High Performance Computing - Micro Aevol

Téo Bouvard

January 31, 2021

1 Introduction

Micro Aevol est une version réduite du modèle biologique Aevol[1], dont l'objectif est d'étudier le processus d'évolution à l'aide de la simulation d'organismes. Cette modèle réduit a pour but d'étudier l'optimisation de la performance du code sous-jacent au modèle.

2 Version CPU

Dans cette partie, on étudie l'optimisation la version CPU du code d'Aevol. Le processeur utilisé est un AMD Ryzen 7 3700X avec 32GB de RAM, le code est compilé à l'aide de CMake 3.19.1, g++ 10.2.0 et OpenMP 4.5 (201511)

2.1 Analyse

Afin de se familiariser avec le code, il est intéressant de profiler son exécution. Cela permet d'identifier la structure globale du programme ainsi que son chemin critique.

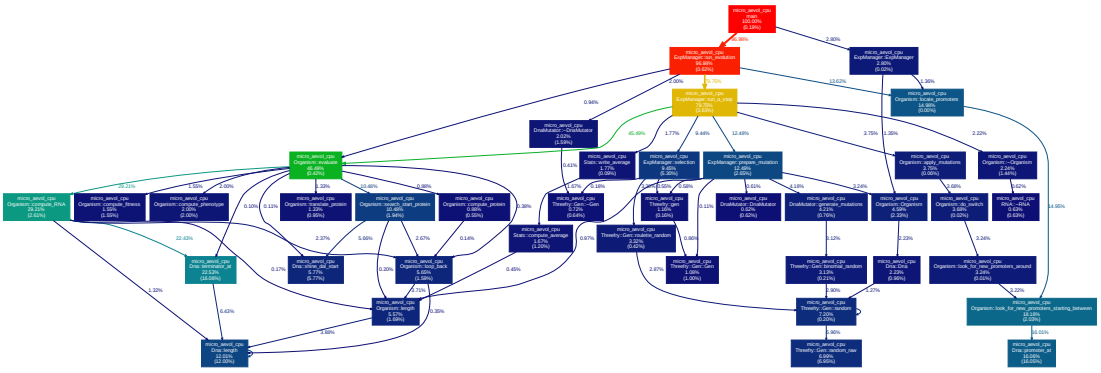

Figure 1: Profilage de l'application avec les paramètres par défaut à l'aide de *perf*

Dans le cas d'Aevol, on remarque que la fonction *run_a_step* est exécutée pour chaque pas de temps de *run_evolution*, dont le nombre d'itérations est spécifié par l'argument *n_steps*. Dans cette fonction, les différentes étapes du modèle biologique sont exécutées pour chaque organisme. Parmis ces étapes, on observe *Organism::compute_RNA* est celle qui cumule le plus de temps de calcul, avec 29% du temps d'exécution passé dans cette fonction et ses descendants.

Parallélisation du modèle

Toutes les étapes de ce modèle sont indépendantes pour chaque organisme sauf l'étape de sélection qui requiert la lecture de l'état des organismes voisins sur la grille. Une optimisation naturelle est donc de paralléliser le traitement de chaque organisme.

Échange de population

Pour effectuer la phase d'échange des populations lors du passage d'un pas de temps au suivant, il peut être plus performant d'échanger les pointeurs sur les grilles d'organismes plutôt que de copier chaque organisme dans une boucle.

Latence du disque

Une autre simple optimisation est de limiter les appels à *printf*, qui ralentissent l'exécution de manière significative lorsque ces derniers sont très fréquents. En effet, en utilisant les arguments par défaut on diminue de moitié le temps d'exécution en redirigeant la sortie standard ainsi que la sortie d'erreur vers */dev/null*. Bien évidemment, cette optimisation n'est pas aussi significative lorsque l'on utilise des tailles de problèmes plus élevées, mais il est sage d'éviter des appels systèmes coûteux lorsqu'ils ne sont pas nécessaires. Une alternative à la redirection des streams serait d'ajouter un argument permettant de contrôler la verbosité du programme, et ainsi s'affranchir du coût de tous ces appels systèmes lors d'une exécution en mode silencieux.

