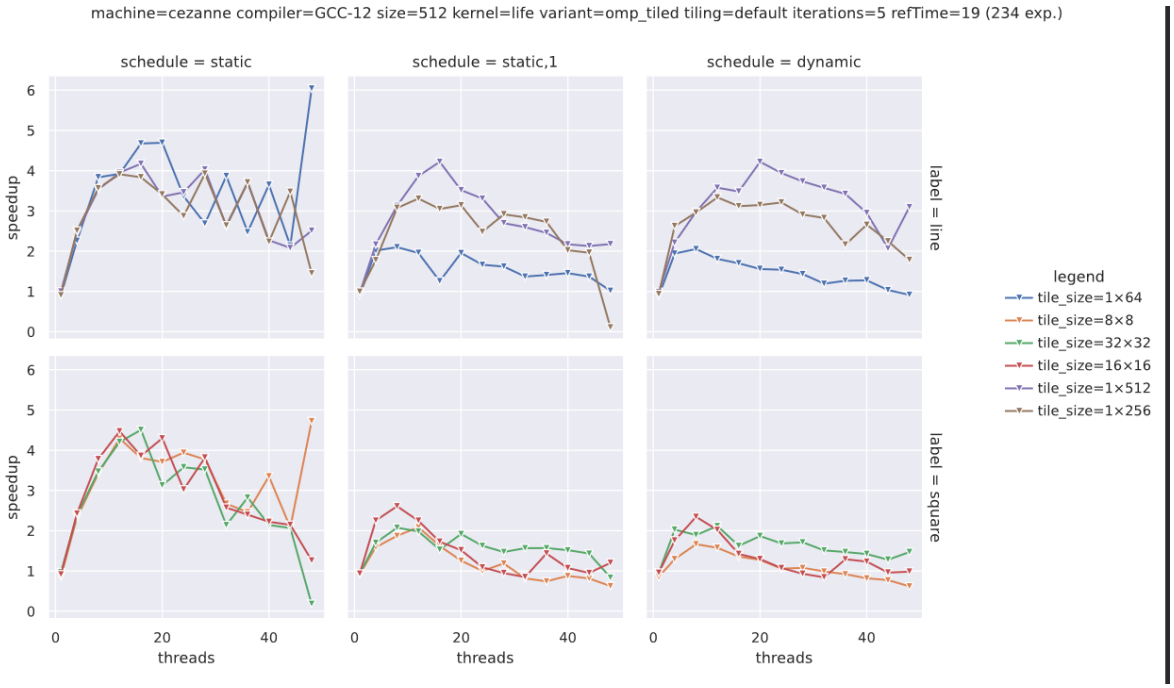

Figure 1: speedup en fonction du nombre de threads et de l'algorithme de scheduling

Nous observons des résultats très disparates, cependant le nombre de threads optimal semble se situer aux alentours de 20, avec un scheduling static. Pour la majorité des expériences, on constate que le paramètre size impact évidemment les performances mais n'impact pas les dynamiques des résultats.

5.2 Variation de la Taille de Simulation

Nous testons les tailles suivantes: 256, 512, 1024, 4096, 8192. L'objectif est d'observer comment la taille de la grille influence l'efficacité du parallélisme.

5.3 Effet de la Taille des Tuiles

Nous testons différentes tailles de tuiles et leur interaction avec le nombre de threads.

