Université Joseph Fourier
GRENOBLE
UFR IM²AG

Compte Rendu de Projet de HPC : Parallélisation d'un Alpha-Bêta

LEFEVRE Henry – M2 MIA parcours GICAO
SEGURET Aymeric – M2 MIA parcours GICAO

Introduction

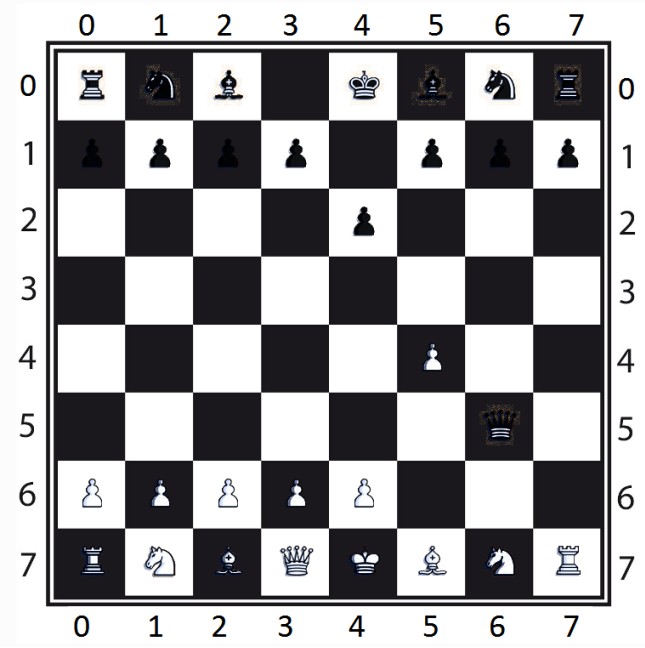
Dans le cadre du cours de HPC (High Performance Computing) enseigné à l'Ensimag par monsieur Picard Christophe, il a été demandé aux étudiants de réaliser un programme où le but était de paralléliser un algorithme de leur choix. Une fois l'algorithme choisi, les étudiants devaient réaliser le code, le tester et par la suite le paralléliser (et pourquoi pas tenter de suivre des pistes afin d'améliorer leurs résultats). L'évaluation du projet se faisant avec un rapport.

Nous avons choisi de paralléliser un Alpha-Bêta dans le contexte d'un jeu d'échec.
Afin de présenter le travail accompli nous commencerons par une présentation du travail préliminaire, nous continuerons avec un rapide rappel de l'algorithme et les différentes pistes exploitées tout au long du projet. Nous terminerons enfin avec la présentation des résultats ainsi que leurs critiques et les différentes idées que nous n'avons pas eues l'occasion de mettre en place.

I / Travail préliminaire :

Comme mentionné dans l'introduction nous avons choisi de paralléliser un algorithme d'Alpha-Bêta. L'élagage Alpha-Bêta (abrégé élagage αβ) est une technique permettant de réduire le nombre de nœuds évalués par l'algorithme minimax en théorie des jeux. Mais le rappel de l'algorithme sera détaillé dans la seconde partie.

Avant de nous lancer dans la parallélisation de l'algorithme, la première étape de notre travail a été la construction complète d'un jeu d'échec avec un affichage simple. Cela nous permettant par la suite de vérifier visuellement la justesse des résultats obtenus par notre algorithme.



Une fois cela accompli, nous avons commencé quelques tests et de la recherche bibliographique sur le sujet car la parallélisation de cet algorithme semble aujourd'hui encore être un sujet de recherche. Nous avons ainsi mis en forme une liste d'idées à exploiter lors de la parallélisation.

II / Rappel de l'algorithme :

Comme mentionné précédemment l'élagage Alpha-Bêta est une technique permettant de réduire le nombre de nœuds évalués par l'algorithme minimax.

