

# High Performance Computing - Micro Aevol

Téo Bouvard

January 19, 2021

## 1 Introduction

Micro Aevol est une version réduite du modèle biologique Aevol<sup>1</sup>, dont l'objectif est d'étudier le processus d'évolution à l'aide de la simulation d'organismes. Cette modèle réduit à 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 DDR4 à 3200MHz, 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.

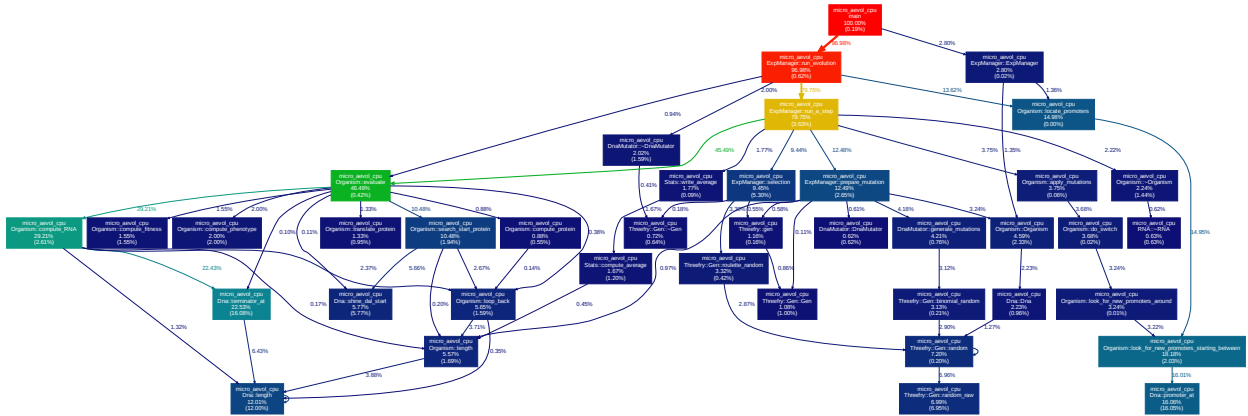


Figure 1: Profilage de l'application avec les paramètres par défaut à l'aide de *perf*

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.

### 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

Lorsque de la phase d'échange des population d'un pas de temps au suivant, il peut être plus performant d'échanger les pointeurs que de copier les éléments du tableau 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 stream 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 en batch, ce qui améliorerait 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 DEFLATE 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 serait 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, on peut multiplier par 7 (respectivement 10) la vitesse de compression en obtenant un ratio de compression de 50.5% (respectivement 62.1%).

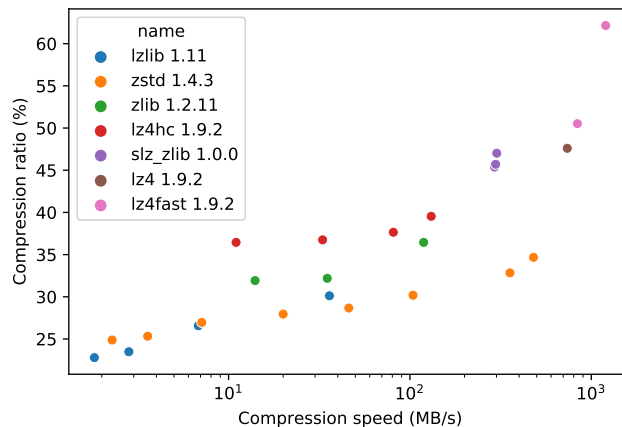


Figure 2: Comparaison d'outils de compression sur le corpus Silesia<sup>2</sup>

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  $512 \times 512$  et d'un génome de taille supérieure à 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*, en utilisant un scheduling statique. Ce choix de scheduling provient de l'observation de 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 couteuse que le nombre de mutations subies est élevé. Cependant, chaque organisme est soumis au même processus aléatoire de mutation, ce qui répartit les mutations de manière uniforme dans la population.

Ces deux observations suggèrent que la charge de calcul est répartie de manière uniforme sur l'ensemble de la population, et qu'il sera donc plus performant de diviser le traitement en chunks de taille égale plutôt que d'opter pour des stratégies de scheduling dynamiques ou guidées qui nécessiteraient un overhead inutile dans ce cas particulier.

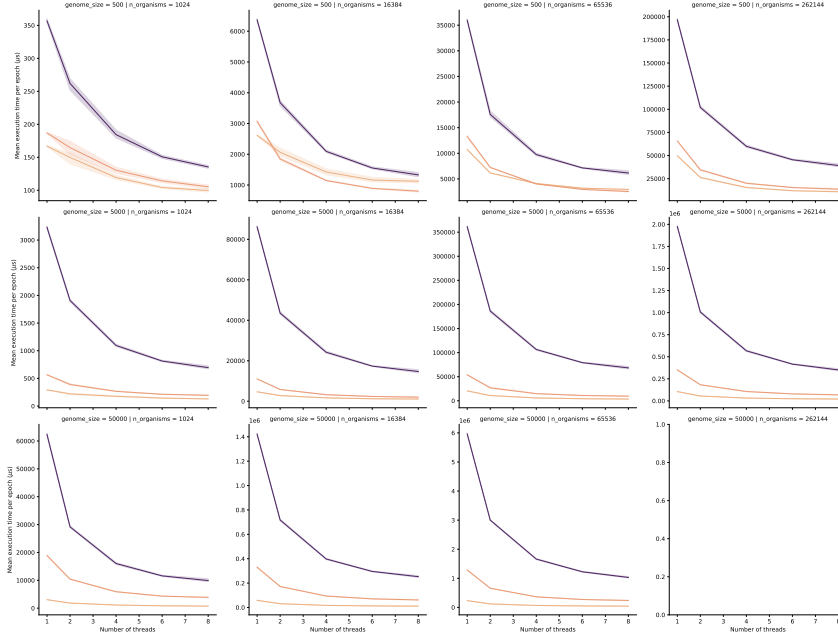


Figure 3: Benchmarks de la parallélisation du modèle biologique à l'aide d'OpenMP

Sur la figure 3, 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 d'évaluation qui n'est effectuée qu'en cas de mutation. On observe aussi un passage à l'échelle fort de cette parallélisation, car le temps d'exécution est inversement proportionnel au nombre de threads utilisés.

