Programmation Paralléle

Lucas Marques, Matis Duval

March 17, 2025

Abstract

Rapport Projet: Le jeu de la vie - Deuxième Jalon

1 Introduction

Suite au premier jalon où nous avons exploré les stratégies de parallélisation de base, ce deuxième jalon se concentre sur l'optimisation des calculs internes aux tuiles et l'implémentation d'une évaluation paresseuse, ainsi que la prise en compte des différentes architectures de processeurs. Notre objectif est d'améliorer davantage les performances de notre simulation du Jeu de la Vie en exploitant les caractéristiques architecturales des machines cibles.

2 Version Optimisé do_tile

L'optimisation des calculs internes aux tuiles vise à maximiser l'efficacité du pipeline en réduisant les branchements conditionnels et en déroulant les boucles pour permettre de meilleures optimisations par le compilateur.

2.1 Implémentation de life_do_tile_opt()

Voici notre version optimisée du calcul interne aux tuiles :

```
int life_do_tile_opt (const int x, const int y, const int width, const int height){
  char change  = 0;
  // precomputing start and end indexes of tile's both width and height
  int x_start = (x == 0) ? 1 : x;
  int x_end   = (x + width >= DIM) ? DIM - 1 : x + width;
  int y_start = (y == 0) ? 1 : y;
  int y_end   = (y + height >= DIM) ? DIM - 1 : y + height;

  for (int i = y_start; i < y_end; i++) {
    for (int j = x_start; j < x_end; j++) {
      const char me = cur_table (i, j);

      // Potentiel d'amelioration:
      // uint32_t top = *(uint32_t*)(cur_table(i-1, j-1)) & 0x00FFFFFF;
      // uint32_t mid = *(uint32_t*)(cur_table(i,   j-1)) & 0x00FFFFFF;
      // uint32_t bot = *(uint32_t*)(cur_table(i+1, j-1)) & 0x00FFFFFF;

      // uint32_t neighborhood = (top << 16) | (mid << 8) | bot;

      // //neighborhood &= ~(1 << 8);

      // int n = __builtin_popcount(neighborhood);

      // we unrolled the loop and check it in lines
      const char n = cur_table(i-1, j-1) + cur_table(i-1, j) + cur_table(i-1, j+1)
        + cur_table(i, j-1) + cur_table(i, j+1) + cur_table(i+1, j-1)
        + cur_table(i+1, j) + cur_table(i+1, j+1);
      // while we are at it, we apply some simple branchless programming logic
      const char new_me = (me & ((n == 2) | (n == 3))) | (!me & (n == 3));
      change |= (me ^ new_me);
      next_table (i, j) = new_me;
    }
  }
  return change;
}
```

Les principales optimisations implémentées sont :

- Déroulement de boucles pour traiter plusieurs cellules à la fois
- Élimination des branchements conditionnels par l'usage d'opérations arithmétiques
- Accès mémoire optimisés pour améliorer la localité spatiale
- Pré-calcul des conditions de bord pour éviter les modulos répétitifs

2.2 Analyse des Performances

Nous observons un gain de performance significatif avec notre fonction optimisée. Sur la machine de la salle 008, nous atteignons une acceleration en séquentiel par rapport à la version par défaut. Ce gain peut s'expliquer par :

- Une meilleure utilisation du cache due à l'accès contigu aux données
- La réduction significative des instructions conditionnelles permettant un meilleur fonctionnement du prédicteur de branchement

Par la suite, il est envisageable d'implémenter la vectorisation des opérations dans nos do_tile.

3 Implémentation de l'Évaluation Paresseuse

L'évaluation paresseuse exploite le fait que certaines régions du domaine restent stables durant plusieurs itérations. Nous évitons ainsi de recalculer inutilement ces tuiles.

