Comparatif des performances d'une application en fonction des paramètres de compilation utilisés et analyse des performances

Pierre TASSEL sous la direction de Sid TOUATI

(Date : 17 juin 2018)

L'utilisation d'outils tels que *GProf* et la suite *Parallel Studio XE d'Intel*© nous permettent une analyse plus poussée des programmes et ouvrent la voie vers de possibles optimisations.

Dans le but de nous familiariser avec les outils d'analyse de performances d'Intel et du projet GNU, nous allons analyser un algorithme de type MinMax avec coupe Alpha Bêta1 sur l'une des variantes du jeu Awale2 avec six cases par joueur écrit en C++. Le but étant de déterminer la meilleure façon de compiler ce programme (compilateur et options à utiliser), d'analyser ce programme pour déterminer ce qui ralentit son exécution ainsi que de tenter de l'optimiser par différents moyens.

CONTENTS

I. INTRODUCTION

A. Programme à analyser

Le code à analyser étant dépendant de Windows (import de la bibliothèque windows.h), il faut tout d'abord le réécrire pour permettre de le rendre non dépendant d'un os en particulier. Cette bibliothèque est utilisée pour les compteurs de temps, il est donc facile de réécrire le programme afin qu'il soit indépendant de l'OS en utilisant les fonctions de la librairie standard.

Ce programme utilise un algorithme de type MinMax avec coupes alpha bêta1 pour le jeu d'Awalé2, le principe d'un algorithme MinMax est de parcourir jusqu'à une certaine profondeur l'arbre des coups possibles à un état donné du jeux et d'attribuer un score à chaque coup possible, il choisira ensuite le coup qui lui est le plus favorable en partant du principe que l'adversaire choisira les meilleurs coups. Les coupes alpha bêta sont une technique permettant d'aller plus profondément dans l'arbre des coups possibles en éliminant certains mauvais coups et évitant ainsi beaucoup d'explorations inutiles.

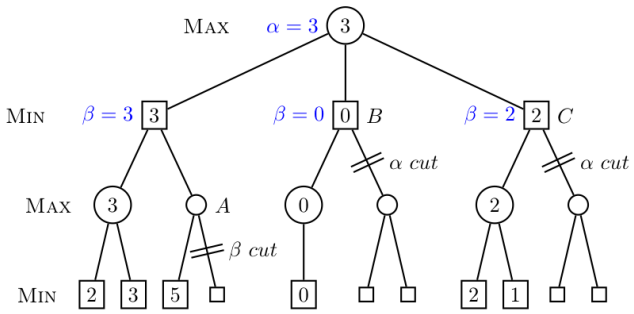

FIGURE 1: Algorithme MinMax Alpha-Beta.
Source: Github Lewis Mato.

FIGURE 2: Le jeux de l'Awale.
Source: Wikipedia.

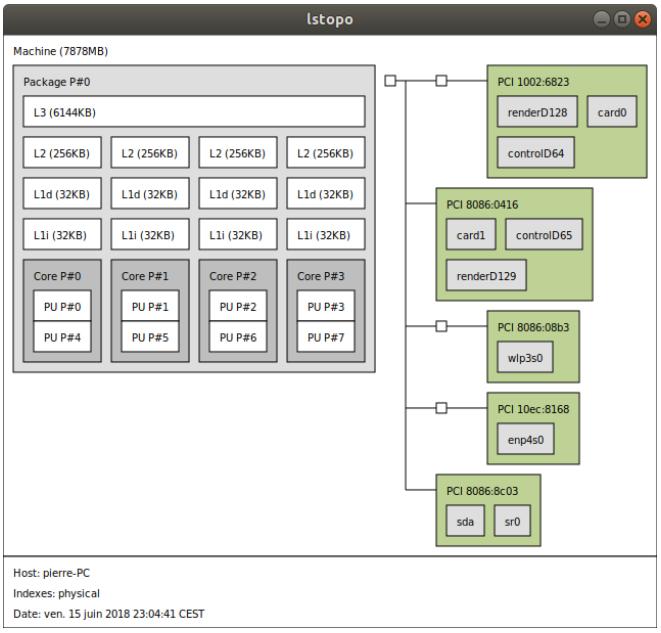

FIGURE 3: Topologie des caches de ma machine.