Le Principe d'un MinMax :

Le principe de la recherche d'un coup par MinMax est un processus de réflexion très naturel que tout un chacun a déjà mis en œuvre. Etant donné une position, on suppose que c'est au joueur noir de jouer. Dans la position donnée, Noir a une série de coups qu'il peut effectuer : pour chacun d'eux, il s'interroge sur les répliques éventuelles que peut faire Blanc, qui lui-même analyse pour chacune de ses répliques celles auxquelles peut procéder Noir, qui à son tour examine à nouveau l'ensemble des coups qu'il peut effectuer suite aux répliques de Blanc, etc. Ce processus de réflexion peut continuer ainsi et, dans le cas où les deux joueurs ont la capacité de mener cette réflexion jusqu'à ce qu'aucun des 2 joueurs ne puisse plus jouer alors il est facile pour Noir de décider du coup qu'il doit jouer. Il lui suffit en effet de choisir le coup qui, s'il existe, quelque soit la suite de répliques de Noir et Blanc imaginée mène à une victoire.

Ce processus de réflexion est généralement associé à un arbre de jeu tel que celui présenté sur la figure 1. Chaque nœud y correspond à une position de jeu et les branches correspondent aux différents coups que peut faire Noir ou Blanc à partir de cette position. Les feuilles sont les nœuds terminaux, desquels ne partent aucune branche, et correspondent dans la situation où une recherche exhaustive peut être conduite jusqu'à la fin de la partie. Des valeurs V ou D pour victoire ou défaite, du point de vue de Noir, sont indiquées au niveau des nœuds terminaux : ces valeurs précisent donc l'issue de la séquence de coups à partir du nœud racine (i.e. le sommet de l'arbre).

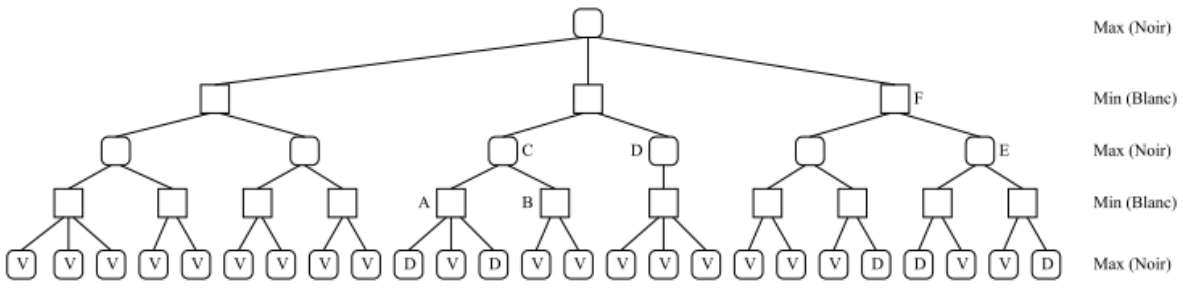


Figure 1 – Arbre de recherche MinMax développé jusqu'à la fin de la partie. V ou D indique si la position terminale est une victoire de Noir ou bien une défaite. Un nœud Max correspond au cas où le joueur en question (ici, Noir) essaie de maximiser son gain alors qu'un nœud Min correspond au cas inverse.

Notons que cette situation de recherche jusqu'à l'atteinte de positions terminales ne peut exister que lorsque mise en œuvre à partir d'une position qui est proche de la fin de la partie. Il est facile d'imaginer que l'arbre de jeu ne peut être développé à partir de positions où le plateau de jeu n'est que peu rempli, et, à fortiori, à partir de la position de départ. (C'est en revanche possible de le faire à partir de la position de départ pour des jeux de réflexion très simples tels que le morpion, qui se joue sur un damier 3×3.) Pour le reste de la partie, il est donc nécessaire de mettre en œuvre la stratégie MinMax en utilisant une profondeur d'arbre prédéfinie et une fonction d'évaluation (exemple dans la figure 2).