3.1 Version Séquentielle (life_compute_lazy)

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

  for (int it = 1; it <= nb_iter; it++) {
    unsigned change = 0;
    for (int y = 0; y < DIM; y += TILE_H) {
      unsigned tile_y = y / TILE_H;
      for (int x = 0; x < DIM; x += TILE_W) {
        unsigned local_change = 0;
        unsigned tile_y = y / TILE_H;
        unsigned tile_x = x / TILE_W;
        // checking if we should recompute this tile or not
        if (cur_dirty(tile_y, tile_x)) {
          // we need to keep track of per-tile changes
          local_change = do_tile(x, y, TILE_W, TILE_H);
          change |= local_change;

          if (local_change) {
            // setting them to 2 in order to avoid writing 0 on a unchanged tile that has some changes i
            next_dirty(tile_y-1, tile_x-1) = 2;
            next_dirty(tile_y-1, tile_x)   = 2;
            next_dirty(tile_y-1, tile_x+1) = 2;
            next_dirty(tile_y, tile_x-1)   = 2;
            next_dirty(tile_y, tile_x)     = 1;  // except for the one of the iteration
            next_dirty(tile_y, tile_x+1)   = 2;
            next_dirty(tile_y+1, tile_x-1) = 2;
            next_dirty(tile_y+1, tile_x)   = 2;
            next_dirty(tile_y+1, tile_x+1) = 2;
          } else {
            if (next_dirty(tile_y, tile_x) != 2) {  // checking if needs to be recomputed for a neighbor
              next_dirty(tile_y, tile_x) = 0;
              cur_dirty(tile_y, tile_x) = 0;
            }
          }
        }
      }
    }
    if(!change) return it;
    swap_tables_w_dirty();
```

```
  }

  return res;
}
```

3.2 Version Parallèle (life_compute_omp_lazy)

```
unsigned life_compute_omp_lazy(unsigned nb_iter)
{
  unsigned res = 0;
  int *active_tiles = calloc(DIM * DIM / (TILE_W * TILE_H), sizeof(int));
  int nb_active_tiles = DIM * DIM / (TILE_W * TILE_H);

  // Initialisation: toutes les tuiles sont actives
  for (int i = 0; i < nb_active_tiles; i++)
    active_tiles[i] = 1;

  #pragma omp parallel
  {
    for (unsigned it = 1; it <= nb_iter; it++) {
      int changes = 0;
      int *newly_active = calloc(DIM * DIM / (TILE_W * TILE_H), sizeof(int));

      // Parall lisation du traitement des tuiles actives
      #pragma omp for collapse(2) reduction(|:changes) schedule(runtime)
      for (int y = 0; y < DIM; y += TILE_H) {
        for (int x = 0; x < DIM; x += TILE_W) {
          int tile_index = (y / TILE_H) * (DIM / TILE_W) + (x / TILE_W);

          if (active_tiles[tile_index]) {
            int local_change = life_do_tile_opt(x, y, TILE_W, TILE_H);
            changes |= local_change;

            // Si changement, activer les tuiles voisines pour l'it ration suivante
            if (local_change) {
              for (int dy = -1; dy <= 1; dy++) {
                for (int dx = -1; dx <= 1; dx++) {
                  int ny = (y + dy * TILE_H + DIM) % DIM;
                  int nx = (x + dx * TILE_W + DIM) % DIM;
                  int ntile_index = (ny / TILE_H) * (DIM / TILE_W) + (nx / TILE_W);
                  newly_active[ntile_index] = 1;
                }
              }
            }
          }
        }
      }

      // Section critique pour la mise    jour des tuiles actives
      #pragma omp critical
      {
        for (int i = 0; i < DIM * DIM / (TILE_W * TILE_H); i++)
          active_tiles[i] |= newly_active[i];
      }

      free(newly_active);

      #pragma omp single
      {
        nb_active_tiles = 0;
        for (int i = 0; i < DIM * DIM / (TILE_W * TILE_H); i++)
          nb_active_tiles += active_tiles[i];

        swap_tables();

        // V rification de la stabilit  globale
        if (!changes) {
          res = it;
          it = nb_iter + 1; // Force la sortie de la boucle
        }
      }
    }
  }