B. Environnement d'analyse

1. *Logiciel*

Afin d'analyser le programme efficacement, il faut aussi mettre en place un environnement propice à l'analyse des performances, pour cela nous utiliserons un système d'exploitation en CLI, avec le minimum de services possible installés. Nous utiliserons donc Arch Linux, basé sur le noyau Linux 4.14.13-1 cette distribution possède le minimum d'applications et de services possibles installés.

Nous utiliserons trois compilateurs, GCC, ICC (Intel C++ Compiler) et CLang. La version de GCC installée est la version 5.4.0, la version de CLang est la version 6.0.0-1 et celle de ICC est la version 18.0.2.

2. *Matériel*

Notre environnement expérimental est composé d'un ordinateur portable utilisant un processeur Intel i7-4710HQ, 8Go de RAM en DDR3, un SSD de 120go où le processeurs scaling a été désactivé dans le BIOS, cela permet de faire tourner le processeur à vitesse constante et ainsi de ne pas fausser les résultats.

II. ANALYSE DES OPTIONS DE COMPILATIONS OPTIMAUX

A. Démarche expérimentale

Il faut tout d'abord modifier le programme pour qu'il n'affiche en sortie uniquement le temps d'exécution de la partie et non plus les informations utiles aux joueurs.

Ensuite, il faut obtenir un ensemble de données. Sachant que le programme est déterministe (pas de modification de la profondeur en fonction du temps, ni de random) les mêmes entrées (coups joués par nous) produiront les mêmes sorties (coups joués par le logiciel). Il nous suffit donc de posséder une liste de coup à envoyer sur la sortie standard des différentes versions du programme (obtenue à partir des différentes options de compilations utilisées) et nous obtiendrons le temps d'exécution de chacun.

Afin d'obtenir les entrées du programme, nous avons fait tourner deux instances du programme, l'un jouant le joueur 1, le second jouant le joueur 2 et nous avons noté les coups joués par chacun.

A la fin de l'obtention de ces différents coups, nous pouvons déduire que le temps d'exécution sera à analyser en seconde, vu que la partie est plutôt longue.

Nous créerons ensuite un script exécutant les différentes compilations et exécuteras les programmes vingt fois chacun pour chaque partie (une fois où le programme commence, une fois où le joueur commence). Cela permettra de déterminer quel est le meilleur compilateur et avec quels options.

Ce script étant très lent à s'exécuter (il y a 480 exécutions différentes au total) nous l'exécuterons au travers

de la commande **nohup** afin d'éviter toute interruption possible.

B. Résultats

En regardant les temps d'exécutions configuration par configuration nous observons que les temps d'exécutions sont plutôt stables. Nous choisissons donc d'afficher les résultats sous forme de barres représentant la médiane des temps d'exécutions de chaque configuration.

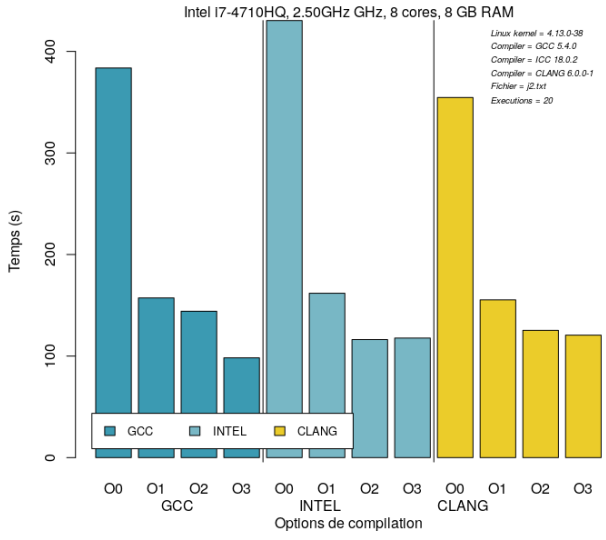

FIGURE 4: Temps d'exécutions quand le programme commence.

