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 bilbiothèque 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 paralélisation du code « naïf ». Puis L'extension de la paralélisation 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 Fosryth-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 coups. 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 la table 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'éxecution du programme séquentiel pour différents paramètres.

2 Parallélisation du code naïf

2.1 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 suffisemment 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. On note que le processus maître ne travaille pas pendant que les esclaves calculent.

Ajouter les résultats (+ parler meta struct ? + avancée par rapport à la premiere deadline ?)

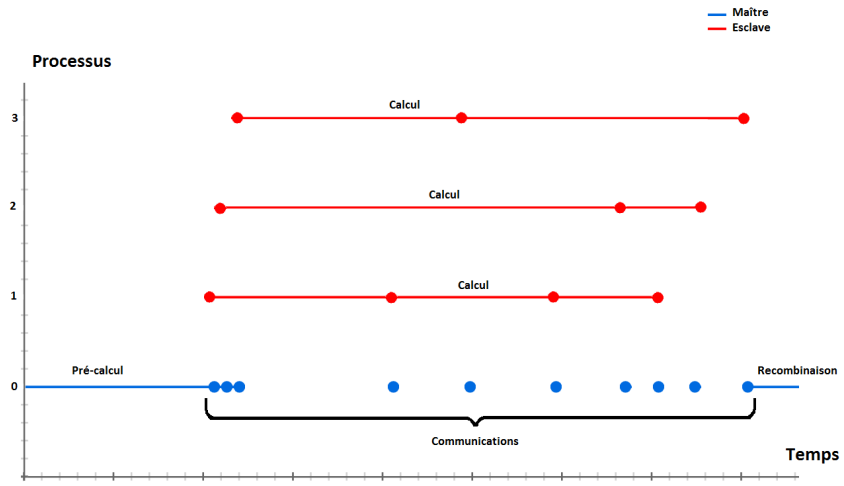

FIGURE 1 – Illustration du fonctionnement du programme avec 4 processus. Le maître pré-calcul les tâches a distribuer, puis les communique aux esclaves. Les esclaves travaillent sur les tâches qui leurs sont attribuées avant de les renvoyer au maître. Enfin, le maître recombine les résultats ensemble.

2.2 OMP

2.3 OMP + MPI

Cette implémentation reprend simplement les principes des deux premières. Un processus maître prépare des tâches qui sont ensuite distribuées aux esclaves. Ceux-ci utilisent alors OpenMP pour paralléliser le calcul de leur tâche, avant de renvoyer le résultat au maître.

Ajouter les résultats

2.4 Analyse et comparaison

3 Parallélisation avec alpha-bêta

3.1 MPI

L'approche alpha-bêta propose d'élaguer les branches les moins intéressantes au fur et à mesure de la progression de l'algorithme. De plus, une fonction nous permet de trier les coups possibles du plus au moins intéressant. Dès lors, une nouvelle approche semblait pertinente ; nous avons choisi de faire jouer au maître toute la branche la

???	???	???

TABLE 2 – résultats

plus à gauche de l'arbre, soit probablement les meilleurs coups (d'après une heuristique qui n'est pas exacte). Il élague ensuite les branches en fonction de ce premier parcours. Ensuite, nous appliquons à nouveau le principe utilisé pour la version naïve. Le maître génère un nombre de tâches suffisamment important puis les distribue aux esclaves. Une différence existe néanmoins avec la version précédente, lorsqu'il reçoit des résultats, il les analyse immédiatement, afin d'effectuer les coupes nécessaires si besoin. Dans la version naïve, l'analyse des résultats était effectuée collectivement une fois tous les résultats obtenus. Notons également l'introduction d'un paramètre supplémentaire pour la quantité de tâches générées. En effet, nous demandons que le maître génère $k * nb_{proc}$ tâches. Nous avons fait des essais avec plusieurs valeurs de k. Cette idée nous est venue en considérant que les coupes allaient probablement diminuer le nombre de tâches à effectivement analyser par les esclaves, augmentant la possibilité de déséquilibre de charge. Les résultats obtenus sont présentés dans la table 2

3.2 OMP

3.3 OMP + MPI

3.4 Résultats et analyse