De manière similaire, les écritures dans les fichiers de statistiques invoquant *std::endl* ou *std::flush* imposent la synchronisation des flux de sortie, ce qui peut avoir un coût d'appel système assez élevé lié à la lenteur de l'écriture sur disque. Une meilleure solution serait de stocker les statistiques en mémoire jusqu'au point de backup afin de les écrire dans des fichiers d'un seul coup, ce qui permettrait d'améliorer le buffering déjà proposé par l'OS.

2.2 Implémentation

Dans l'optique de mesurer uniquement l'optimisation de la performance du modèle biologique, les benchmarks qui suivent n'incluent pas le temps d'écriture des fichiers de backup. En effet, cette écriture est réalisée par un seul thread et peut nécessiter la compression de plusieurs GB de données à l'aide de zlib, ce qui influe de manière significative sur le temps d'exécution du programme. Cependant, c'est un coût qui n'est pas directement lié au modèle biologique que l'on cherche à optimiser, et qui n'est effectué qu'à chaque point de backup. Si l'on souhaitait optimiser cette partie du programme, il pourrait s'avérer utile d'utiliser un algorithme de compression plus performant que DEFLATE tel que *lz4* ou *zstd*. Comme on peut l'observer sur la figure 2, l'utilisation de zstd 1.4.3 peut multiplier par 4 la vitesse de compression maximale de zlib tout en améliorant le ratio de compression de 36.4% à 34.7%. Si l'on se permet de sacrifier de la mémoire pour gagner en performance, l'utilisation de *lz4* permet de multiplier par 7 (respectivement 10) la vitesse de compression en obtenant un ratio de compression de 50.5% (respectivement 62.1%).

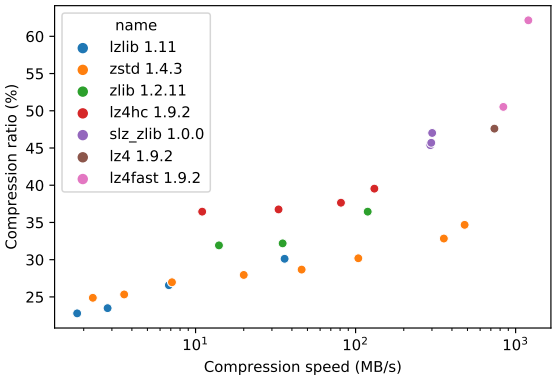

Figure 2: Comparaison d'outils de compression sur le corpus Silesia[2]

De manière similaire, le passage à l'échelle n'est mesuré que pour des tailles de problèmes tenant en mémoire. Au delà d'une taille de population supérieure à 512×512 et d'un génome comprenant plus de 50000 gènes, la taille des structures de données dépasse la capacité des 32 GB de RAM, et la mesure des temps d'exécution serait influencée par la vitesse du disque utilisé en swap.

2.2.1 Parallélisation du modèle

Chaque boucle itérant sur l'ensemble des organismes de la simulation est parallélisée à l'aide d'une directive *omp parallel for*. On cherche tout d'abord à déterminer la stratégie de scheduling optimale pour la paraléllisation de ces boucles. Pour cela, on mesure le temps d'exécution moyen par pas de temps, pour différentes tailles de population et de génomes, à l'aide d'un nombre de threads allant de 1 à 8. Pour rendre compte de la variation du nombre d'individus ainsi que de la taille de génome sur une variable unidimensionnelle, on introduit la variable du nombre total de gènes, définie de la manière suivante.