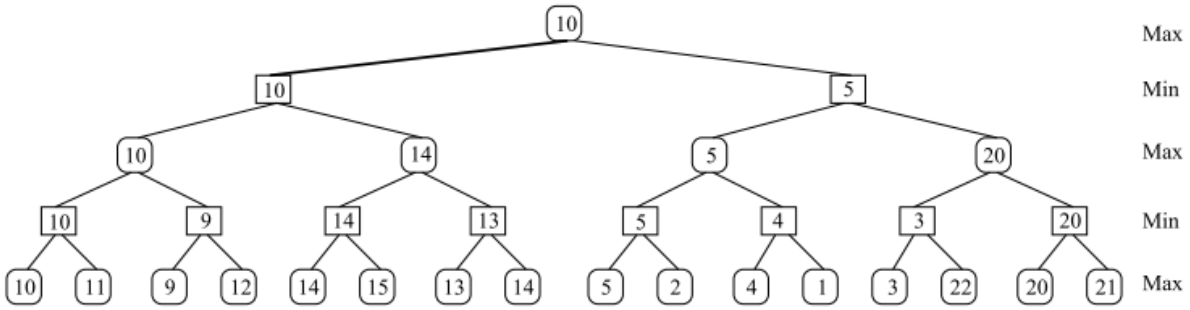


Figure 2 – Application de l'algorithme MinMax en utilisant les notes obtenues par une fonction d'évaluation.

L'élagage Alpha-Bêta :

Le problème de MinMax est que les informations ne circulent que dans un seul sens : des feuilles vers la racine. Il est ainsi nécessaire d'avoir développé chaque feuille de l'arbre de recherche pour pouvoir propager les informations sur les scores des feuilles vers la racine.

Le principe de l'élagage AlphaBeta, que nous appellerons par abus de langage algorithme Alpha-Beta, est précisément d'éviter la génération de feuilles et de parties de l'arbre qui sont inutiles. Pour ce faire, cet algorithme repose sur l'idée de la génération de l'arbre selon un processus dit en « profondeur d'abord » où, avant de développer un frère d'un nœud (rappel : deux nœuds frères sont des nœuds qui ont le même parent), les fils sont développés. A cette idée vient se greffer la stratégie qui consiste à utiliser l'information en la remontant des feuilles et également en la redescendant vers d'autres feuilles.

Plus précisément, le principe de AlphaBeta est de tenir à jour deux variables α et β qui contiennent respectivement à chaque moment du développement de l'arbre la valeur minimale que le joueur peut espérer obtenir pour le coup à jouer étant donné la position où il se trouve et la valeur maximale. Certains développements de l'arbre sont arrêtés car ils indiquent qu'un des joueurs a l'opportunité de faire des coups qui violent le fait que α est obligatoirement la note la plus basse que le joueur Max sait pouvoir obtenir ou que β est la valeur maximale que le joueur Min autorisera Max à obtenir.

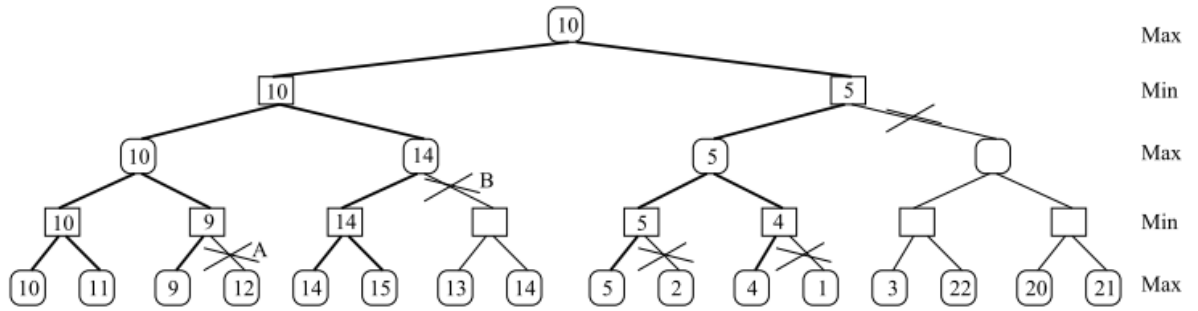


Figure 3 – Algorithme AlphaBeta : les branches développées sont en trait gras ; les autres parties de l'arbre sont celles développées par MinMax mais pas par AlphaBeta.

III / Pistes exploitées :

Ainsi il semble évident que paralléliser l'algorithme d'Alpha-Bêta va poser quelques problèmes et est un problème difficile.
La parallélisation d'un tel algorithme entraîne des « frais » dans trois domaines :

- La communication
- La synchronisation
- Les recherches inutiles

Le surcout de communication :
La communication entre processeurs est normale comme dans tout algorithme parallèle; Cependant, il existe un type de communication qui semble unique pour la parallélisation de l'alpha-bêta.
L'algorithme séquentiel alpha-bêta met à jour ses deux bornes, alpha et bêta, comme la recherche d'un arbre de jeu. Lors de la recherche en parallèle, si un processeur constate une amélioration d'alpha ou de bêta, il doit en informer les autres processeurs, travaillant en dessous de celui-ci, de sorte qu'ils puissent faire usage de la valeur qui vient d'être découverte.

