Décision de finale d'échecs

Encadrant : **C. Bouillaguet,**

Etudiants : **N. Guittonneau, H. Lacour**

Table des matières

1 Introduction

1.1 Travail réalisé

Nous avions réalisé dans un premier temps une solution naïve qui consistait à paralleliser seulement le premier appel à `evaluate`, l'efficacité étant évidemment désastreuse en augmentant les ressources. L'arbre étant souvent déséquilibré, seulement un processus se retrouvait au final à faire le plus gros des calculs.

La solution maître esclave s'est donc imposée, sa programmation étant facilement réalisable, garantissant en théorie des résultats satisfaisants et offrant la possibilité de changer la granularité des données.

Une tentative de programmation de vol de travail à été effectuée, cependant ses résultats étant décevants, seul son algorithme sera brièvement décrit dans ce rapport.

1.2 Information techniques

Les scripts de tests on été écrits en `sh` et toutes les mesures de temps on été effectuées à l'aide du programme `time`. Les temps mesurés sont donc légérement supérieurs au temps réels d'execution des programmes, car il faut prendre en compte le temps d'initialisation de MPI et autre opérations auxiliaires. Nous avons cependant jugés ces variations négligeables.

Chaque sous-dossier possède son propre Makefile. Le projet possède un Makefile à la racine permettant de compiler, nettoyer ou tester tous les sous-dossiers.

Toutes les mesures ont été effectuées dans la salle ppti-14-403/ppti-14-302, sur 4 machines équipées de processeurs `Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz` Voici les résultats de la version séquentielle du programme :

Table 1 : Resultats séquentiels

Position	Temps (s)
4k//4K/4P w	15.33
4k//4K//4P w	26.60
/ppp//PPP//7k//7K w	90.07

2 Partie 1 : Parallélisation de la version naïve

2.1 Approche MPI : Maître esclave

Pour la version naïve de projet nous avons opté pour une approche maître esclave. Le processus maître (de rang MPI 0), crée un ensemble de tâches qui seront ensuite distribuées

à chaque esclave. Lorsque l'un d'eux a finit son travail, il demande une nouvelle tâche au maître, qui répondra soit par une nouvelle charge de travail, soit par une notification de fin de calcul. Lorsque le maître parvient à prendre la décision finale il notifie tous les esclaves.

Une tâche est représentée par une `struct tree_t` et le résultat via une `struct result_t`. Ces données sont sockées par le maître dans des tableaux grandissant dynamiquement. Le temps de traitement d'une tâche pouvant sévèrement varier, la réception des résultats n'a aucune garantie d'être dans le même ordre qu'à l'envoi. Pour savoir quel esclave à effectué quelle tâche, l'indice du tableau associé à la dite tâche est inclus dans les échanges.

Les primitives appelées par le maîtres sont les suivantes :

- `evaluate_master()` : permet de créer les tâches. cette fonction est similaire à celle présente dans le code sequentiel, à la différence que lorsque qu'une certaine hauteur est atteinte (constante `MAXHEIGHT`), le `tree_t` est sauvegardé et l'appel récursif n'est pas effectué.

- `distribute_work()` : envoie préemptivement une tâche à chaque esclave et attend de recevoir les résultats. A la réception, envoie une nouvelle tâche si il en reste à distribuer, sinon envoie `TAG_END_DEPTH` à l'esclave.

- `send_work()` : fonction annexe de `distribute_work`, prend en paramètre un rang mpi, une tâche, son indice et l'envoie proprement à l'esclave.

- `evaluate_continue()` : cette fonction est appelée lorsque tous les esclaves ont communiqué leurs résultats. Arrivée à la hauteur où les tâches avaient été crées, elle substitue l'appel récursif à `evaluate` par une recopie de la solution renvoyée par l'esclave dans le `child_result`, de cette manière, la fonction va pouvoir calculer le résultat pour la profondeur actuelle.