$$n_{genes} = n_{organismes} \times taille_{genome}$$

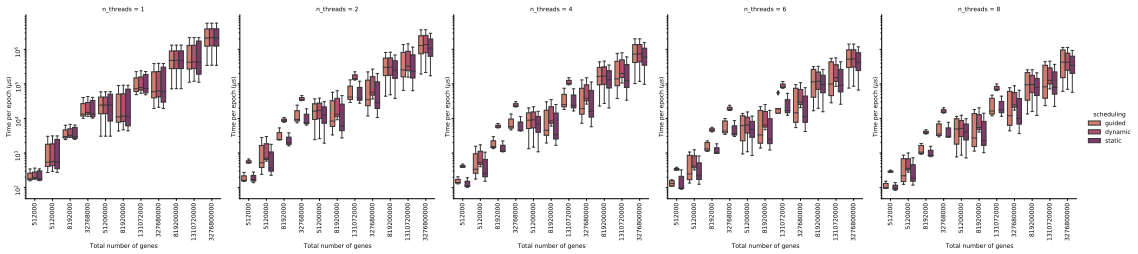

Figure 3: Benchmarks des différentes stratégies de scheduling d'OpenMP

Sur la figure 3, on observe que les meilleurs temps moyens par époque sont obtenus lorsqu'on utilise une stratégie de scheduling statique. La seconde stratégie la plus performante est celle du scheduling guidée, suivie du scheduling dynamique. Ce résultat peut être expliqué en observant la répartition des charges de calcul sur les différents organismes.

- Lors des phases de sélection, de swap de population et de recherche de meilleur individu, les calculs sont identiques pour chaque organisme.
- La phase d'évaluation n'est effectuée que si l'organisme a subi une mutation, ce qui est lié à un processus aléatoire. De plus, cette phase est d'autant plus coûteuse que le nombre de mutations subies est élevé. Cependant, chaque organisme est soumis au même processus aléatoire de mutation.

Ces deux observations suggèrent que la charge de calcul est répartie de manière uniforme sur l'ensemble de la population à chaque époque, et qu'il sera donc plus performant de diviser le traitement en chunks de grande taille pour maximiser l'occupation des processeurs plutôt que d'opter pour des stratégies de scheduling dynamiques ou guidées dont le load balancing nécessiterait un overhead inutile dans ce cas particulier.

On utilise donc cette stratégie pour le évaluer la performance de la parallélisation du modèle. Sur la figure 4, la première observation qui peut être réalisée est que toutes choses étant égales par ailleurs, le temps d'exécution du programme augmente avec le taux de mutation. Cela s'explique par la phase coûteuse d'évaluation qui n'est effectuée qu'en cas de mutation. On observe aussi que cette stratégie de paraléllisation fonctionne, car le temps d'exécution est inversement proportionnel au nombre de threads utilisés.

Strong scaling

Le speedup de cette parallélisation est mis en évidence sur la figure 5, qui visualise le gain de temps sur différentes tailles de problèmes pour un nombre croissant de threads. Ce speedup S est défini par $S_n = \frac{t_1}{t_n}$, où n représente le nombre de threads et t le temps d'exécution.

On remarque une accélération d'un facteur de 5 pour une multiplication par 8 du nombre de threads. Cette accélération n'est pas linéairement proportionnelle au nombre de threads car le code contient des parties intrinsèquement séquentielles qui ne peuvent pas être parallélisées. On observe plutôt un speedup de la forme