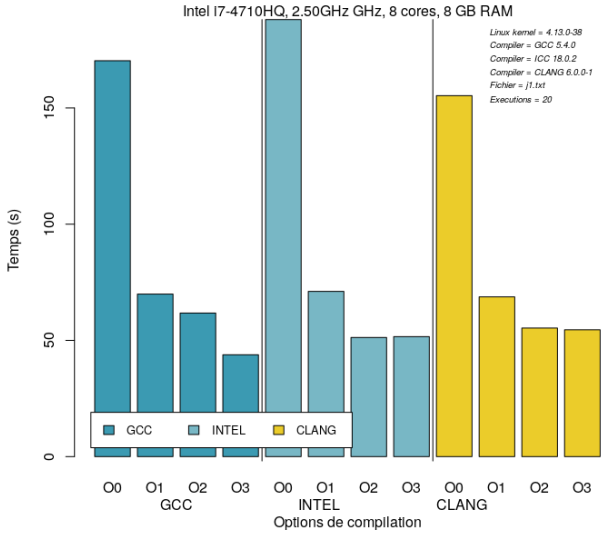

FIGURE 5: Temps d'exécutions quand le joueur commence.

On remarque que le fait que le joueur commence3 ou non3 n'a pas d'influence sur le choix du compilateur, on pourra donc se concentrer sur l'analyse des temps d'exécutions quand le programme commence.

On remarque aussi que GCC avec l'option **O3** activée est le meilleur compilateur pour ce programme.

Pour GCC et Intel, chaque niveau d'optimisation réduit grandement le temps d'exécution, il y a un rapport de quatre entre la compilation sans optimisation (**O0**) et la compilation avec le plus d'optimisations (**O3**). Cette différence est légèrement inférieure avec CLang, car la version sans optimisation (**O0**) est plus rapide que celle des autres compilateurs, mais elle reste d'un facteur trois.

Une chose étonnante est que la version **O2** d'Intel est légèrement plus rapide que la version **O3**.

1. *Taille des exécutables*

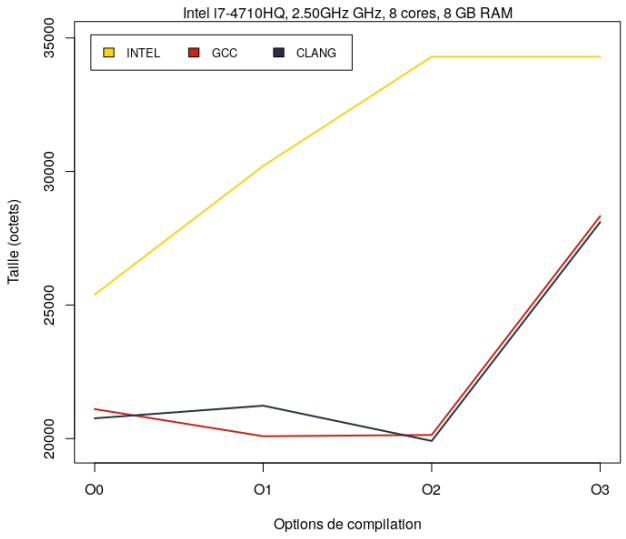

FIGURE 6: Comparatif des tailles d'exécutables produits par diverses méthodes de compilations.

GCC avec l'option **O3** semble être, là aussi, la meilleure option pour ce programme.

III. PROFILAGE DU CODE

Il existe deux grands logiciels pour analyser les performances d'un programme et les fonctions chaudes, GProf qui est un logiciel libre et vTune qui est un logiciel propriétaire appartenant à Intel. Nous allons utiliser ces deux logiciels, afin de trouver des possibilités d'optimisation de code.

