

# Parallélisation d'un solveur d'échecs

## Comparaison d'implémentations

### HPC

Mathis CARISTAN & Alexandre FERNANDEZ

5 mai 2017

#### Résumé

Ce rapport présente la démarche suivie pour paralléliser un code de solveur d'échecs. Trois types d'implémentations ont été réalisées. Une purement MPI (avec la bibliothèque OpenMPI), une deuxième purement OpenMP, et une troisième utilisant un mélange des deux. Le travail a été découpé en deux blocs distincts, tout d'abord la parallélisation du code « naïf ». Puis l'extension de la parallélisation à une approche plus intelligente du problème qui utilise « l'élagage alpha-bêta ». Les trois implémentations ont été réalisées pour les deux blocs, et les résultats sont comparés ici.

## 1 Introduction

Nous considérons ici une version simplifiée du jeu d'échecs, dans laquelle on n'utilise que les pions et rois. Nous nous sommes vus fournir le code séquentiel du solveur. Celui-ci présentait deux paramètres d'optimisations, dont l'activation rendait le code plus dur à paralléliser. Le premier était l'utilisation de l'algorithme négamax avec un arbre alpha-bêta, et le second l'utilisation d'une table de transposition. Nous n'avons pas traité le second paramètre d'optimisation. Le programme prend en entrée un état du plateau en notation Forsyth-Edwards. Il cherche ensuite tous les coups possibles jusqu'à un nombre de coups donné. Il joue tous les coups, et analyse quel est le meilleur choix possible pour chaque joueur à chaque coup. Si il ne parvient pas à trouver une fin à la partie, il réitère en jouant un coup supplémentaire jusqu'à trouver une issue à la partie.

Les performances du programme séquentiel sont données dans le tableau 1. Ces valeurs serviront d'étalon pour mesurer les performances des parallélisations.

Optimisation	Machine(s)	Entrée	Temps
« Naïf »	gpu-3	???	???
« Naïf »	???	???	???
Alpha-bêta	gpu-3	???	???
Alpha-bêta	???	???	???

TABLE 1 – Temps d'exécution du programme séquentiel pour différents paramètres.

## 2 Parallélisation du code naïf

### 2.1 OMP

### 2.2 MPI

Pour cette implémentation, nous avons fait le choix d'utiliser une répartition de charge dynamique avec un modèle maître-esclave. Afin de ne pas subir un potentiel déséquilibre entre les différentes tâches, le maître prépare environ 10 fois plus de tâches qu'il n'y a de processus esclaves. Pour cela, il effectue un parcours en largeur jusqu'à arriver à une profondeur dont le nombre de branches vérifie le critère précédent. Nous nous appuyons sur une structure C, permettant de "remonter l'arbre" d'un noeud vers ses parents. La structure permet également d'accéder aux structures `tree_t` et `result_t` d'un noeud, pré-existante dans le code séquentiel. Ainsi, avec un tableau de cette structure que nous avons créée, le processus maître maintient une liste des noeuds du haut de l'arbre, qu'il utilise ensuite pour distribuer le travail aux processus esclaves.

Une fois que le maître a trouvé suffisamment de tâches, commence la répartition des tâches. Il envoie à chacun des processus esclaves, un couple de structures `tree_t/result_t` pour que celui-ci puisse appeler `evaluate`. Une fois sa tâche finie, un processus esclave renvoie son résultat au processus 0, qui a son tour lui renvoie une nouvelle tâche à effectuer. Lorsqu'il n'y a plus de tâches à une profondeur donnée, les esclaves se bloquent et se mettent en attente. Le maître recombine alors les données des esclaves. Ce déroulement est illustré par la figure 1. Lorsque le processus maître a identifié une situation correspondant à la fin de la partie, il indique aux esclaves qu'ils peuvent se terminer, avant de terminer lui-même.



FIGURE 1 – Illustration du fonctionnement du programme avec 4 processus. Le maître pré-calcul les tâches a distribuer, puis les communique aux esclaves. Les esclaves travaillent sur les tâches qui leurs sont attribuées avant de les renvoyer au maître. Enfin, le maître recombine les résultats ensemble.

### **2.3 OMP + MPI**

### **2.4 Résultats et analyse**

## **3 Parallélisation avec alpha-bêta**

### **3.1 OMP**

### **3.2 MPI**

### **3.3 OMP + MPI**

### **3.4 Résultats et analyse**