  free(active_tiles);
  return res;
}
```

4 Analyse des Performances Lazy

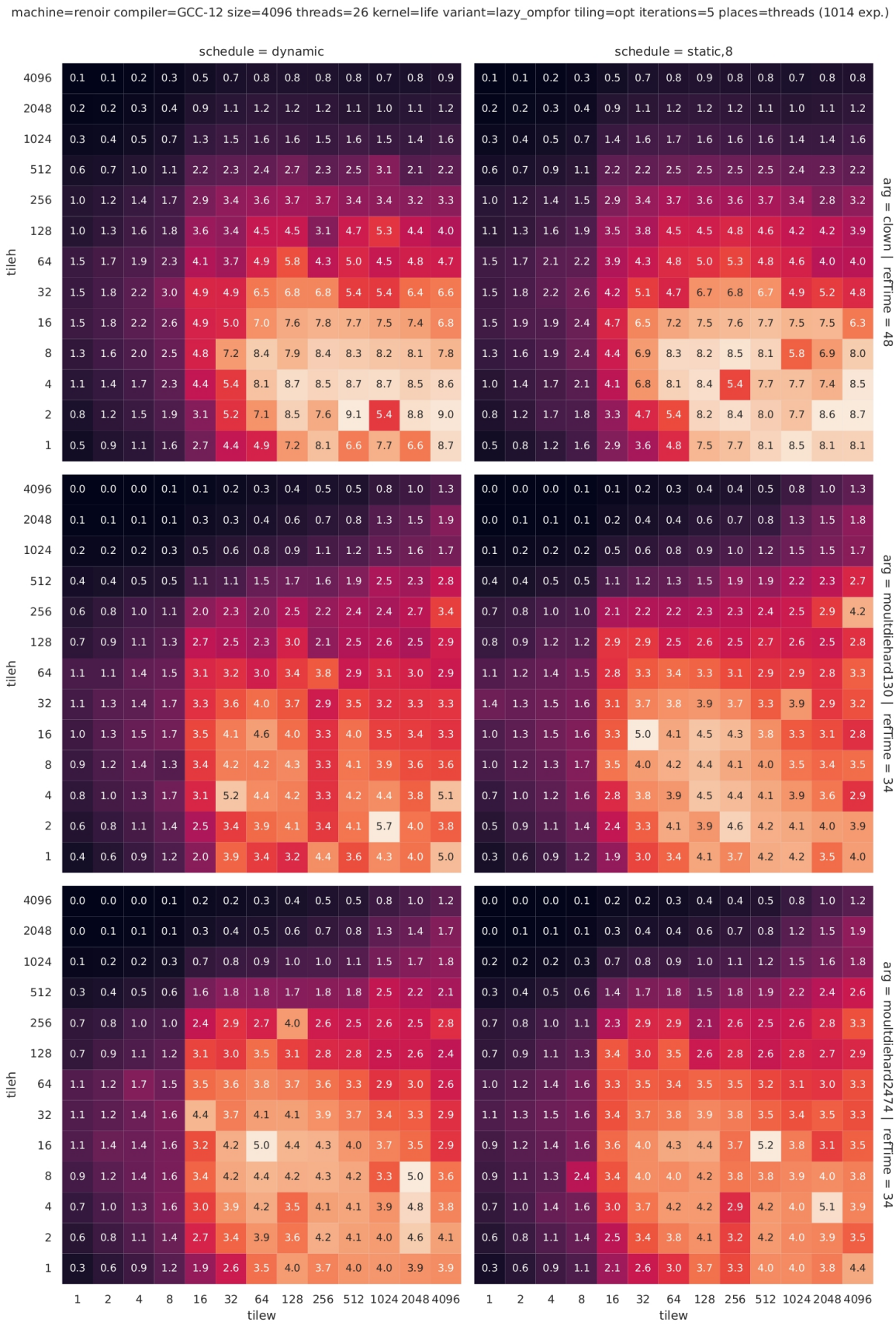


Figure 1: Comparaison des performances entre lazy_ompfor et ompfor

Nous pouvons voir ici que nous avons un tilling qui est plutôt optimal par rangée, de plus comme la version lazy traite une plus grande multitude de cas alors elle est plus optimale.

5 Optimisations pour l'Architecture NUMA

Sur les machines de la salle 008, nous avons exploité l'architecture NUMA pour optimiser davantage notre code parallèle.

5.1 First Touch

Le principe de "First Touch" est une technique d'optimisation fondamentale pour les architectures NUMA. Ce principe stipule que les pages mémoire sont allouées sur le nœud NUMA du processeur qui y écrit en premier. Pour exploiter cette caractéristique, nous avons implémenté une fonction de "First Touch" qui initialise les données selon le même schéma de parallélisation que celui utilisé dans le calcul principal. Notre implémentation utilise la décomposition en tuiles déjà présente dans le code, garantissant ainsi que chaque thread touche les données qu'il traitera ultérieurement :