Ce passage à l'échelle est mis en évidence sur la figure 4, qui visualise l'agrégation des speedups des différentes tailles de problème 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.

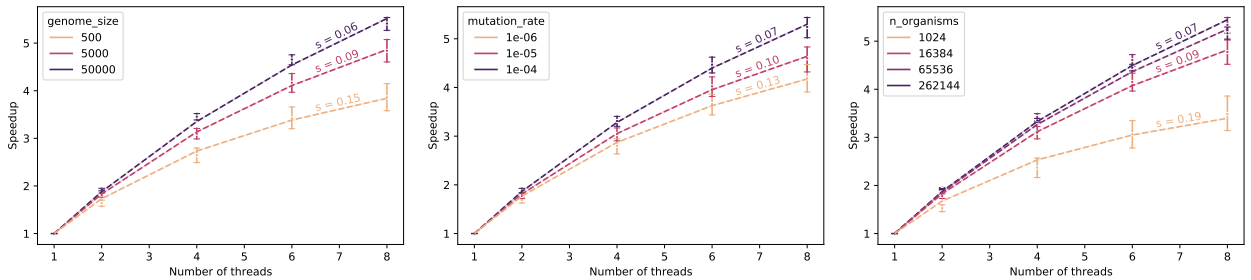


Figure 4: Speedup de la parallélisation du modèle biologique à l'aide d'OpenMP

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 être parallélisées. On observe plutôt un speedup de la forme

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

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<sup>3</sup> 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.

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

$$S_{max} = \lim_{n \rightarrow \infty} \frac{1}{0.06 + \frac{0.94}{n}} = 16.7$$

cependant, si l'on fixe la taille du génome à 5000 gènes, on obtient un speedup théorique maximal de

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

On en conclue que pour obtenir une efficacité maximale de cette parallélisation, il est nécessaire d'utiliser des tailles de génome et de population les plus grandes possibles. Cependant, on en conclue aussi que plus le taux de mutation est élevé, moins la parallélisation est efficace, ce qui indiquerait la nécessité d'étudier une autre stratégie de parallélisation pour de la méthode *evaluate* pour des taux de mutation élevés.

### 2.2.2 Échange de population

La figure 5 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 5 présente les temps d'exécution pour le nombre total de gènes, défini par

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

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 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*.

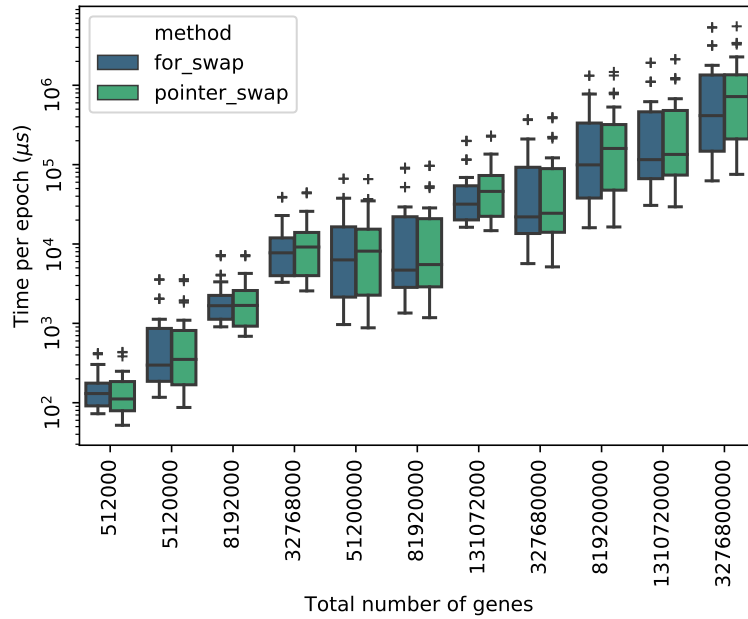


Figure 5: Stratégies d'échange de population

### 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 6 la logarithme du temps d’exécution moyen de chaque kernel constituant le modèle.

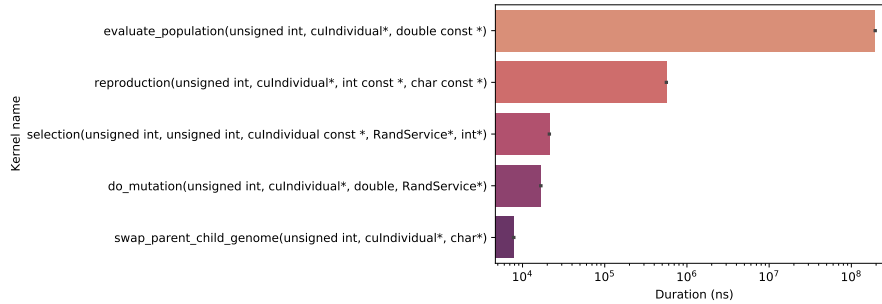


Figure 6: Logarithme du temps d’exécution moyen de chaque kernel

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 nombre d’instructions devant être exécutées par chaque warp<sup>4</sup>.

### 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. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, 2007.
- [4] NVIDIA Developer Documentation. CUDA C Best Practices Guide. <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#branching-and-divergence>.