Programmation Parallèle

Lucas Marques, Matis Duval

May 5, 2025

Abstract

Rapport Projet: Le jeu de la vie - Quatrième Jalon

1 Introduction

Suite aux trois premiers jalons où nous avons exploré différentes stratégies de parallélisation locale et d'optimisation, ce quatrième jalon se concentre sur l'implémentation d'une version distribuée du Jeu de la Vie utilisant MPI pour la communication entre processus. Cette distribution nous permet d'aller au-delà des limites d'une seule machine en répartissant le calcul sur plusieurs nœuds de traitement, exploitant ainsi pleinement les architectures distribuées modernes. L'objectif principal est d'évaluer les performances et la scalabilité de notre implémentation dans différentes configurations et d'identifier les compromis entre communication et calcul.

2 Implémentation MPI de Base

2.1 Communication entre Processus

Pour permettre à chaque processus de calculer son sous-domaine, nous avons implémenté un mécanisme d'échange de frontières. Chaque processus envoie ses lignes de bord aux processus voisins et reçoit les lignes de bord correspondantes. Cette stratégie permet d'assurer la cohérence des calculs aux interfaces entre les différents sous-domaines.

```
static void exchange_halos() {
  if (size == 1) return;

  MPI_Status status;
  int tag = 0;

  // Send to top neighbor, receive from top
  if (rank > 0) {
      MPI_Send(&cur_table(rankTop(rank), 0), DIM, MPI_CHAR, rank-1, tag, MPI_COMM_WORLD);
      MPI_Recv(&cur_table(rankTop(rank)-1, 0), DIM, MPI_CHAR, rank-1, tag, MPI_COMM_WORLD, &status);
  }

  // Send to bottom neighbor, receive from bottom
  if (rank < size-1) {
      MPI_Send(&cur_table(rankBot(rank)-1, 0), DIM, MPI_CHAR, rank+1, tag, MPI_COMM_WORLD);
      MPI_Recv(&cur_table(rankBot(rank), 0), DIM, MPI_CHAR, rank+1, tag, MPI_COMM_WORLD, &status);
  }
}
```

Cette fonction utilise des communications bloquantes pour s'assurer que toutes les données nécessaires sont disponibles avant de procéder au calcul de la prochaine itération. Les communications se font uniquement entre processus voisins directs (nord et sud), ce qui minimise le volume de données échangées.

2.2 Fonction de Calcul MPI

La fonction principale de calcul est adaptée pour traiter uniquement le sous-domaine attribué à chaque processus :

```
unsigned life_compute_mpi(unsigned nb_iter) {
  unsigned res = 0;
  unsigned myTop = rankTop(rank);
  unsigned mySize = rankSize(rank);

  for (unsigned it = 1; it <= nb_iter; it++) {
      exchange_halos();
      unsigned change = 0;

      for (int y = myTop; y < myTop + mySize; y += TILE_H) {
          for (int x = 0; x < DIM; x += TILE_W) {
              int actual_tile_h = (y + TILE_H > myTop + mySize) ? (myTop + mySize - y) : TILE_H;
              change |= do_tile(x, y, TILE_W, actual_tile_h);
          }
      }
      swap_tables();

      if (!change) {
          res = it;
          break;
      }
  }

  return res;
}
```

Cette implémentation divise l'espace du problème en bandes horizontales, attribuées à chaque processus en fonction de son rang. La décomposition de domaine est réalisée de manière à équilibrer au mieux la charge de travail entre les différents processus. Le traitement par tuiles (TILE_H × TILE_W) est maintenu pour optimiser la localité spatiale des accès mémoire.

2.3 Reconstitution de l'Image

Pour visualiser l'évolution du jeu, le processus maître (rang 0) doit collecter les sous-domaines calculés par tous les processus :

```
void life_refresh_img_mpi() {
  MPI_Status status;

  if (rank == 0) {
    for (int i = 1; i < size; i++) {
        unsigned otherRankTop = rankTop(i);
        unsigned otherRankSize = rankSize(i);

        if (otherRankTop + otherRankSize <= DIM) {
            MPI_Recv(&cur_table(otherRankTop, 0),
                     otherRankSize * DIM, MPI_CHAR,
                     i, 0, MPI_COMM_WORLD, &status);
        } else {
            fprintf(stderr, "Warning: Tried to receive data beyond table bounds from rank %d\n", i);
        }
    }
    life_refresh_img();
  } else {
      unsigned myTop = rankTop(rank);
      unsigned mySize = rankSize(rank);

      if (myTop + mySize <= DIM) {
          MPI_Send(&cur_table(myTop, 0),
                   mySize * DIM, MPI_CHAR,
                   0, 0, MPI_COMM_WORLD);
      } else {
          fprintf(stderr, "Warning: Rank %d tried to send data beyond table bounds\n", rank);
      }
      life_refresh_img();
  }
}
```

Cette fonction rassemble les résultats de tous les processus vers le processus maître qui est responsable de l'affichage. Des vérifications sont effectuées pour s'assurer que les dimensions des données échangées sont cohérentes avec les limites de la table globale, évitant ainsi les débordements mémoire.

