# Décision de finales d'échecs

Charles Bouillaguet

23 février 2017

Le but du projet est de paralléliser un programme séquentiel (que nous vous fournissons). Ce programme prend en entrée une *position* du jeu d'échec, qu'il tente de *décider*, c'est-à-dire renvoyer l'une de ces trois options :

— Le joueur blanc peut gagner, quoi que fasse le joueur noir.
— Le joueur noir peut gagner, quoi que fasse le joueur blanc.
— La partie sera nulle.

De plus, une *variation principale* est calculée : ce sont les meilleurs coups possibles pour les deux joueurs.

Si vous n'êtes pas familier avec le jeu d'échecs, vous pouvez consulter la très bonne page Wikipédia sur le sujet [1].

On considère toutefois ici une version *simplifiée* du jeu d'échecs : les seules pièces autorisées sont les rois et les pions. La prise en passant [2] est ignorée. Les conditions de gain sont les suivantes :

— Blanc gagne s'il a un de ses pions en 8ème rangée au début de son tour.
— Noir gagne s'il a un de ses pions en 1ère rangée au début de son tour.
— Un joueur gagne s'il met l'autre échec et mat.

Une partie nulle sera prononcée dans l'une des condition suivantes :

— Le joueur dont c'est le tour n'a pas de coups légaux et il n'est pas échec (pat).
— Il n'y a plus de pions sur le plateau.
— Le coup qui vient d'être joué amène la reproduction d'une position antérieure de la partie.

Il est possible qu'une position décidée dans un sens avec les règles « officielles » soit décidée autrement avec ces règles modifiées.

## 1 Description du code séquentiel

Le code séquentiel est découpé en deux fichiers : `main.c` contient la procédure de recherche à paralléliser tandis que `aux.c` contient des fonctions auxiliaires que vous n'avez pas besoin de comprendre à fond (gestion du plateau, des pièces, génération des coups légaux à partir d'une position, détection de la fin de partie...). Les fonctions dont vous avez besoin sont décrites sommairement dans `projet.h`.

1. `https://fr.wikipedia.org/wiki/Echecs`
2. cf. Section « Règles Spéciales » de la page Wikipédia « Échecs ».

Le programme prend en unique argument une position du jeu décrite par une chaine de caractères, en notation Forsyth-Edwards [3]. De nombreux exemples sont fournis dans les fichiers `positions_v1.txt` et `positions_v2.txt`, avec leur temps de traitement séquentiel approximatif sur les machines des salles de TP.

La procédure `decide` fonctionne de la façon suivante : on tente de décider la position en s'autorisant à jouer $k$ coups. Si aucune conclusion ferme ne peut être atteinte, on incrémente $k$ et on recommence. À chaque essai, on affiche le « score » attribué à la position (plus il est grand, plus Blanc est près de gagner, plus il est négatif, plus Noir est près de gagner), ainsi que les quelques premiers coups optimaux pour les deux joueurs.

Pour une valeur de $k$ fixée, la fonction `evaluate` est invoquée. C'est le coeur de la procédure de recherche. Par défaut, il s'agit de l'algorithme *minimax* [4], présenté sous la forme « negamax ».

Deux `#define` situés au début de `projet.h` ont une influence majeure sur son fonctionnement :

- Si `ALPHA_BETA_PRUNING` vaut 1, alors la fonction `evaluate` utilise la technique de l'*élagage* $\alpha$-$\beta$ [5]. Ceci permet d'éviter l'exploration de positions inutiles.
- Si en plus `TRANSPOSITION_TABLE` vaut 1, alors la fonction `evaluate` utilise une *table de transposition* [6] pour éviter d'évaluer plusieurs fois la même position.

En fait, avec les deux `#define` à zéro, le code séquentiel implémente exactement le premier algorithme décrit sur la page Wikipédia (anglaise) « Negamax » [7]. Avec :

```
#define ALPHA_BETA_PRUNING 1
```

il s'agit du second, et avec :

```
#define TRANSPOSITION_TABLE 1
```

il s'agit du troisième.

Chacune de ces deux améliorations algorithmique accélère très sensiblement le code séquentiel en évitant des calculs inutiles... et rend la parallélisation plus difficile.

En tout état de cause, la fonction `evaluate` est récursive : si c'est mon tour, le score d'une position donnée est le max des scores des positions accessibles (car je joue le coup le plus avantageux pour moi). L'adversaire essaye lui de minimiser mon score, donc de maximiser son opposé. Le nombre d'appels récursifs est limité par $k$. Si $k = 0$, on fait appel à une fonction d'évaluation heuristique qui renvoie un score approximatif en temps constant. Tous les programmes d'échecs fonctionnent plus ou moins comme cela.

