Compte Rendu Blobwar 2023

Idées Essayées

Itérateurs parallèles avec `bridge`

Nous nous sommes immédiatement intéressés à la parallélisation de l'algorithme MinMax. Notre première solution a tout simplement été de bridger les itérateurs itératifs en itérateurs parallèles en utilisant les fonctions natives de `rayon`. Cela nous a immédiatement fourni un gain de temps considérable, mais cette solution n'est pas parfaite, comme détaillé dans les prochaines idées.

Itérateur nativement parallèle

Après l'utilisation du profiler `perf`, nous avons constaté que bridge utilisait près de 30% du temps processeur. Nous avons donc décidé d'essayer de créer un itérateur nativement parallèle sur les positions, pour essayer d'utiliser certaines connaissances sur l'itérateur afin que rayon optimise son utilisation. Comme nous allons le voir, cette démarche n'a pas abouti, mais nous avons décider de laisser le code "fantôme" qui devait être utilisé.

Nous nous sommes inspirés de ce site internet, qui décrit une implémentation manuelle des itérateurs parallèles sur les slices. Après l'avoir suivi avec nos propres traits adaptés à notre utilisation, nous nous sommes rendus compte qu'une information essentielle manquait : la longueur de l'itérateur. Après avoir essayé de calculer la taille à chaque demande ou de la stocker dans la struct après un premier calcul, nous avons donc essayé de trouver une méthode qui ne la nécessitait pas tout en donnait un size_hint qui pouvait fonctionner afin de tout de même optimiser la découpe en 2 de l'itérateur. Ces démarches étaient toutes plus lentes que la méthode `bridge`, que nous avons donc décidé de garder.

Afin de réduire le temps de découpe qui était trop long mais tout de même intéressant, nous avons fait le choix de n'utiliser des itérateurs parallèles que pour des profondeurs assez peu élevées, et de laisser les cas les plus finaux en itératif. Cela a donné un gain de temps considérable, c'est donc l'algorithme MinMax que nous avons gardé.

Memoïzation

A chaque itération des algorithmes MinMax et Alpha Beta, des coups étaient recalculés de nouveau, nous avons donc cherché à retenir les résultats calculés lors des coups précédents et ainsi gagner du temps. Nous nous sommes donc servis de la crate **memoize** qui est détaillée sur le lien suivant : https://crates.io/crates/memoize. Ainsi il nous fallait retenir les différentes configurations observées au cours de la partie dans une hashmap, ce qui a nécessité l'implémentation des Traits `std::hash::Hash` et `PartialEq`, dont les implémentations sont écrites dans le fichier **configuration.rs**. Cette alternative ne fonctionnait que si l'on passait les traits de vie utilisées dans le fichier **configuration.rs** en `'static` et ne nous a pas apporté d'améliorations de performance.

Évaluation des performances

Benchmarks

Nous avons réalisé un benchmark à l'aide de la crate **bench** , dans le fichier **versus_testing.rs** localisé dans le répertoire **benches**. Ce benchmark nous sert à calculer le temps moyen passé par coup à la fin d'une partie, il faut donc faire en sorte d'opposer deux même algorithmes ayant la même profondeur. Ce benchmark fonctionne très bien jusqu'à une profondeur de 3 pour MinMax, en revanche à partir de 4, il commence à prendre beaucoup de temps, et ne

termine pas, on lui soupçonne donc d'empêcher le parallélisme qui a été mis en place. Pour AlphaBeta qui est séquentiel, nous avons pu mponter jusqu'à 5, sans trop de problèmes.