- `decide()` : appelée uniquement par le maître, elle utilise respectivement les primitives décrites ci-dessus pour chaque profondeur jusqu'a ce qu'une décision soit prise. La solution est ensuite affichée et les esclaves sont notifiés par le tag `TAG_END`. La somme du nombre de noeuds parcourus par chaque esclave est calculée, affichée, et le programme se termine.

En revanche, un esclave est définit par une unique primitive :

- `slave_work()` : une simple boucle infinie consistant à recevoir un message de la part du maître puis en le traitant en fonction du tag :

  - `TAG_WORK` : il signifie la réception d' un `tree_t`. L'esclave peut ensuite lancer la primitive `evaluate()` et renvoyer le `result_t` ainsi formé au maître avant de repasser en attente d'un message.

– `TAG_END_DEPTH` : continuer l'écoute de messages ; ce tag permet surtout au maître de pouvoir finaliser le calcul.
– `TAG_END` : une décision à été prise, le nombre de noeuds parcourus est envoyé au maître et le processus peut se terminer.

Il est possible de faire varier la granularité de cette approche en modifiant la hauteur à laquelle le maître crée les tâches. Plus celle ci est élévée et plus les tâches à distribuer seront nombreuses, mais la parallélisation commencera à une profondeur plus tardive.

2.2 Résultats MPI

cf. Table 2

2.3 Algorithme OpenMP

Les tâches OpenMP on été crées pour paralléliser des algorithmes non équilibrés dont notamment les algorithmes récursifs. Cela permet exactement de répondre à notre problème, apportant une solution similaire à celle donnée avec MPI, laissant l'API openMP abstraire la gestion des tâche pour le développeur.

Des tâches OpenMP sont crées jusqu'a ce qu'une hauteur définie soit dépassée. De même que pour MPI, il est possbile de jouer avec la granularité via cette variable.

2.4 Résultats OpenMP

cf. Table 3

2.5 Algorithme OpenMP + MPI

L'approche openMP et MPI est une combinaison des deux solutions apportées dans les paragraphes précedents, une approche maître/esclaves où les tâches sont parallélisées avec openMP. Le processus mpi de rang 0 fait toujours office de maître et crée les tâches, en revanche l'appel à `evaluate` des esclaves est optimisé via openMP.

2.6 Résultats OpenMP + MPI

cf. Table 4

3 Partie 2 : Parallélisation de la version moins naïve

3.1 Réalisation d'un vol de travail

Afin de répondre à la deuxième partie du projet, nous avons essayé de développer un algorithme de vol de travail pour de garder une bonne efficacité malgré de gros déséquilibres

dans l'arbre des appels résursifs. En théorie, le top level d'`evaluate` est partagé entre tous les processus, ensuite, dès que l'un deux a fini, il tente de déléguer une partie des appels récurifs à un autre processus ayant fini son propre travail. La topologie est en anneau afin de détecter la terminaison plus facilement. Pour cela nous utilisons une solution librement inspirée de l'algorithme de Rana :

- Lorsqu'un processus termine, il envoie un message spécifique `TAG_WORK_DONE` à son successeur avec la valeur `"1"`.
- À la réception d'un de ces messages :
  - Le site est passif et le message est retransmis à son successeur avec la valeur n+1.
  - Le site est inactif et le message n'est pas retransmis.
- Lorsque qu'un site passif recoit la valeur `NB_SITES`, c'est que son message à fait le tour de l'anneau et que tous les sites sont inactifs, la terminaison est donc détécté et est retransmise à tous les sites.

. Il faut cependant faire attention car plusieur détéction peuvent être faites au même moment et donc des messages érronés peuvent circuler. Le deux primitives majeures sont :

`share_work()` appelée dans `evaluate` avant les appels récursifs, permet de vérifier si un message de type `TAG_WORK_DONE` à été reçu de la part du site précedent lui envoie la moitié des appels récursif qu'il devait faire. Le message doit toutefois avoir été initialement envoyé par le prédécesseur (valeur = 1) et ne doit pas provenir de la profondeur précedente (message périmé). La distribution lève un flag qui permettra d'attendre le résultat du site lors de la sortie des appels récursifs.