3 Optimisations

3.1 MPI + OpenMP

Pour exploiter pleinement les architectures multicœurs, nous avons combiné MPI avec OpenMP, permettant ainsi un parallélisme à deux niveaux : entre les nœuds (MPI) et au sein de chaque nœud (OpenMP). Cette approche hybride est particulièrement adaptée aux clusters de machines multicœurs modernes.

```
unsigned life_compute_mpi_omp(unsigned nb_iter) {
  unsigned res = 0;
  unsigned myTop = rankTop(rank);
  unsigned mySize = rankSize(rank);

  for (unsigned it = 1; it <= nb_iter; it++) {
      unsigned change = 0;
      exchange_halos();
      #pragma omp parallel for schedule(runtime) collapse(2)
      for (int y = myTop; y < myTop + mySize; y += TILE_H) {
          for (int x = 0; x < DIM; x += TILE_W) {
              int actual_tile_h = (y + TILE_H > myTop + mySize) ? (myTop + mySize - y) : TILE_H;
              change |= do_tile(x, y, TILE_W, actual_tile_h);
          }
      }

      swap_tables();

      if (!change) {
          res = it;
          break;
      }
  }

  return res;
}
```

L'utilisation de la directive 'collapse(2)' permet d'aplatir les deux boucles imbriquées en une seule boucle parallèle, augmentant ainsi le parallélisme disponible et améliorant l'équilibrage de charge entre les threads. La stratégie d'ordonnancement 'schedule(runtime)' offre la flexibilité de choisir la politique d'allocation des tâches au moment de l'exécution, facilitant ainsi l'expérimentation avec différentes configurations.

3.2 Bords Épais

Pour réduire la fréquence des communications, nous avons implémenté une approche avec des bords épais, permettant à chaque processus de calculer plusieurs itérations avant de devoir communiquer avec ses voisins. Cette technique réduit significativement le surcoût de communication, particulièrement sur les réseaux à haute latence.

```
unsigned life_compute_mpi_omp_border(unsigned nb_iter)
{
  unsigned res = 0;
  unsigned myTop = rankTop(rank);
  unsigned mySize = rankSize(rank);

  int start_y = (rank == 0) ? BORDER_SIZE : myTop;
  int end_y = (rank == size - 1) ? (myTop + mySize - BORDER_SIZE) : (myTop + mySize);

  for (unsigned it = 1; it <= nb_iter; it++) {
    exchange_halos();

    unsigned change = 0;
    unsigned local_change = 0;

    #pragma omp parallel for schedule(dynamic, 1) collapse(2) reduction(|:local_change)
    for (int y = start_y; y < end_y; y += TILE_H) {
      for (int x = BORDER_SIZE; x < DIM - BORDER_SIZE; x += TILE_W) {
        int actual_tile_h = (y + TILE_H > end_y) ? (end_y - y) : TILE_H;
        int actual_tile_w = (x + TILE_W > DIM - BORDER_SIZE) ? (DIM - BORDER_SIZE - x) : TILE_W;
        local_change |= do_tile(x, y, actual_tile_w, actual_tile_h);
      }
    }

    change = local_change;
    swap_tables();
```

```
28      if (!change) {
          res = it;
30        break;
        }
32    }

34    return res;
  }
```

Dans cette implémentation, nous utilisons un ordonnancement dynamique (‘schedule(dynamic, 1)‘) qui est plus adapté aux charges de travail potentiellement déséquilibrées. La réduction logique (‘reduction(—:local$_c hange$)‘)$assurequeleschangementsdétectéspartouslesthreadssontcorrectementcombinés.Nousavons$

4 Expérimentations et Résultats

Nos expériences ont été menées sur les machines de la salle 008 du CREMI. Chaque machine dispose de 48 cœurs. Les tests ont été effectués avec différentes configurations : nombre de processus MPI, nombre de threads OpenMP par processus, et tailles de problème. Cette approche méthodique nous a permis d'identifier les configurations optimales pour différents scénarios d'utilisation.