A. Gprof

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 48.83    167.72   167.72 5172852992     0.00     0.00  jouer_coup(Next*, Pos*, Pos*, int, int)
 18.25    230.39    62.67 3262157329     0.00     0.00  copier(Pos*, Pos*)
 17.22    289.56    59.16 3262157262     0.00     0.00  est_affame(Pos*, int)
  8.79    319.75    30.20 1457826958     0.00     0.00  calculer_coup(Next*, Pos*, int, int, int, int, bool)
  5.31    337.98    18.23 3252245939     0.00     0.00  valeur_minimaxAB(Next*, Pos*, int, int, int, int, bool)
  0.87    340.99     3.00 1775297034     0.00     0.00  evaluer(Pos*)
  0.52    342.77     1.79       67     0.03     0.03  test_fin(Pos*)
  0.16    343.34     0.57                             print_position_ordi_haut(Pos*)
  0.09    343.65     0.31       33     0.01    10.34  decisionAB(Next*, Pos*, int, bool)
  0.01    343.70     0.05                             decision(Next*, Pos*, int)
  0.00    343.71     0.01        1     0.01     0.01  _GLOBAL__sub_I__Z13init_positionP3Pos
  0.00    343.71     0.00       67     0.00     0.00  print_position_ordi_bas_inv(Pos*)
  0.00    343.71     0.00        1     0.00     0.00  position_debut(Pos*)
  0.00    343.71     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
```

FIGURE 7: Fonctions chaudes d'après GProf.

Il nous apprend que la fonction la plus utilisée est ***jouer_coup***, et ce très largement, cependant on remarque qu'il y a énormément d'appel (plus de cinq milliards) mais que chaque appel est extrêmement rapide. On note aussi que la fonction **decisionAB** est moins souvent appelée, mais chaque appel est d'une plus longue durée (dix secondes par appel en moyenne).

Ce sont donc les deux fonctions qu'il faut tenter d'optimiser en priorité pour améliorer les performances d'après GProf.

B. Intel vTune

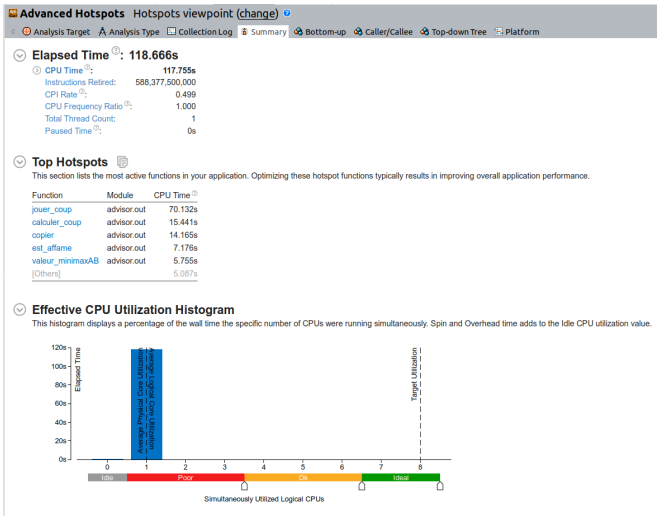

FIGURE 8: Profilage du code avec vTune.

Comme GProf, vTune nous indique que c'est la fonction **jouer_coup** qui a pris le plus de temps à calculer. Cependant, il place **calculer_coup** second contrairement a GProf.