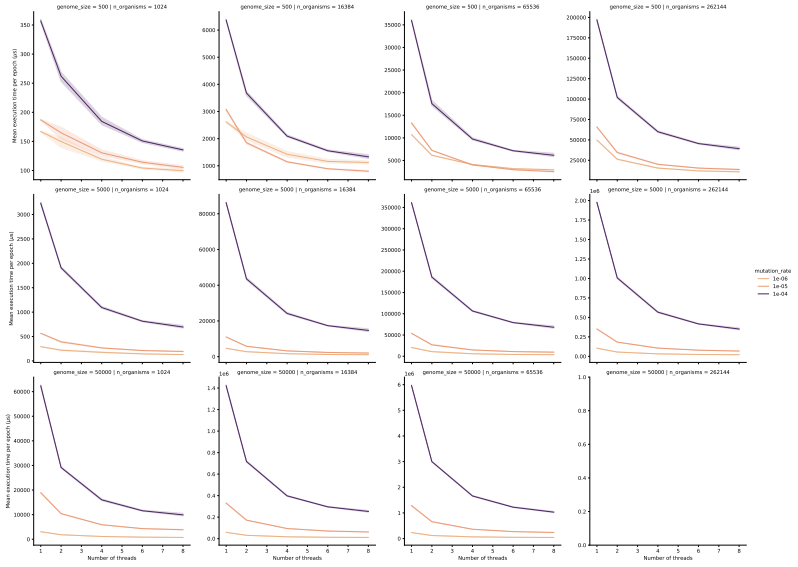

Figure 4: Benchmarks de la parallélisation du modèle biologique à l'aide d'OpenMP

$$S = \frac{1}{s + \frac{1-s}{n}}$$

où s représente le temps nécessaire à l'exécution de la partie séquentielle du code, $1-s$ le temps nécessaire à l'exécution de la partie parallèle et n le nombre de threads utilisés.

Ce phénomène a été théorisé par Amdahl[3] et est connu aujourd'hui sous le nom de loi d'Amdahl. Pour chaque valeur des différentes variables, on détermine s à l'aide d'une régression par moindres carrés de la loi d'Amdahl sur les données obtenues lors des benchmarks.

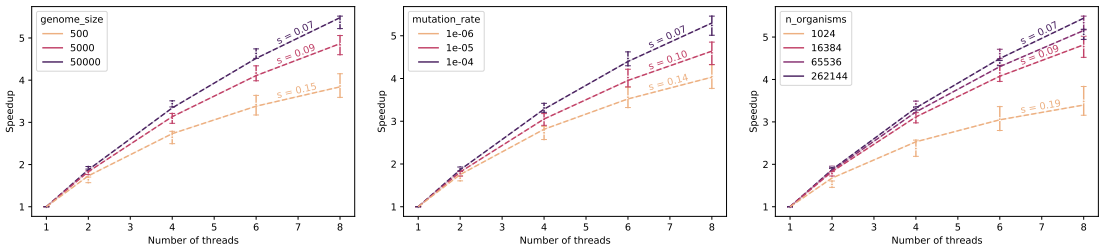

Figure 5: Speedup de la parallélisation du modèle biologique à l'aide d'OpenMP

Ces valeurs de s nous permettent de déterminer le speedup théorique maximal pour les tailles de problèmes étudiées. Par exemple, pour un génome comportant 50000 gènes, on peut calculer

$$S_{max} = \lim_{n\to\infty} \frac{1}{0.07 + \frac{0.93}{n}} = 14.3$$

cependant, si l'on fixe la taille du génome à 500 gènes, on obtient un speedup théorique maximal de

$$S_{max} = \lim_{n\to\infty} \frac{1}{0.15 + \frac{0.85}{n}} = 6.7$$

On en conclue que pour obtenir un speedup maximal en utilisant cette parallélisation, il est nécessaire d'utiliser des tailles de génome et de population les plus grandes possibles. On en conclue aussi que plus le taux de mutation est élevé, plus cette parallélisation est efficace.

Weak scaling

Le paradigme d'analyse du speedup utilisant des tailles de problème fixées, proposé par Amdahl, n'est cependant pas le seul moyen de quantifier l'efficacité de la parallélisation. L'observation théorisée par Gustafson[4] est que l'utilisation d'un plus grand nombre de processeurs permet de résoudre des problèmes de taille plus grande. Selon lui, une métrique plus pertinente est le scaled speedup S', défini de la manière suivante