`finishTheJob()`, appelée lorsque qu'un site à fini son travail initial, prévient son successeur de son changement d'état et s'attend à recevoir soit une notification de fin de travail venant du site le précedant, soit une charge de travail du site le succedant, ou, une notification de terminaison de la part de n'importe quel site.

Cet algorithme n'est pas complètement fonctionnel, le bon résulat est calculé mais certains processus ne parviennent pas à voler du travail à leur successeur. Cet algorithme n'étant pas trivial à mettre en place et à déboguer, et ne garantissant pas forcément de bons résultats, son developpement à été abandonné.

Table 2 : Resultats MPI

Position	Proc num	MAXHEIGHT	Time (s)	Efficiency
4k//4K/4P w	2	2	15.46	0.49
4k//4K//4P w	2	2	26.49	0.50
/ppp//PPP//7k//7K w	2	2	89.74	0.50
4k//4K/4P w	4	2	5.70	0.67
4k//4K//4P w	4	2	9.99	0.67
/ppp//PPP//7k//7K w	4	2	34.44	0.65
4k//4K/4P w	8	2	3.01	0.63
4k//4K//4P w	8	2	5.34	0.62
/ppp//PPP//7k//7K w	8	2	23.49	0.48
4k//4K/4P w	2	3	15.44	0.50
4k//4K//4P w	2	3	26.79	0.50
/ppp//PPP//7k//7K w	2	3	89.59	0.50
4k//4K/4P w	4	3	5.53	0.69
4k//4K//4P w	4	3	9.51	0.70
/ppp//PPP//7k//7K w	4	3	32.27	0.70
4k//4K/4P w	8	3	3.03	0.63
4k//4K//4P w	8	3	4.66	0.71
/ppp//PPP//7k//7K w	8	3	16.32	0.69
4k//4K/4P w	2	4	15.27	0.50
4k//4K//4P w	2	4	26.45	0.50
/ppp//PPP//7k//7K w	2	4	88.33	0.51
4k//4K/4P w	4	4	6.24	0.61
4k//4K//4P w	4	4	9.75	0.68
/ppp//PPP//7k//7K w	4	4	34.05	0.66
4k//4K/4P w	8	4	2.76	0.69
4k//4K//4P w	8	4	4.38	0.76
/ppp//PPP//7k//7K w	8	4	13.70	0.82

Table 3 : Resultats OpenMP

Position	Thread num	MAXHEIGHT	Time (s)	Efficiency
4k//4K/4P w	2	2	8.60	0.89
4k//4K//4P w	2	2	17.83	0.75
/ppp//PPP//7k//7K w	2	2	53.03	0.85
4k//4K/4P w	4	2	6.41	0.60
4k//4K//4P w	4	2	11.68	0.57
/ppp//PPP//7k//7K w	4	2	41.15	0.55
4k//4K/4P w	8	2	9.36	0.20
4k//4K//4P w	8	2	9.28	0.36
/ppp//PPP//7k//7K w	8	2	41.17	0.27
4k//4K/4P w	2	3	8.61	0.89
4k//4K//4P w	2	3	17.83	0.75
/ppp//PPP//7k//7K w	2	3	52.83	0.85
4k//4K/4P w	4	3	6.40	0.60
4k//4K//4P w	4	3	11.62	0.57
/ppp//PPP//7k//7K w	4	3	41.32	0.54
4k//4K/4P w	8	3	7.30	0.26
4k//4K//4P w	8	3	9.73	0.34
/ppp//PPP//7k//7K w	8	3	41.13	0.27
4k//4K/4P w	2	4	8.61	0.89
4k//4K//4P w	2	4	17.82	0.75
/ppp//PPP//7k//7K w	2	4	53.06	0.85
4k//4K/4P w	4	4	6.37	0.60
4k//4K//4P w	4	4	11.66	0.57
/ppp//PPP//7k//7K w	4	4	41.18	0.55
4k//4K/4P w	8	4	6.63	0.29
4k//4K//4P w	8	4	9.74	0.34
/ppp//PPP//7k//7K w	8	4	40.83	0.28