Le surcout de synchronisation :
Le surcout de synchronisation résulte du fait qu'un processeur reste inactif en attendant qu'un événement se produise.
Par exemple, si quatre processeurs, P1, P2, P3 et P4, travaillent ensemble sur un même nœud, les processeurs P2, P3 et P4 attendent peut-être le résultat d'une recherche menée par le processeur P1 sur une branche de ce nœud. Ce qui peut provoquer des attentes et un surcoût dû à la communication.

Le surcout dû aux recherches inutiles :
La parallélisation d'un algorithme tel qu'alpha-bêta entraîne parfois des recherches inutiles qui auraient pu être évitées par la version séquentielle.
Lorsque la recherche est lancée en parallèle sur un même nœud, le meilleur « score » pourrait ne pas avoir été découvert. Par conséquent, la recherche est effectuée en parallèle avec une fenêtre plus large que dans le cas séquentiel. De plus, après que la recherche parallèle a été lancé, un processeur peut découvrir que celle-ci n'est plus nécessaire au niveau du nœud en raison d'une condition de coupure - les autres processeurs ont alors effectué un travail inutile.

Voici donc les grands problèmes de la parallélisation d'alpha-bêta. Mais comme dit précédemment, ce problème reste un sujet de recherche et après nos recherches nous avons trouvé des solutions potentielles :

- Principal Variation Splitting (PVS) : le principe est de débuter par la feuille « la plus à gauche » (un processeur effectue le calcul et les autres sont inactifs). Une fois cette branche examinée, les autres processeurs deviennent actifs et examinent les branches au niveau du nœud père de la feuille précédemment examinée (chaque processeur enlève une branche à la fois et détermine sa valeur). Si un processeur découvre une amélioration du score au niveau du nœud, il en informe les autres processeurs. Quand il n'y a pas de branches à analyser, un processeur qui est à court de travail reste inactif jusqu'à ce que les autres processeurs terminent. Une fois que toutes les branches ont été examinées, la recherche se déplace vers le parent du nœud courant. Ce processus se poursuit vers le haut de l'arborescence jusqu'à ce que toutes les branches au niveau du nœud racine aient été étudiées.

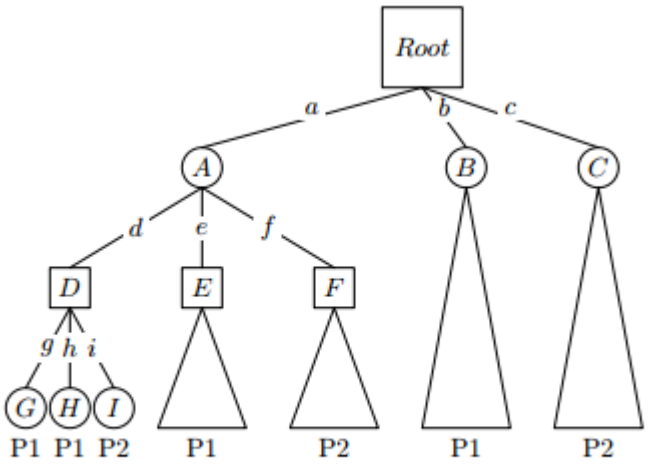


Figure 3.2: Two processors using PVSplit to divide up the work in a small tree.

- Young Brothers Wait Concept (YBWC) : Au début, un processeur a la propriété du nœud racine, tandis que les autres processeurs restent dans un état de repos. Un processeur, P1, qui est inactif sélectionne un autre processeur, P2, au hasard et transmet un message demandant du travail. Le processeur P2 a du travail disponible s'il y a au moins un nœud dans le sous-arbre qu'il examine qui satisfait le critère d'YBWC. Qui est « P2 a du travail disponible s'il possède un nœud dont le frère aîné a été évalué ». Le nœud qui satisfait le critère devient le point de partage; s'il y a beaucoup de nœuds qui satisfont le critère, alors le nœud qui est le plus haut dans l'arborescence est sélectionné comme point de partage. Si P2 a du travail disponible, une relation maître-esclave est établie entre P2 et P1. Le maître et ses esclaves partagent l'effort de recherche en un point de partage. Chaque processeur enlève une branche à la fois jusqu'à ce que la recherche en point de partage soit terminée.