Il nous apprend aussi que seul un processeur est utilisé sur les huits disponibles. Il y a donc un manque de parallélisme, ce qui ouvre une première voie vers une possible optimisation.

Nous pouvons aussi utiliser ce logiciel pour analyser les accès mémoire.

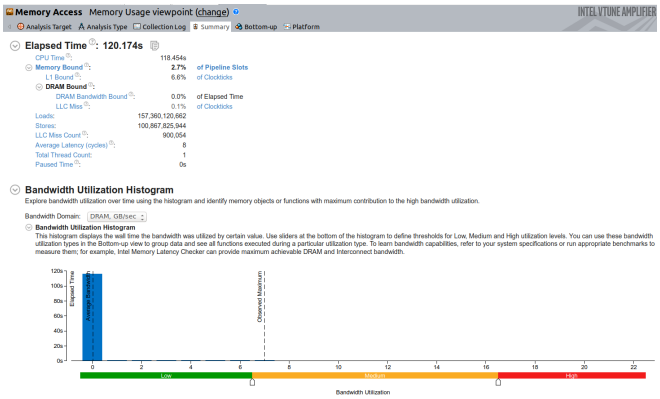

FIGURE 9: Profilage des accès mémoire.

On remarque très clairement que le programme n'est pas "Memory Bottleneck" mais "CPU Bottleneck" (le CPU est le facteur limitant de cette application).

Car il y a très peu d'erreurs de cache et les accès mémoire sont petits et rapides.

IV. AJOUT DE PARALLÉLISME

Comme ce programme est CPU Bottleneck et qu'il n'y a qu'un seul processeur utilisé, l'ajout de parallélisme semble être une possibilité d'optimisation.

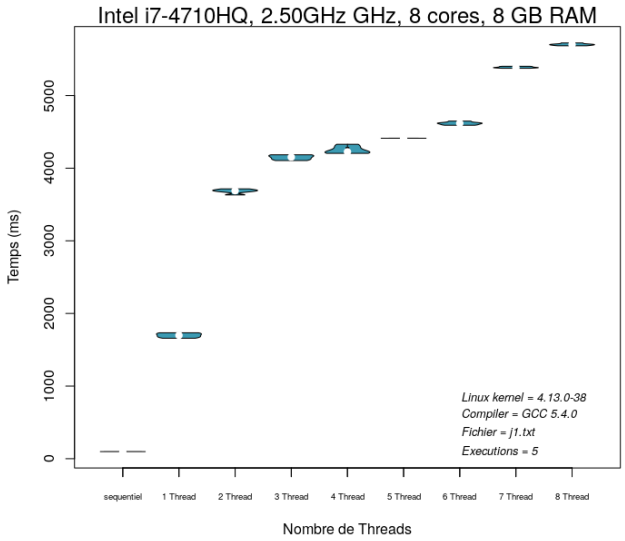

FIGURE 10: Résultat ajout parallélisme de façon naïve avec GCC.

A. Approche naïve

1. *Démarche expérimentale*

On a ajouté du parallélisme un peu partout où il n'y a pas de dépendance de données en utilisant la librairie OpenMP.

Malheureusement, il a été impossible de rajouter du parallélisme de contrôle, mais seulement du parallélisme de données, on peut d'ores et déjà s'attendre à ce que le parallélisme soit peu pertinent vu la faible taille des données et la simplicité des actions effectuées dans chacune des itérations des boucles parallélisées.

Ensuite, nous réitérons ce que nous avons fait pour la version sans parallélisme en utilisant chaque compilateur avec l'option **O3** activé mais, en faisant varier le nombre de processeurs pour déterminer le nombre optimal de processeur à utiliser.

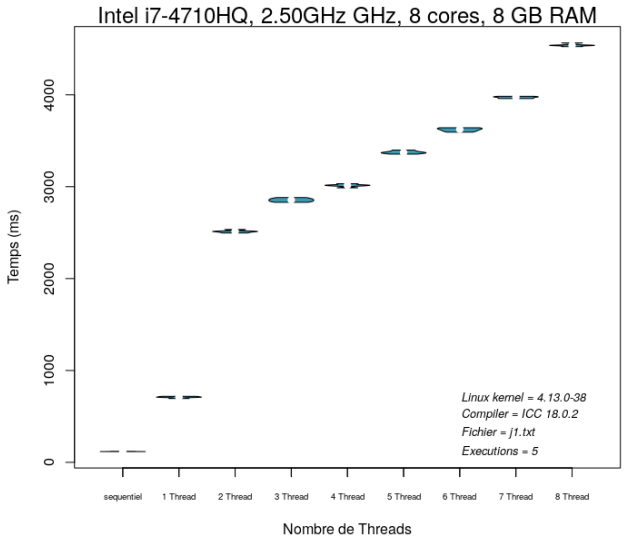

FIGURE 11: Résultat ajout parallélisme de façon naïve avec Intel.

2. *Résultats*

Nous représentons les résultats sous forme de Violin Plot nous permettants de représenter à la fois les écarts types et la fonction de densité.

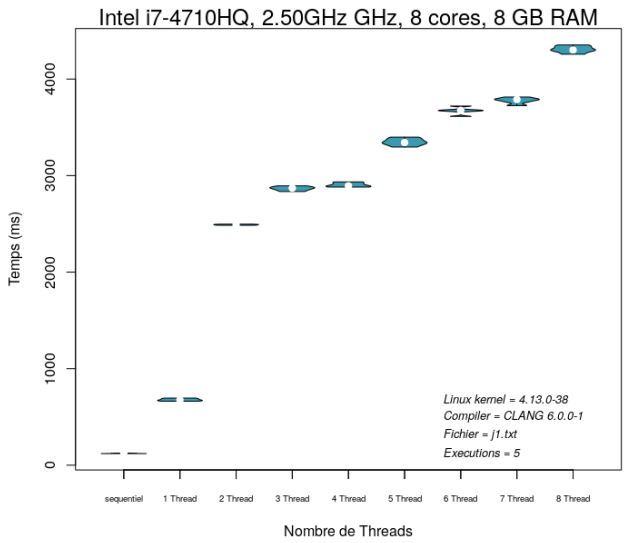

FIGURE 12: Résultat ajout parallélisme de façon naïve avec CLang.