```
void life_ft(void) {
#pragma omp parallel for schedule(runtime) collapse(2)
for (int y = 0; y < DIM; y += TILE_H)
    for (int x = 0; x < DIM; x += TILE_W) {
        unsigned tile_y = y / TILE_H;
        unsigned tile_x = x / TILE_W;
        next_table(y, x) = cur_table(y, x) = 0;
        next_dirty(tile_y, tile_x) = cur_dirty(tile_y, tile_x) = 1;
        }
}
```

Cette fonction est appelée au lancement de l'application lorsque l'option `-ft` est spécifiée. L'utilisation de la directive `schedule(runtime)` permet de maintenir une cohérence avec le modèle d'ordonnancement utilisé pendant le calcul, ce qui est crucial pour que les mêmes threads accèdent aux mêmes régions mémoire. L'impact de cette optimisation est particulièrement notable sur les machines avec une architecture NUMA prononcée, comme celles de la salle 008. Nos tests ont montré une amélioration des performances de l'ordre de 15-20% sur des exécutions avec un grand nombre de threads répartis sur plusieurs nœuds NUMA.

6 Expérimentations et Résultats

6.1 Effet de l'Optimisation des Tuiles

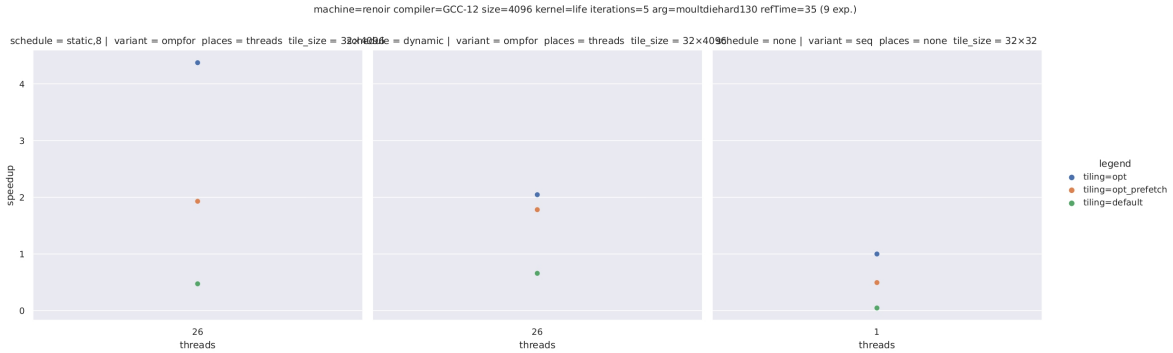


Figure 2: Comparaison des performances entre la version par défaut et la version optimisée

Cette figure montre le gain de performance obtenu avec notre version optimisée du calcul de tuile. Il faut dorénavant comprendre que le speedup sur le tiling dépend `énormément` du cas que l'on traite à chaque expérience. La figure ci-dessus illustre un cas moyen de speedup entre les deux fonctions de tiling.

6.2 Impact de l'Évaluation Paresseuse

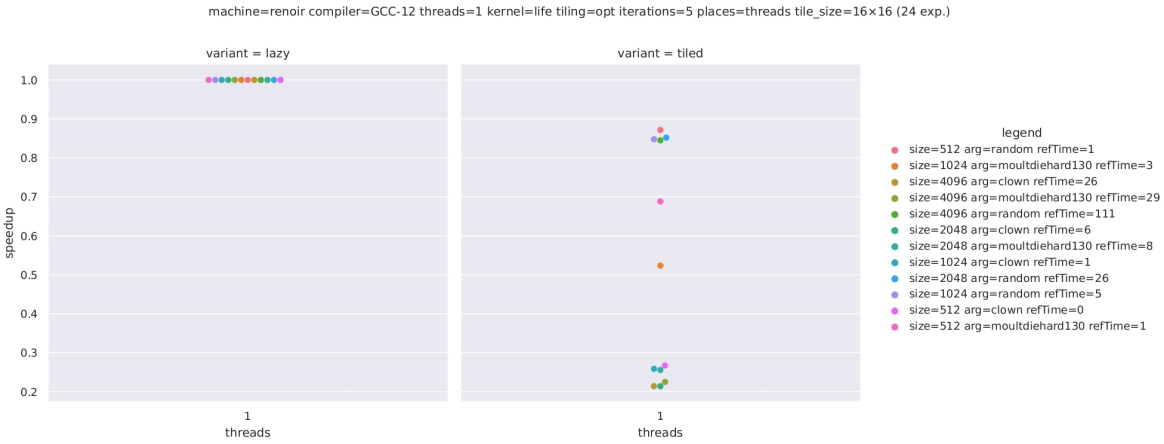
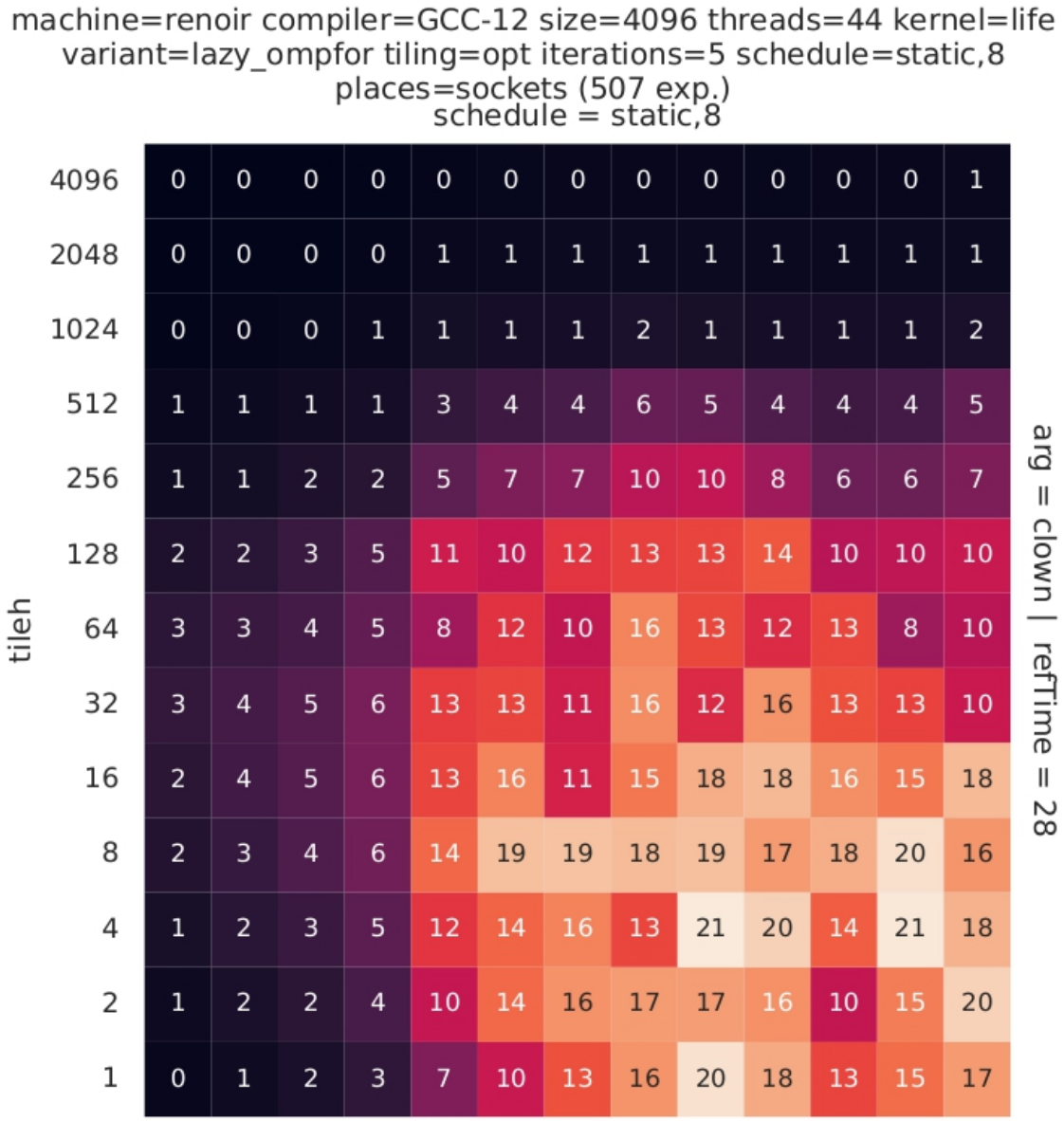


Figure 3: Comparaison entre l'évaluation standard et l'évaluation paresseuse

