Décision de finales d'échecs
Comparaison d'implémentations
HPC

Mathis Caristan & Alexandre Fernandez

7 mai 2017

Résumé

Ce rapport présente la démarche suivie pour paralléliser un code de solveur d'échecs. Trois types d'implémentation 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 parallélisation du code « naïf ». Puis l'extension de la parallélisation à une approche plus intelligente du problème, qui utilise « l'élagage alpha-bêta ». Les résultats des implémentations pour chaque bloc sont analysés et comparés ici.

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 présente deux paramètres d'optimisation, dont l'activation rend le code plus dur à paralléliser. Le premier paramètre est 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 la table 1. Ces valeurs serviront d'étalon pour mesurer les performances des parallélisations. Étant donné les temps d'exécution de ces instances, les temps d'exécution n'ont été mesurés qu'une seule fois, et en conséquence, sont sujets à une incertitude de mesure liée à la possible utilisation des machines par d'autre utilisateurs pendant la mesure. Néanmoins, on observe qu'ils semblent en accord avec les valeurs prévues dans les les

Optimisation	Machine(s)	Entrée	Temps	Nœuds cherchés
« Naïf »	14-15-401-05	"7K//k1P/7p b"	26min21	6 287 824 726
« Naïf »	gpu-3	"/ppp//PPP//7k//7K w"	1m58	258 991 723
Alpha-bêta	14-15-401-05	"/5p/4p/4P/4KP///k w"*	13min19	3 622 607 245

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

fichiers `positions_v1.txt` et `positions_v2.txt` pour les tests sur 14-15-401-05. Il est normal que les exécutions soient plus lentes sur gpu-3 puisque les processeurs de cette machine sont moint puissants individuellement que ceux des autres salles.

1 Parallélisation du code naïf

1.1 MPI

Pour cette implémentation, nous avons fait le choix d'utiliser une répartition de charge dynamique avec un modèle maître-esclaves. 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 esclave. Pour cela, il effectue un parcours en largeur jusqu'à arriver à une profondeur à laquelle 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-éxistantes 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 à 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.

Dans la table 2 sont présentés les résultats obtenus avec la parallélisation avec MPI. Ces résultats semblent corrects, compte-tenu du modèle de parallélisation utilisé

∗. Il est à noter que sur un solveur en ligne utilisant GNUChess 6.2.4, nous avons une solution à la profondeur 11 au lieu de 16. Lien vers le solveur : nextchessmove

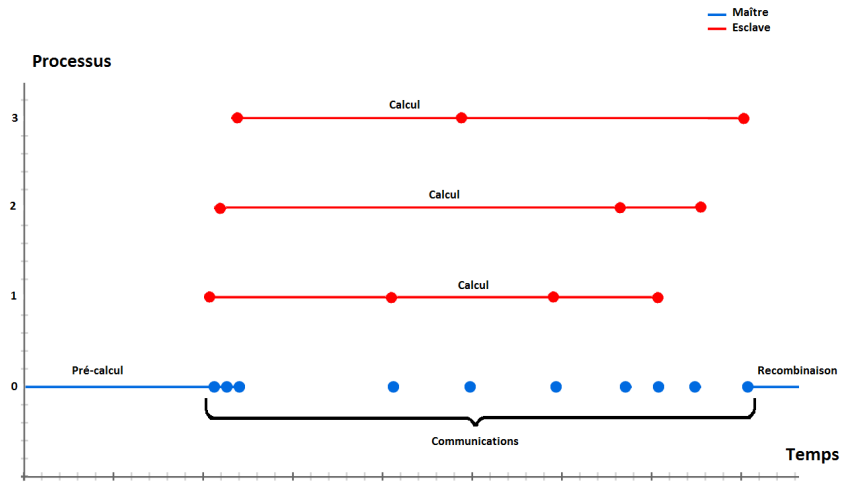

FIGURE 1 – Illustration du fonctionnement du programme avec 4 processus. Le maître pré-calcule les tâches a distribuer, puis les communique aux esclaves. Les esclaves travaillent sur les tâches qui leurs sont attribuées avant de les renvoyer au maître. Enfin, le maître recombine les résultats ensemble.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

Temps	**Accélération**	**Efficacité**
2min49±3.4s	9.34	77.82%

TABLE 2 – Résultats de la parallélisation du code « naïf ». Le temps d'exécution correspond à une moyenne sur 10 exécutions, afin de réduire les erreurs de mesure. Les tests ont été faits sur 12 machines de la salle 14-15-401.

(maître-esclave). En effet, bien que nous utilisons 12 processus, nous ne pouvons pas attendre une accélération meilleure que 11, puisque le maître ne participe quasiment pas (le pré-calcul ne prenant pas beaucoup de temps). On peut donc considérer que l'efficacité parallèle réelle est de $9.34/11 = 84,9\%$, et c'est vers cette valeur que devrait tendre l'efficacité si on augmentait le nombre de processus (en supposant que notre solution soit toujours efficace avec un nombre important de processus).