Nous devons malheureusement réduire le nombre d'exécutions par configuration passant de vingt à cinq, car le temps d'exécutions est extrêmement lent.

On remarque ici que CLang6 et Intel5 sont côte à côte niveau performance et qu'ici GCC5 pourtant le plus rapide sur le code séquentiel est le plus lent avec la version parallèle.

On remarque aussi que le temps d'exécutions c'est nettement dégradé quel que soit le compilateur utilisé et qu'il se détériore de plus en plus quand on ajoute un plus grand nombre de processeurs. Cela est probablement dû à deux facteurs :

1. Le phénomène de False Sharing6, les différents processeurs doivent assurer la cohérence des caches, à chaque fois qu'un processeur modifie une donnée, il invalide le cache d'un autre processeur s'il modifie la même ligne cache. Ce qui explique pourquoi augmenter le nombre de threads n'améliore pas les performances du programme, mais au contraire le ralentit encore plus.

2. OpenMP casse les optimisations du compilateur en ajoutant du code qui va aveugler le compilateur. Cela explique pourquoi les performances sont nettement inférieure entre le code séquentiel et le code avec un seul thread.

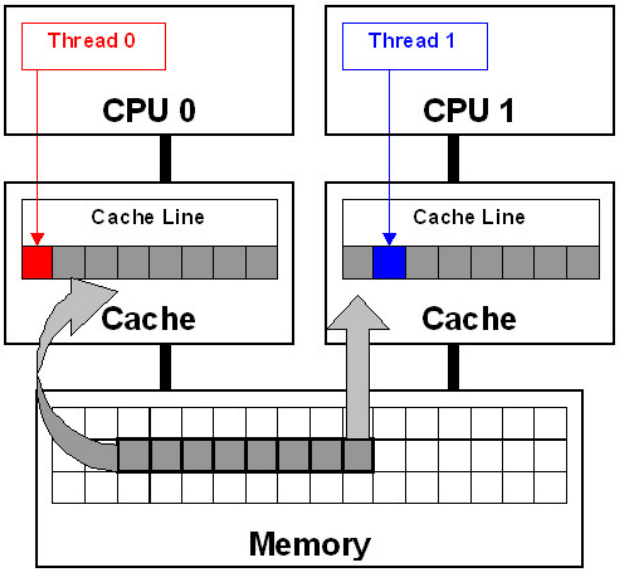

FIGURE 13: Mécanisme de False Sharing.
Source : Documentation Intel.

Pour confirmer notre hypothèse, nous pouvons utiliser le logiciel vTune afin d'analyser cette version parallélisée avec huit threads.

A cause de la limitation de 500MB de collection de données par analyse, le programme s'arrête avant la fin, mais sachant que le programme est constitué d'une boucle principale qui se répète jusqu'à la fin de la partie, les données collectées restent pertinente et représentative de l'ensemble du programme.

FIGURE 14: vTune version parallèle

On remarque que le CPU passe beaucoup de temps à attendre la fin de chaque thread (Spin time) seize secondes sur les vingt secondes d'exécutions et beaucoup de perte de temps en création des threads et en temps de