- Dynamic Tree Splitting (DTS) : DTS utilise une approche peer-to-peer plutôt qu'une approche maître-esclave comme dans YBWC. Alors que de nombreux processeurs peuvent collaborer sur un nœud, le processeur qui termine la recherche est le dernier et doit retourner l'évaluation du nœud à son père.
  Au début, un processeur est configuré pour rechercher le nœud racine, tandis que les autres processeurs sont dans un état inactif. Un processeur inactif consulte une liste globale de « points divisés actifs » (SP-liste) pour trouver du travail à faire. Si un point de partage avec le travail est trouvé, le processeur inactif rejoint les autres processeurs qui travaillent à ce point de partage et le travail à ce nœud est partagé. Toutefois, si aucun travail ne peut être trouvé dans la SP- liste, le processeur inactif diffuse un message d'aide à tous les processeurs. Le processeur inactif examine ensuite la zone partagée afin de trouver un point de partage approprié. Si un point de partage peut être trouvé, le point de partage est d'abord copié dans la SP-liste et le processeur inactif partage alors le travail au point de partage avec le processeur qui a initialement développé le nœud. Si un point de partage approprié ne peut être trouvé dans la zone partagée, le processeur inactif diffuse un nouveau message d'aide après un petit retard.
  Si un processeur est le dernier processeur au point de partage, alors le processeur est responsable de retourner l'évaluation du nœud à son parent. En outre, ce processeur ne pénètre pas dans l'état de repos, mais continue à travailler au niveau du nœud parent.

Par manque de temps nous n'avons pas eu l'occasion de tester plusieurs des méthodes présentées ci-dessus. Nous avons donc choisi de nous concentrer sur la PVS, qui sera présentée par la suite.

IV /Etude des résultats & Axes d'améliorations :

Ainsi nous avons débuté notre parallélisation par une version multi-threadée, permettant ainsi de contourner les problèmes présentés ci-dessus. En effet avec une telle version il suffit de mettre en place une structure de donnée partagée à laquelle chaque thread puisse accéder afin de pouvoir réaliser un algorithme correct.

Dans un premier temps nous fixons une variable comme étant le nombre de threads que nous allons utiliser (ici 4), et nous créons la liste des coups possibles dans un vecteur de coups. Le but de ce vecteur est de créer une seule structure pour tous les coups possibles pour tous les threads et ainsi chacun aura accès à une partie des coups (d'où le calcul de min et de max dans la boucle pragma).
Les threads ont une variable partagée min qui est la valeur minimale renvoyée par l'algorithme et ainsi on lance classiquement l'algorithme MinMax pour chaque thread. Une fois que tous les threads ont terminé leurs calculs, on joue le meilleur coup extrait par l'algorithme.

La Figure 4 présente le code ainsi obtenu.

```
void IA_AlphaBeta::jouerParalleleAlphaBeta() {
    int nb_threads = 4;
    int value;
    int value_min = INT_MAX;
    std::vector< coup* > listCoup = p->getListCoup( false );

    int taille = listCoup.size();
    taille = taille / nb_threads;

    #pragma omp parallel num_threads(nb_threads) shared(value_min)
    {
        #pragma omp critical
        {
            int min = omp_get_thread_num() * taille;
            int max = listCoup.size();
            if( (min + 2 * taille) < max ){
                max = min + taille;
            }
            for (int i = min; i < max; ++i) {
                coup* c = listCoup.at(i);
                p->jouerCoup(c);

                value = ab_max_min(p, INT_MIN, INT_MAX, depth-1);

                p->getBack(c);

                if (value <= value_min) {
                    value_min = value;
                    this->bestCoup = c;
                }
            }
        }
    }

    p->jouerCoup( bestCoup );
    p->majPlateau();
}
```

Figure 4 – Version multithreadée

La seconde version que nous souhaitions réaliser était une version multiprocesseur mais cette fois multi-machine, avec cette fois-ci le problème présenté précédemment à prendre en compte. Malheureusement étant donné la complexité du problème et le grand nombre de projets que nous avions à réaliser en simultané nous n'avons pas réussi à obtenir une version définitive et fonctionnelle.