Chaque appel à `evaluate` explore un arbre, dont les noeuds représentent des positions du jeu, et dont les arêtes représentent les coups que les joueurs peuvent jouer. Le nombre de configurations examinées par `evaluate` est grosso-modo exponentiel en la profondeur maximale autorisée.

Les deux fonctions `decide` et `evaluate` prennent deux arguments : une position (lue) et un résultat (écrit).

3. `https://fr.wikipedia.org/wiki/Notation_Forsyth-Edwards`
4. `https://fr.wikipedia.org/wiki/Algorithme_minimax`
5. `https://fr.wikipedia.org/wiki/Elagage_alpha-beta`
6. `https://en.wikipedia.org/wiki/Transposition_table`, désolé, pas de page en français...
7. `https://en.wikipedia.org/wiki/Negamax`

# 2 Travail à effectuer

Le travail à effectuer se décompose en deux parties.

## 2.1 Partie 1 : parallélisation de la version naïve

Il s'agit de paralléliser l'algorithme `negamax` « de base », c'est-à-dire sans les deux `define`. Proposer une manière de paralléliser le calcul. Concevez une implémentation MPI pure, une implémentation OpenMP puis une implémentation MPI+OpenMP. Discutez des résultats obtenus. Testez votre code sur les positions du fichier `positions_v1.txt`.

## 2.2 Partie 2 : parallélisation de la version moins naïve

Modifiez `projet.h` en mettant l'option `ALPHA_BETA_PRUNING` à 1. Adaptez vos parallélisations (MPI, OpenMP, MPI+OpenMP), et testez le résultat sur les positions du fichier `positions_v2.txt`.

Si tout se passe bien (bonne efficacité, etc.), mettez l'option `TRANSPOSITION_TABLE` à 1, et recommencez avec les différentes parallélisations (MPI, OpenMP, MPI+OpenMP).

# 3 Travail à remettre

Pour chacune des deux parties, vous devrez remettre le code source, sous la forme d'une archive `tar` compressée et nommée suivant le modèle `projet_HPC_nom1_nom2.tar.gz`. L'archive ne doit contenir aucun exécutable, et les différentes versions demandées devront être localisées dans des répertoires différents. Chaque répertoire devra contenir un fichier `Makefile` : la commande `make` devra permettre de lancer la compilation, et la commande `make exec` devra lancer une exécution parallèle représentative avec des paramètres appropriés. Un fichier `Makefile` situé à la racine de votre projet devra permettre (avec la commande `make`) de lancer la compilation de chaque version.

A la fin du projet, vous devrez remettre un rapport au format `pdf` (de 5 à 10 pages, nommé suivant le modèle `rapport_HPC_nom1_nom2.pdf`) présentant vos algorithmes, vos choix d'implémentation (sans code source), vos résultats (notamment vos efficacités parallèles) et vos conclusions pour les deux parties. L'analyse du comportement de vos programmes sera particulièrement appréciée. Les machines n'étant pas strictement identiques d'une salle à l'autre, on précisera dans le rapport la salle utilisée pour les tests de performance.

# 4 Quelques précisions importantes

— Le projet est à réaliser par binôme.
— Vous **devez** lire le document intitulé « Projet HPC : conditions d'utilisation d'OpenMPI ». Vous veillerez notamment à n'utiliser qu'une salle à la fois pour vos tests, et vous n'oublierez pas de tuer **tous** vos processus sur **toutes** les machines utilisées à la fin de vos tests.

— Le code de la première partie («Parallélisation de la version naïve»), accompagné des slides de votre soutenance au format `pdf` (et donc sans animations) (prévoir une soutenance de 10 minutes, suivie de 5 minutes de questions), est à remettre au plus tard le dimanche 26 / 03 / 2017 à 23h59 (heure locale). Les soutenances de présentation de la première partie auront lieu lors de la séance de TDTP du mardi 28 / 03 / 2017.
Le code de la seconde partie («Parallélisation de la version moins naïve») et le rapport final sont à remettre au plus tard le dimanche 07 / 05 / 2017 à 23h59 (heure locale).
— Les remises se feront par courriel à : `charles.bouillaguet@univ-lille1.fr` et `pierre.fortin@lip6.fr`
— En cas d'imprévu ou de problème technique commun, n'hésitez pas à nous contacter pour que nous puissions vous proposer une solution ou une alternative.