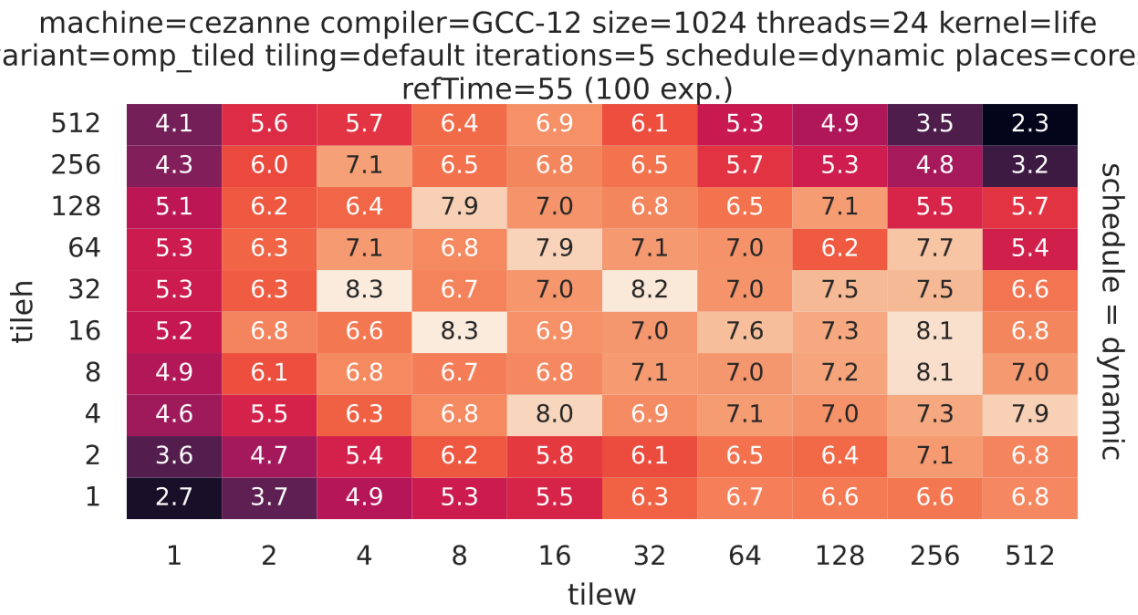

Figure 2: heatmap du speedup en fonction de la taille des tuiles

Nous constatons un speedup de 8.2 sur la taille 32*32, qui est ce que nous constatons expérimentalement également. Les 8.3 pour une taille de 32*4 et 8*16 ressemblent à du bruit que nous ne savons expliquer mais ne ressortent pas à l'expérimentation.

5.4 Comparaison GCC vs Clang

Maintenant que nous avons trouvés des paramètres semblant être optimaux, nous allons nous atteler à une dernière comparaison entre les compilateurs GCC et CLang. Nous observerons que la version GCC fournit de meilleurs performances que la version CLang.
Tout d'abord, il est important de préciser que l'option de compilation -O3 a été utilisée sur la version gcc.