Comparaison entre les différentes versions :

Afin de comparer nos différentes versions nous avons défini un état du plateau, regardé si le coup joué était correct et comparer les temps obtenus :

La Figure 6 présente l'état du plateau avant et après application de nos algorithmes.
La figure 7 présente la méthode de récupération du temps d'exécution.
La figure 8 présente les résultats obtenus.

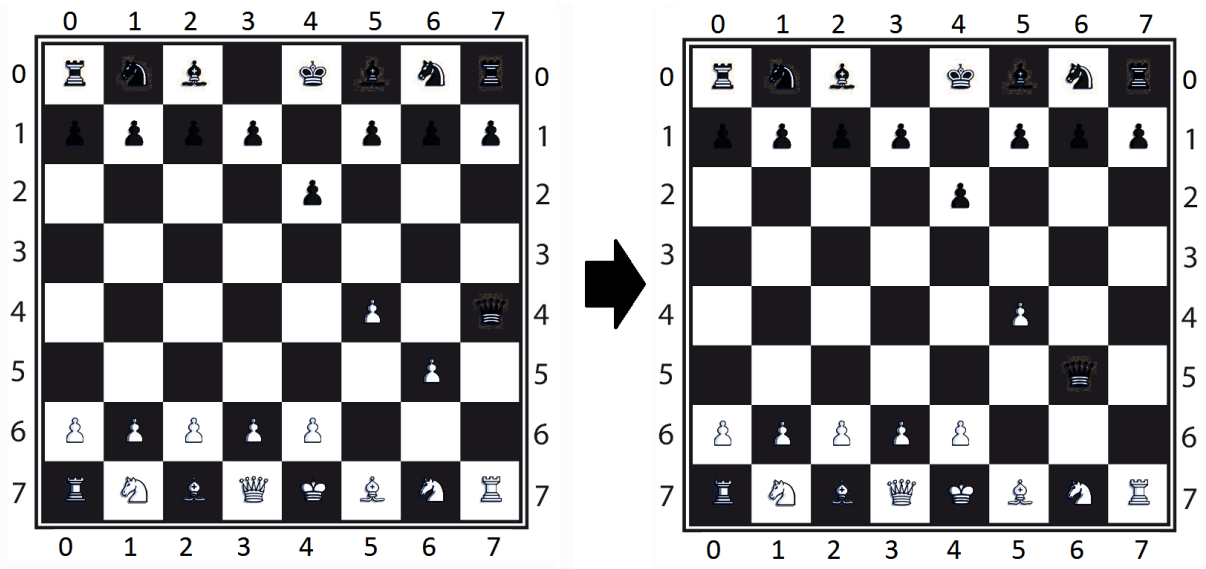


Figure 6 – Test Algorithmes

```
IA_AlphaBeta * cerveau = new IA_AlphaBeta(p);
QTime time;
time.start();
cerveau->jouerParalleleAlphaBeta();
//    cerveau->jouerAlphaBeta();
int temps = time.elapsed();
cout << temps << endl;
```

Figure 7 – Récupération temps

Version :	Coup correct :	Temps d'exécution (ms) :
Séquentielle	oui	700 - 750
Threadée	oui	650 - 700
Multiprocess	Non terminée	Non terminée

Figure 8 – Résultats obtenus

V / Avec plus de temps :

La version multiprocess n'ayant pas été terminée, nous commencerions par la terminer. En effet le déploiement du PVS est une solution complexe et les problèmes de communication et de synchronisation sont omniprésents et difficile à corriger.

Une fois cette version fonctionnelle, nous aurions souhaité améliorer nos versions parallèles, selon nous nous pouvons encore gagner du temps d'exécution et certains papiers trouvés sur internet confirment nos dires.

Sources :

https://chessprogramming.wikispaces.com/Parallel+Search

http://www.netlib.org/utk/lsi/pcwLSI/text/node350.html

http://www.top-5000.nl/ps/Parallel%20Alpha-Beta%20Search%20on%20Shared%20Memory%20Multiprocessors.pdf