Table 4 : Resultats MPI+OpenMP

Position	Proc num	Thread num	MAXHEIGHT	Time (s)	Efficiency
4k//4K/4P w	2	2	2	9.20	0.41
4k//4K//4P w	2	2	2	15.80	0.42
/ppp//PPP//7k//7K w	2	2	2	48.47	0.46
4k//4K/4P w	2	4	2	6.89	0.27
4k//4K//4P w	2	4	2	11.94	0.27
/ppp//PPP//7k//7K w	2	4	2	37.46	0.30
4k//4K/4P w	4	2	2	5.07	0.37
4k//4K//4P w	4	2	2	8.28	0.40
/ppp//PPP//7k//7K w	4	2	2	25.44	0.44
4k//4K/4P w	4	4	2	4.38	0.21
4k//4K//4P w	4	4	2	7.61	0.21
/ppp//PPP//7k//7K w	4	4	2	23.38	0.22
4k//4K/4P w	8	2	2	2.62	0.36
4k//4K//4P w	8	2	2	4.38	0.37
/ppp//PPP//7k//7K w	8	2	2	13.89	0.40
4k//4K/4P w	8	4	2	2.48	0.19
4k//4K//4P w	8	4	2	4.02	0.20
/ppp//PPP//7k//7K w	8	4	2	12.78	0.22
4k//4K/4P w	2	2	3	9.54	0.40
4k//4K//4P w	2	2	3	18.00	0.36
/ppp//PPP//7k//7K w	2	2	3	48.11	0.46
4k//4K/4P w	2	4	3	7.25	0.26
4k//4K//4P w	2	4	3	11.90	0.27
/ppp//PPP//7k//7K w	2	4	3	35.68	0.31
4k//4K/4P w	4	2	3	5.27	0.36
4k//4K//4P w	4	2	3	8.40	0.39
/ppp//PPP//7k//7K w	4	2	3	25.02	0.44
4k//4K/4P w	4	4	3	4.89	0.19
4k//4K//4P w	4	4	3	7.94	0.20
/ppp//PPP//7k//7K w	4	4	3	23.63	0.23
4k//4K/4P w	8	2	3	2.40	0.39
4k//4K//4P w	8	2	3	3.99	0.41
/ppp//PPP//7k//7K w	8	2	3	11.62	0.48
4k//4K/4P w	8	4	3	2.43	0.19
4k//4K//4P w	8	4	3	3.83	0.21
/ppp//PPP//7k//7K w	8	4	3	11.32	0.24
4k//4K/4P w	2	2	4	15.77	0.24
4k//4K//4P w	2	2	4	26.83	0.24
/ppp//PPP//7k//7K w	2	2	4	87.17	0.25
4k//4K/4P w	2	4	4	16.41	0.11
4k//4K//4P w	2	4	4	27.01	0.12

Table 5 : Resultats MPI+OpenMP (suite)

Position	Proc num	Thread num	MAXHEIGHT	Time (s)	Efficiency
/ppp//PPP//7k//7K w	2	4	4	146.39	0.07
4k//4K/4P w	4	2	4	6.71	0.28
4k//4K//4P w	4	2	4	12.38	0.27
/ppp//PPP//7k//7K w	4	2	4	37.66	0.29
4k//4K/4P w	4	4	4	7.71	0.12
4k//4K//4P w	4	4	4	12.72	0.13
/ppp//PPP//7k//7K w	4	4	4	35.79	0.15
4k//4K/4P w	8	2	4	3.19	0.30
4k//4K//4P w	8	2	4	5.16	0.32
/ppp//PPP//7k//7K w	8	2	4	15.82	0.35
4k//4K/4P w	8	4	4	3.57	0.13
4k//4K//4P w	8	4	4	5.52	0.15
/ppp//PPP//7k//7K w	8	4	4	15.79	0.17