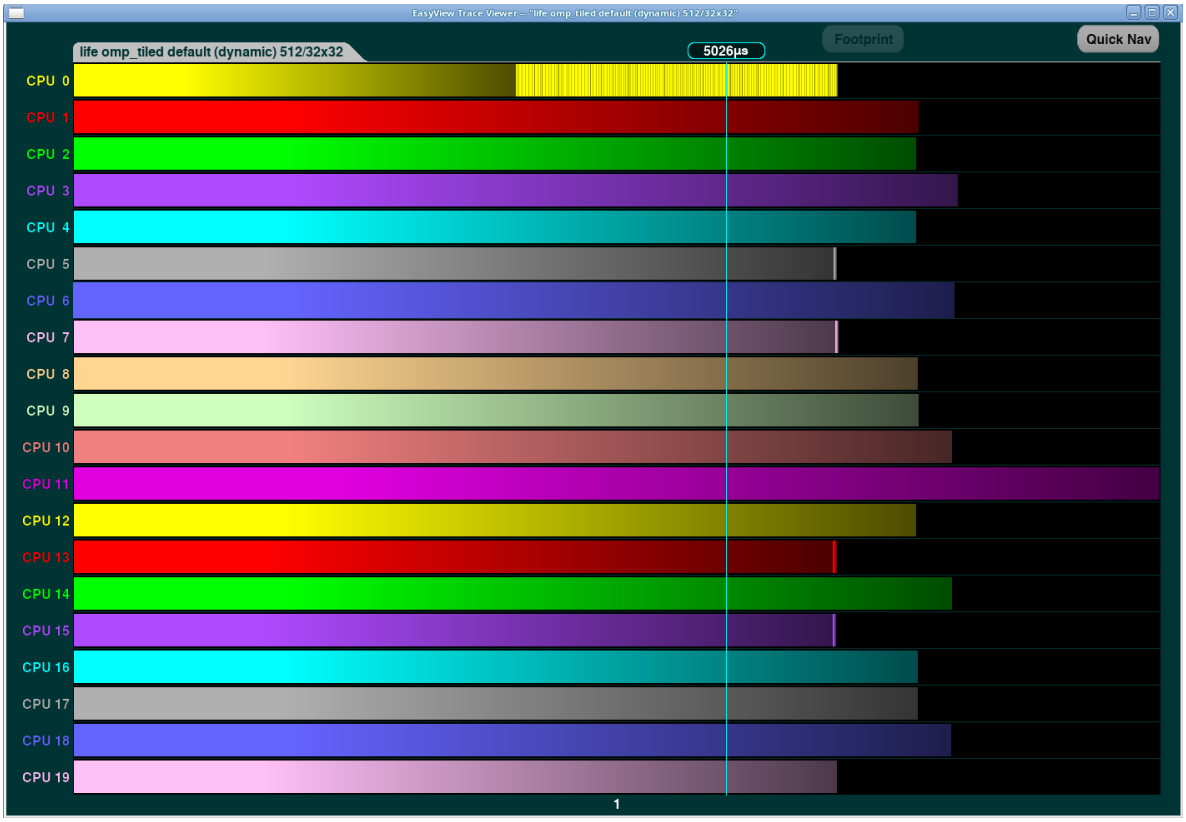

Figure 3: trace d'une run avec GCC

Avec -O3, nous observons une bonne répartition des tâches. La partie exécutée dans un thread seul (pragma omp single) est toujours exécutée sur le même thread. Cette répartition plus optimale que ce que nous avons pu observer sur la version compilée avec CLang nous aide à comprendre les différences dans les résultats des exécutions des deux versions.

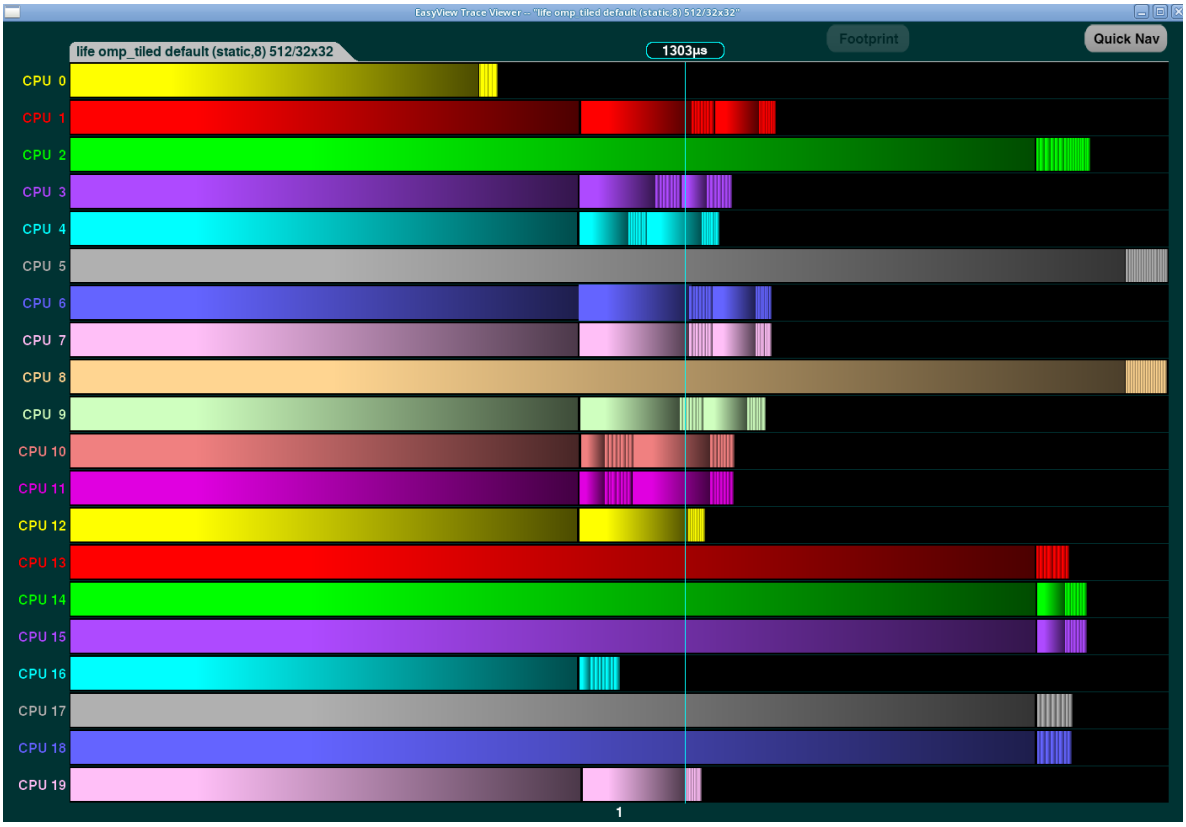

Figure 4: trace d'une run avec Clang

Nous observons sur cette trace une bien moins bonne répartition des charges. Tous les threads peuvent s'occuper de la partie exécutée sur un seul thread. Cela montre que nous n'avons pas su répartir correctement les charges parmis les différents threads dans notre code. Ce n'est donc pas forcément tant une "moins bonne" performance de CLang qu'un code sous optimal.