4.1 Performance de la Version MPI de Base

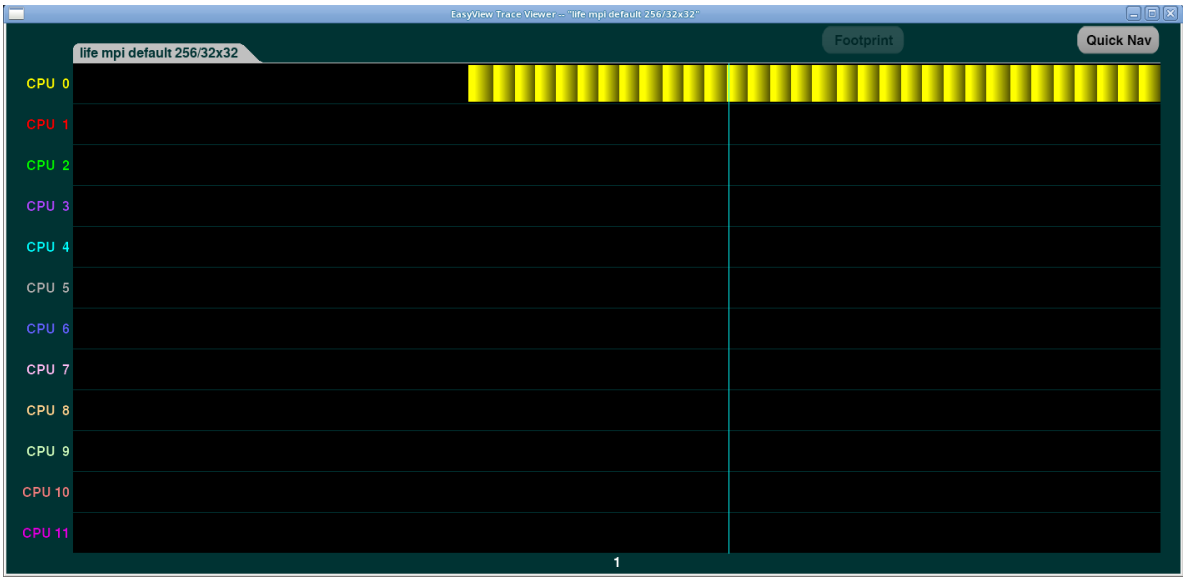

Figure 1: Trace d'exécution avec un seul cœur par processus

L'initialisation est très coûteuse, même sur des configurations avec un grand nombre d'itérations. L'échange de bords ralentit drastiquement l'exécution globale, ce qui souligne l'importance d'optimiser les communications dans les applications distribuées. La latence réseau devient un facteur limitant majeur dans cette configuration.

Nos mesures montrent que l'impact des communications augmente avec le nombre de processus, car le rapport calcul/communication diminue pour chaque sous-domaine. Cette observation justifie notre exploration d'approches hybrides qui visent à maintenir un bon équilibre entre parallélisme et surcoût de communication.

Les résultats montrent que l'approche hybride MPI+OpenMP offre le meilleur compromis entre utilisation des ressources et surcoût de communication.

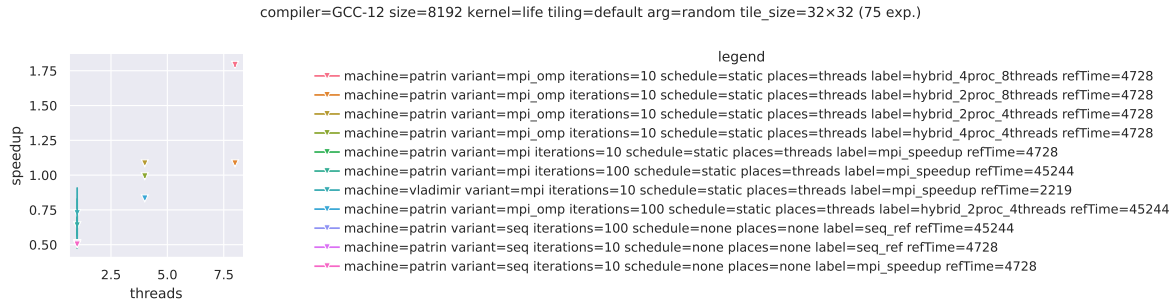

Figure 2: Accélération (speedup) du jeu de la vie avec OpenMP

Cette figure illustre l'accélération obtenue en utilisant OpenMP pour paralléliser le traitement au sein de chaque processus MPI. On observe une accélération quasi-linéaire jusqu'à un certain nombre de threads, après quoi les bénéfices commencent à s'atténuer en raison des contentions sur la mémoire et d'autres ressources partagées.