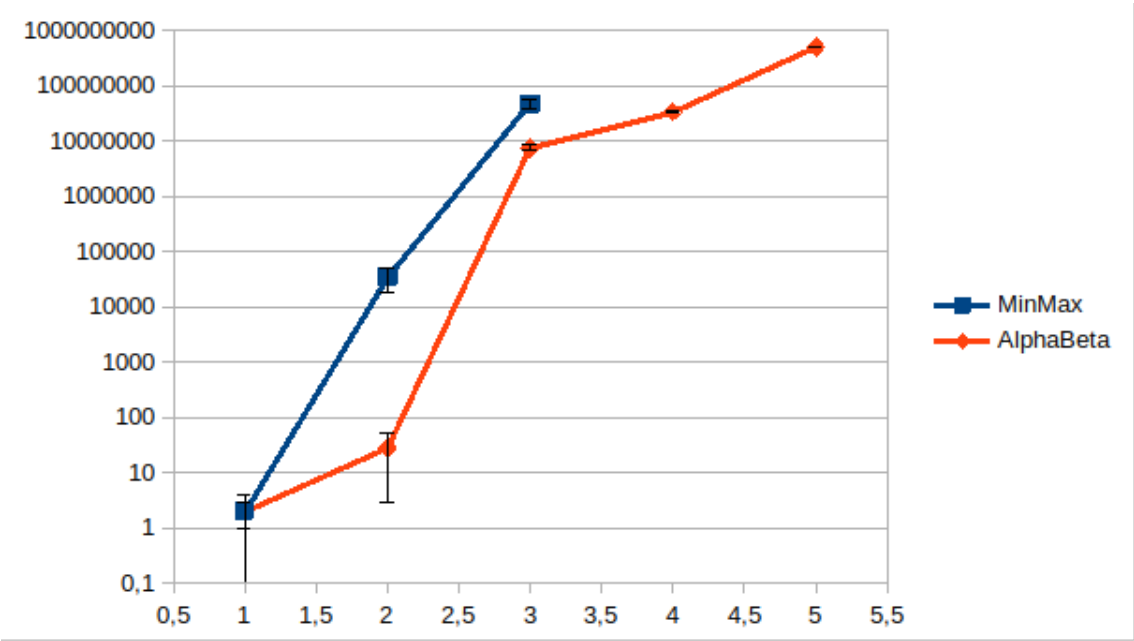

FIGURE 1 – Temps en ns par coups en fonction de la profondeur pour MinMax (pas parllèle) et AlphaBeta

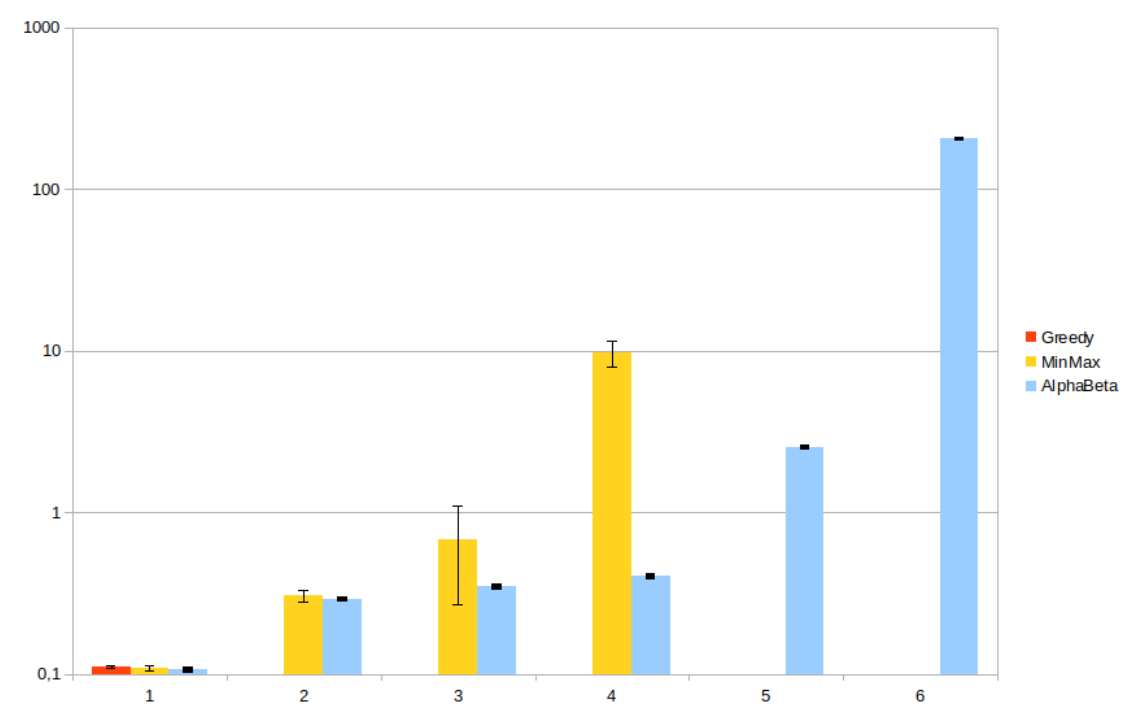

FIGURE 2 – Temps en s d'une bataille entre 2 instances de même profondeur (en abscisse) en fonction de la profondeur pour MinMax parallèle et AlphaBeta, mesuré par `time`