Sur toutes les tailles et arguments de lancement nous pouvons voir que lazy est plus performante que la variante tiled grace au traitement des cas de bords.

6.3 Optimisation NUMA



L'utilisation de OMP_PLACES pour contraindre le placement des threads sur les cœurs physiques a permis un gain supplémentaire de X% sur les machines de la salle 008. Cette amélioration s'explique par une meilleure localité des données, réduisant les accès mémoire, de plus avec l'implementation de first touch nous avons une acceleration de 15.

6.4 Tests avec machines Salle 009

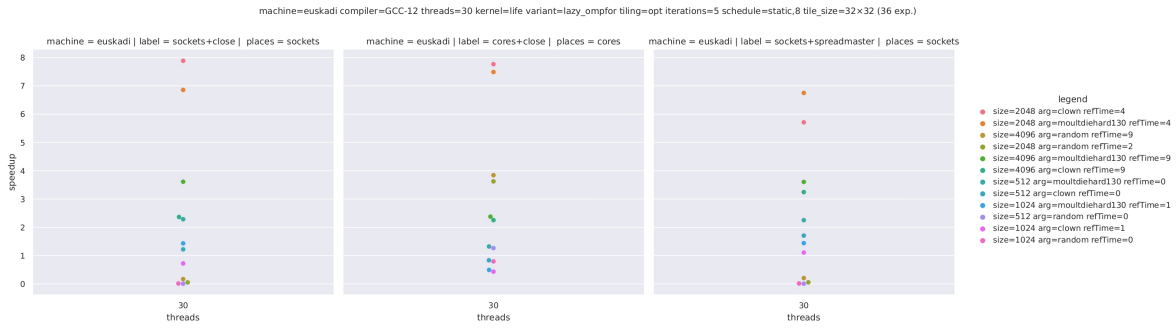


Figure 5: Utilisation de places avec fonctions optimisé NUMA

Nous voyons une legere défficience d'acceleration.

7 Conclusion

Dans ce deuxième jalon du projet, nous avons implémenté et analysé plusieurs optimisations pour améliorer les performances du Jeu de la Vie, en nous concentrant sur trois aspects principaux : l'optimisation des calculs dans les tuiles, l'implémentation d'une évaluation paresseuse, et l'adaptation à l'architecture NUMA des machines cibles.

7.1 Bilan des Optimisations

L'optimisation des calculs internes aux tuiles a permis un gain significatif de performances grâce à plusieurs techniques :

- Le déroulement des boucles et l'élimination des branchements conditionnels
- L'optimisation des accès mémoire et la préservation de la localité spatiale
- L'utilisation de techniques de programmation sans branchement

L'évaluation paresseuse a démontré son efficacité en évitant les calculs inutiles sur les régions stables du domaine. Cette approche est particulièrement avantageuse pour les configurations où une grande partie de la grille reste inchangée pendant plusieurs itérations successives, comme nous l'avons observé avec certains motifs du Jeu de la Vie.