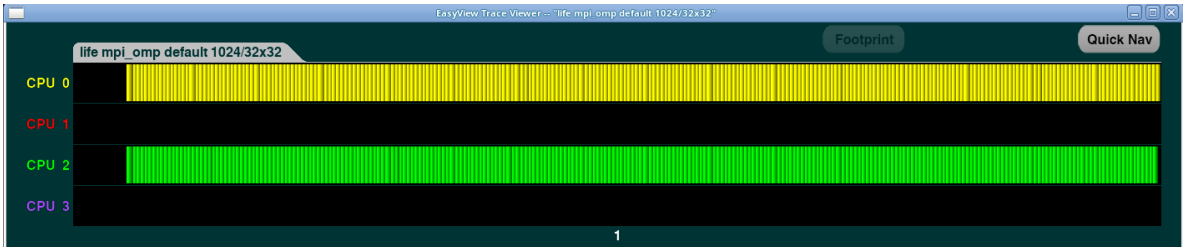

Figure 3: Trace d'exécution avec utilisation combinée de MPI et OpenMP

La trace d'exécution de la version hybride montre une réduction significative du temps passé en communication par rapport à la version MPI pure. L'utilisation efficace des cœurs disponibles sur chaque nœud permet de maximiser le travail utile effectué entre les phases de communication, améliorant ainsi le rendement global du système.

4.2 Impact du Ratio Processus/Threads

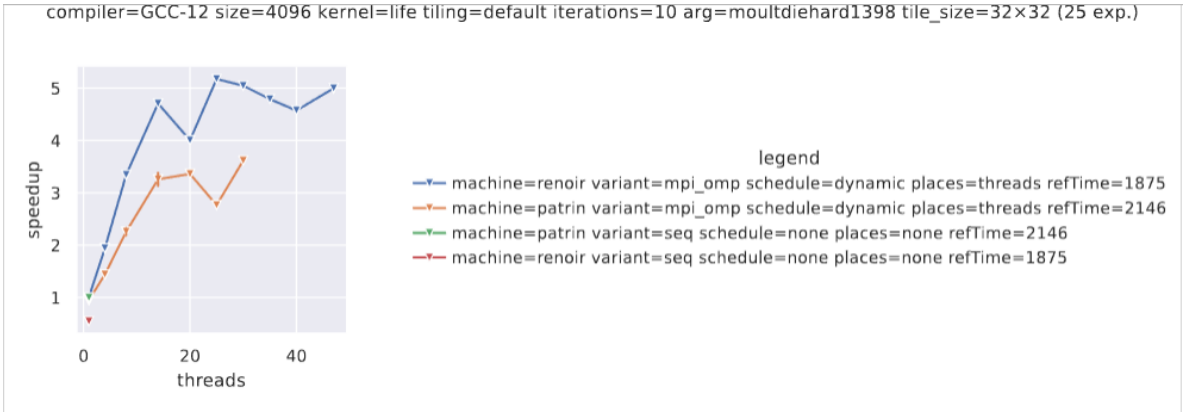

Figure 4: Évolution des performances en fonction du nombre de threads avec 4 processus MPI

Notre étude du ratio optimal entre processus MPI et threads OpenMP révèle qu'une configuration avec 25 threads par processus et 3 processus MPI offre les meilleures performances sur notre architecture de test. Ce résultat souligne l'importance de trouver le bon équilibre entre parallélisme à gros grain (MPI) et à grain fin (OpenMP).

Plusieurs facteurs influencent ce ratio optimal :

- La topologie du réseau et sa latence
- La hiérarchie mémoire des machines utilisées
- Les caractéristiques de localité de l'application
- Le surcoût de gestion des threads et des processus

Cette configuration optimale minimise les communications inter-processus tout en exploitant efficacement le parallélisme intra-nœud, ce qui est particulièrement important pour les applications à forte intensité de calcul comme le Jeu de la Vie.