$$S' = s + (1 - s) \times n$$

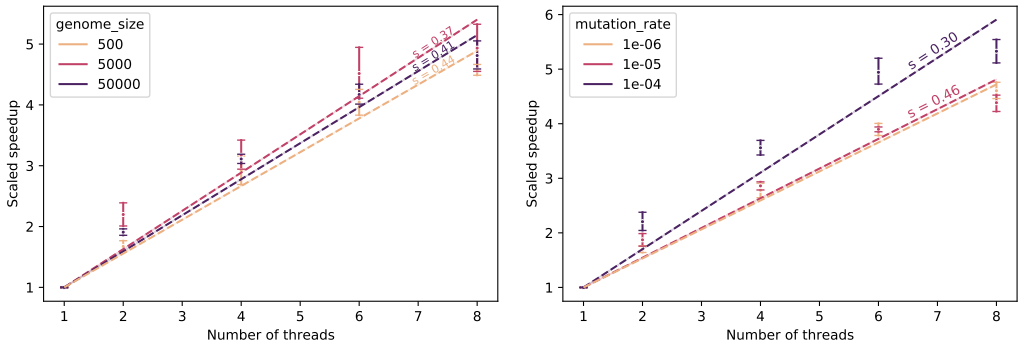

Figure 6: Scaled speedup de la parallélisation du modèle biologique à l'aide d'OpenMP

Le scaled speedup quantifie le temps qu'il faudrait à un seul thread pour exécuter le programme que l'on exécute à l'aide de plusieurs threads. La figure 6 montre l'évolution de ce scaled speedup lorsque l'on fixe la taille du génome et le taux de mutation. Sur cette figure, on observe la tendance linéaire de la loi de Gustafson jusqu'à 6 threads, mais on remarque un ralentissement du speedup lors de l'utilisation de 8 threads.

Contrairement à la loi d'Amdahl qui restreint le speedup maximal atteignable pour une taille de problème fixée, la loi de Gustafson nous fournit une métrique plus encourageante qui ne limite pas le speedup théorique, car l'ajout de nouveaux threads nous permettra de résoudre des problèmes toujours plus grands.

La figure 7 offre une comparaison de l'approche d'Amdahl et de Gustafson pour quantifier le speedup d'un programme parallélisé.

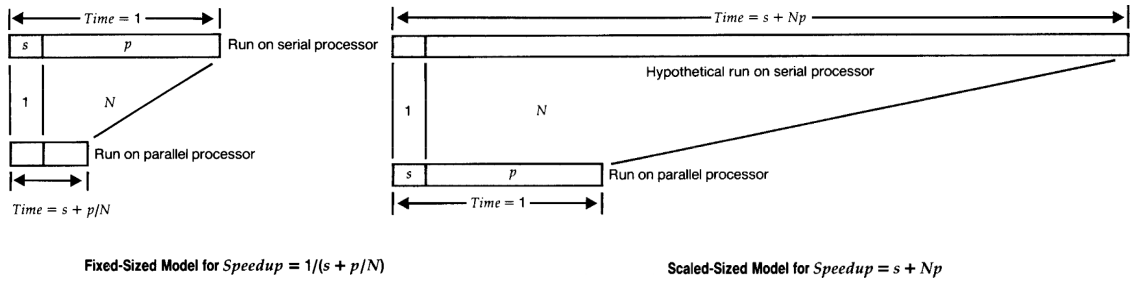

Figure 7: Modèles à taille de problème fixe (Amdahl) et à temps d'exécution fixe (Gustafson)

2.2.2 Échange de population

La figure 8 montre la distribution du temps par époque pour deux stratégies d'échange de population. La première stratégie *for_swap* consiste à itérer sur les organismes de la population et d'échanger les génomes de la génération n et $n-1$. La seconde stratégie *pointer_swap* consiste à échanger directement les pointeurs sur les populations de la génération n et $n-1$. On suppose que la taille du génome ainsi que celle de la population peuvent avoir une influence sur la performance de cette optimisation, ainsi la figure 8 présente les temps d'exécution pour le nombre total de gènes.