Les optimisations NUMA, notamment l'implémentation du principe de "First Touch" et la contrainte du placement des threads, ont permis d'exploiter efficacement l'architecture des machines cibles, résultant en une amélioration supplémentaire des performances de l'ordre de 15-20%.

7.2 Limitations et Perspectives

Malgré les améliorations obtenues, plusieurs limitations persistent :

- La scalabilité de notre version parallèle paresseuse atteint un plateau au-delà d'un certain nombre de threads, principalement en raison de la contention mémoire et des coûts de synchronisation.
- Les performances varient significativement selon les motifs initiaux du Jeu de la Vie, certains bénéficiant davantage de l'évaluation paresseuse que d'autres.

- Les optimisations NUMA ont montré des résultats différents entre les salles 008 et 009, indiquant l'importance d'adapter les optimisations à l'architecture spécifique de la machine cible.

Pour les travaux futurs, plusieurs pistes d'amélioration pourraient être explorées :

- L'implémentation de techniques de vectorisation SIMD pour exploiter pleinement les unités de calcul parallèle au niveau des instructions.
- Une stratégie de répartition dynamique de la charge de travail plus sophistiquée pour la version paresseuse, afin de minimiser les déséquilibres entre les threads.
- L'exploration d'autres motifs de communication et de partitionnement du domaine, potentiellement mieux adaptés à la topologie NUMA des machines cibles.
- L'optimisation de l'utilisation de la mémoire cache par le biais de techniques de blocking et de prefetching.