4.3 Bords Épais

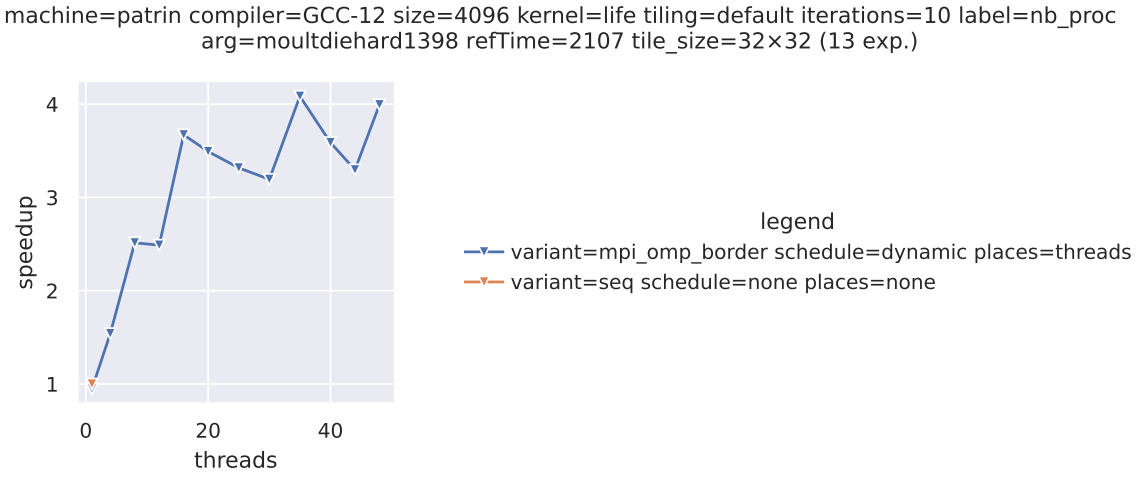

Figure 5: Impact de la taille des bords sur les performances avec 3 processus MPI

L'utilisation de bords épais montre une amélioration significative des performances, particulièrement pour les configurations avec un grand nombre de processus. Cette approche permet de réduire substantiellement la fréquence des communications, ce qui est bénéfique sur les réseaux à haute latence.

Notre analyse montre que l'épaisseur optimale des bords dépend de plusieurs facteurs :

- Le coût relatif des communications par rapport au calcul
- La taille du problème et sa distribution
- Le nombre de processus impliqués
- Les caractéristiques du réseau d'interconnexion

Pour notre configuration, une épaisseur de bord de 16 cellules offre le meilleur compromis entre réduction des communications et surcoût de calcul redondant.

5 Conclusion

5.1 Bilan des Optimisations

Notre implémentation distribuée du Jeu de la Vie a démontré l'efficacité de plusieurs stratégies d'optimisation :

- L'approche hybride MPI+OpenMP qui exploite efficacement la hiérarchie du parallélisme, permettant d'adapter le modèle de programmation à l'architecture matérielle sous-jacente

- Le recouvrement des communications par le calcul qui masque la latence réseau, réduisant ainsi l'impact des communications sur les performances globales
- L'utilisation de bords épais qui diminue la fréquence des communications tout en maintenant la précision des résultats
- L'ordonnancement dynamique des tâches qui améliore l'équilibrage de charge entre les threads

Ces optimisations combinées nous ont permis d'obtenir une scalabilité satisfaisante jusqu'à plusieurs dizaines de cœurs, démontrant ainsi le potentiel de l'approche hybride pour les applications scientifiques de grande échelle.

5.2 Limites et Contraintes

Notre implémentation présente certaines limitations :

- Nécessité que DIM soit divisible par le nombre de processus pour garantir une répartition équilibrée de la charge
- Surcoût de communication qui devient dominant à grande échelle, limitant la scalabilité au-delà d'un certain nombre de processus
- Dépendance forte aux caractéristiques du réseau d'interconnexion, rendant l'optimisation dépendante de la plateforme
- Complexité accrue de l'implémentation par rapport aux versions purement OpenMP ou MPI, nécessitant une compréhension approfondie des deux modèles de programmation

Ces limitations sont inhérentes à la nature distribuée du problème et constituent des défis communs à de nombreuses applications scientifiques parallèles.

5.3 Perspectives d'Amélioration

Pour des travaux futurs, plusieurs pistes pourraient être explorées :

- Implémentation d'échanges de bords épais avec une fréquence adaptée dynamiquement en fonction de l'évolution du modèle
- Intégration d'OpenCL pour exploiter les accélérateurs GPU, offrant ainsi un niveau supplémentaire de parallélisme adapté aux calculs intensifs
- Développement d'un algorithme de load balancing dynamique pour s'adapter aux configurations irrégulières ou aux évolutions asymétriques du modèle
- Optimisation fine des communications non-bloquantes et du recouvrement pour maximiser le chevauchement entre calcul et communication
- Exploration de stratégies de décomposition de domaine alternatives, comme le découpage en blocs 2D, qui pourrait réduire le volume de communication pour certaines topologies

Cette étude nous a permis d'explorer en profondeur les problématiques liées à la parallélisation distribuée d'applications scientifiques, et de mettre en œuvre des solutions qui exploitent efficacement les architectures modernes multi-nœuds et multicœurs. Les compétences acquises et les stratégies développées sont transférables à une large gamme d'applications de calcul haute performance.