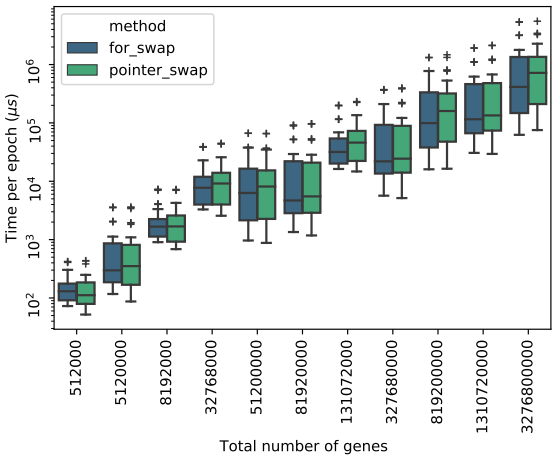

Figure 8: Stratégies d'échange de population

Bien que la meilleure performance par taille de problème soit généralement atteinte par l'échange de pointeurs, on observe que la performance médiane de cette stratégie est statistiquement moins bonne que celle de la boucle d'échange. L'hypothèse émise pour expliquer ce résultat est que le compilateur est capable de fournir une optimisation de la boucle d'échange au moins aussi bonne que la stratégie du *pointer_swap*.

3 Version GPU

Dans cette partie, on étudie l'optimisation la version GPU du code d'Aevol. Le processeur utilisé est un Intel Core i7-9700 avec 16GB de RAM, le code est compilé à l'aide de CMake 3.13.14, g++ 8.3.0 et nvcc 9.2.

3.1 Analyse

De même que pour la version CPU, on analyse le comportement du programme en profilant son exécution. Le profiler NVVP nous indique que le kernel *evaluate_population* occupe 99.7% du temps d'exécution. On observe sur la figure 9 le temps d'exécution moyen de chaque kernel constituant le modèle, sur une échelle de temps logarithmique.

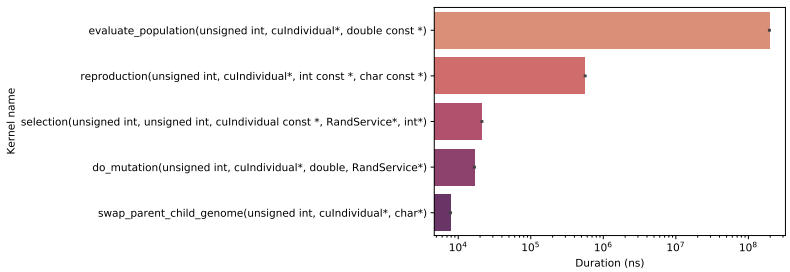

Figure 9: Logarithme du temps d'exécution moyen de chaque kernel

Ce temps d'exécution important s'explique principalement par le haut degré de divergence dans le code de ce kernel. En effet, certaines parties du kernel sont systématiquement exécutées par un seul thread dans chaque warp, ce qui augmente significativement le nombre d'instructions devant être exécutées par chaque warp[5].

Afin de profiler davantage les kernels les plus couteux pour identifier plus précisément les phases de divergence inter-warp, il est nécessaire de bénéficier des privilèges administrateur, qui ne nous sont pas accordés sur les machines du département.

3.2 Conclusion

La parallélisation mise en place pour la version CPU d'Aevol permet d'obtenir un speedup satisfaisant pour une introduction de complexité minimale à l'aide d'OpenMP. Ce speedup serait d'autant plus intéressant si la version GPU exécutait le traitement de chaque organisme de manière indépendante, mais l'architecture de CUDA limite la facilité de parallélisattion pour un processus qui comprend une phase d'aléatoire et donc un degré de divergence élevé.

References

[1] David Parsons, Guillaume Beslon, and Carole Knibbe. Aevol. http://www.aevol.fr/, 2012.

[2] Przemyslaw Skibinski. lzbench : an in-memory benchmark of open-source LZ77/LZSS/LZMA compressors. https://github.com/inikep/lzbench, 2020.

[3] Amdahl, Gene M. Validity of the single processor approach to achieving large scale computing capabilities. 1967. URL https://doi.org/10.1145/1465482.1465560.

[4] John L. Gustafson. Reevaluating amdahl's law. 1988. URL https://doi.org/10.1145/42411.42415.

[5] NVIDIA Developer Documentation. CUDA C Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#branching-and-divergence.