1.2 OMP

La première approche que nous avons considérée pour openMP consistait à utiliser un `# pragma omp parallel for` sur la boucle principale de `evaluate`. Bien que cette approche ait permis une réduction du temsp de calcul, son efficacité n'était pas bonne.
Nous avons donc cherché à utiliser les taùches de OMP afin d'améliorer l'éfficacité. Le

Profondeur maximum		Nombre de threads				
		2	4	8	16	32
2	Temps	1m41	1m25	1m10	1m9	1m8
	Accelération	1.2	1.4	1.7	1.7	1.7
	Efficacité	60%	35%	21%	10%	5%
4	Temps	1m46	1m13	1m13	1m10	1m8
	Accelération	1.1	1.6	1.6	1.7	1.7
	Efficacité	55%	40%	20%	10%	5%
8	Temps	1m10	1m20	1m13	1m15	1m30
	Accelération	1.7	1.5	1.6	1.6	1.3
	Efficacité	85%	37%	20%	10%	4%

TABLE 3 – résultats

code a paralléliser est le contenu de cette boucle `for`. Nous avons encapsulé l'appel a `play_move` et l'appel récursif à `evaluate` dans une tâche OMP. La partie du code executée quand un nœud a un meilleur score que son père avait initialement été mis dans cette tâche, et protégé par une zone critique. Cependant, nous avons finalement choisi de l'en sortir pour limiter l'utilisation de sémaphore. Á la place, les scores des enfants sont stockés dans un tableau, pour lequel on cherche ensuite le meilleur score. Cette technique permet d'éviter de multiples comparaisons avec le score du parent.

Un point important, et qui fut pour nous source d'erreur, est que le premier appel à `evaluate` doit être fait dans une zone parallèle OMP (afin que plusieurs threads soient créés), mais doit également être protégé par `# pragma omp single` afin d'éviter de lancer le calcul plusieurs fois.

Nottons de plus que nous devons limiter le nombre de tâches OMP créées, si on ne veut pas voir les performances réduites. Ce problème a été résolu en mettant une condition sur la création des tâches OMP, liée à la profondeur de récursion (et donc au nombre de tâches déjà créées). Les résultats liés à cette méthode sont très variables en fonction de la profondeur maximum autorisée pour la création de tâches supplémentaires, et du nombre de threads OMP.

1.3 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

1.4 Analyse et comparaison

On note que

2 Parallélisation avec alpha-bêta

2.1 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 autre approche que celle utilisée dans la prmière partie semble pertinente. Nous avons choisi de faire jouer au maître toute la branche la plus à gauche de l'arbre, soit probablement les meilleurs coups (d'après une heuristique qui n'est pas exacte). L'algorithme part ensuite de la feuille qu'il a calculé, et remonte l'arbre. A chaque niveau de l'arbre, il vérifie si le nœud de gauche permet d'élaguer les autres nœud avec lesquels il partage son parent. Dans le cas contraire, le calcul des « nœds frères »est lancé en parallèle avec MPI.

Malheureusement, cette méthode est peu efficace, car elle préesente deux principaux désaventages :

1. Tout d'abord, dans nos parties d'échecs, le nombre de coups est souvent limité (autour de 5-6). Or chaque coup est joué sur un processus MPI, donc un grand nombre de processus ne bénéficie pas du tout à cet algorithme.
2. De plus, pour une profondeur importante (>14), en arrivant sur le haut de l'arbre, si les branches de droite n'ont pas été élaguées, leur calcul se révèle quasiment aussi long que le calcul séquentiel initial.

Plusieurs solutions ont été envisagées pour résoudre ces problèmes. D'une part il a été envisagé d'utiliser, comme pour la parallélisation naïve, une liste de tâches à faire, et de ne lancer la parallélisation qu'après que cette liste ait atteint une taille suffisamment importante (de l'ordre de 10 fois le nombre de processus). La mise en place de cette solution n'a pas été concluante. En effet, Lorsque des nœuds de différents niveaux sont parallélisés ensemble le calcul n'est plus correct. Nous avons par exemple obtenus une solution pour l'entrée "/5p/4p/4P/4KP///k w" en moins d'une seconde, et qui n'était pas la bonne (profondeur de 10 au lieu de 16, mauvais choix de mouvement pour le joueur noir dès son premier tour de jeu, moins de 50 000 nœuds visités). Nous n'avons pas réussi à isoler la cause de ce dysfonctionnement et avons donc simplement choisi de ne pas utiliser cette méthode.
La solution pour le second problème, était d'appeler récursivement la fonction `alpha` (la fonction du processus maître) lorsque l'algorithme était haut dans l'arbre. Nous avons également tenté d'implémenter cette solution, sans succès. Une erreur de segmentation survenait de manière irrégulière (ce qui rendait sa correction difficile) lors

???	???	???

TABLE 4 – résultats

de l'exécution.

Pendant nos tests et tentatives de résolution, afin d'augmenter la chance de détecter des problèmes d'exécution non systématique (à cause de la parallélisation), nous avons réduit le nombre de processus MPI à deux, ce qui permet d'avoir toujours le même ordre d'exécution (dans le case maître-esclave). Les résultats obtenus sont présentés dans la table 4

2.2 OMP

2.3 OMP + MPI

2.4 Résultats et analyse