placement de ces threads (3 secondes sur 20). Le temps de calcul effectif n'est que de 0.6 secondes sur 20, ce qui confirme la non-pertinence du parallélisme tel qu'il a été ajouté.

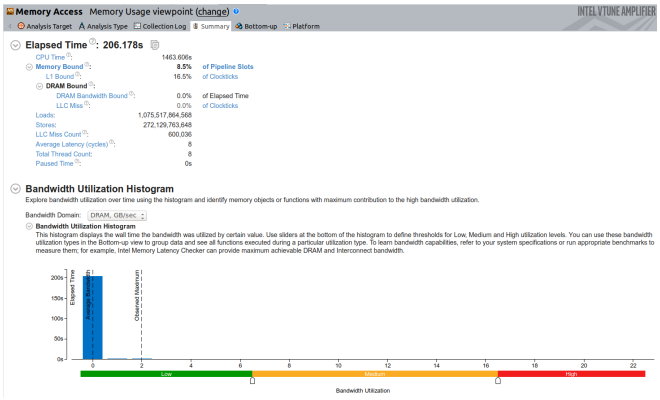

FIGURE 15: vTune version parallèle avec analyse de la mémoire.

L'analyse de mémoire confirme le False Sharing6, en effet il y a 16.5% de "L1 Bound" (défaut de cache L1) qui ont sûrement été causé par le mécanisme de False Sharing expliqué précédemment.

B. Tentative d'amélioration des performances du code parallèle en modifiant le placement des threads

Un facteur qui peut être important dans les performances des codes parallèles est la manière dont sont placés les threads, cette méthode de placement de thread est appelée le "thread affinity".

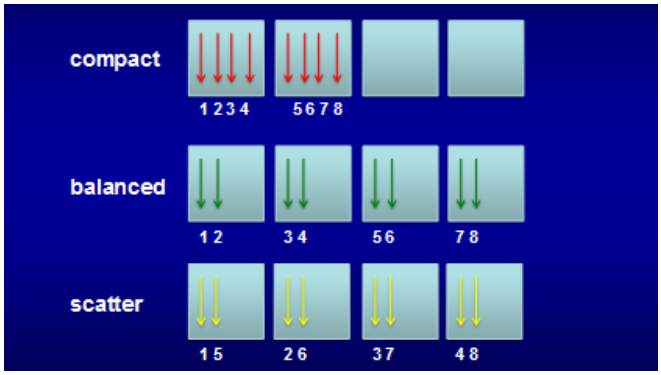

FIGURE 16: Différentes méthodes de placement de thread.

Cornell University

Nous allons voir si modifier le placement des threads améliore ou non les performances parallèles en utilisant le compilateur d'Intel avec huit threads car c'est l'un des compilateurs qui offrait les meilleures performances parallèle et il a une gestion du placement des threads plus complète que Clang. Nous testerons deux méthodes de placement "Compact" et "Scatter"7 supporté par le compilateur Intel en utilisant la variable d'environnement KMP_AFFINITY.

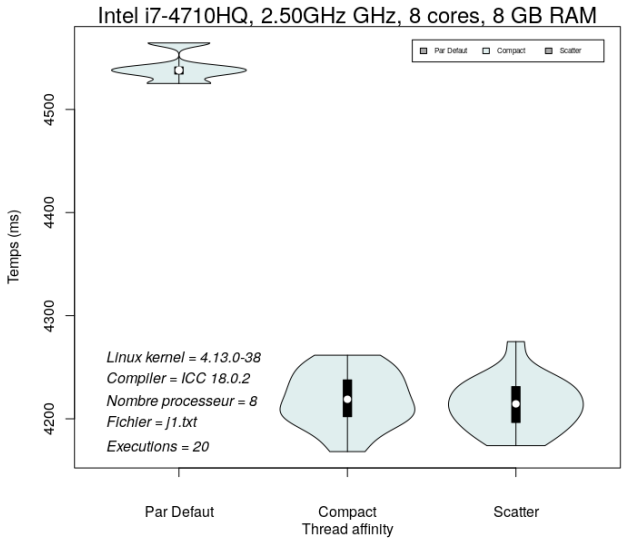

FIGURE 17: Comparaison performances Scatter vs Compact.

On remarque que les performances se sont bien améliorées par rapport au mode de placement par défaut (celle de l'OS lui-même). Les performances de Compact et Scatter sont à peu près équivalentes7 pour ce programme-ci dans cette configuration.

C. Intel Advisor

Advisor est un autre logiciel de la suite Parallel Studio qui, en plus de nous informer sur les fonctions chaudes du programme, nous donne des indications sur comment optimiser le code.

Nous avons vu que l'ajout naïf de parallélisme était défavorable aux performances du programme, nous allons ici tenter d'ajouter du parallélisme de manière plus intelligent à l'aide des outils de la suite Intel Parallel Studio XE.

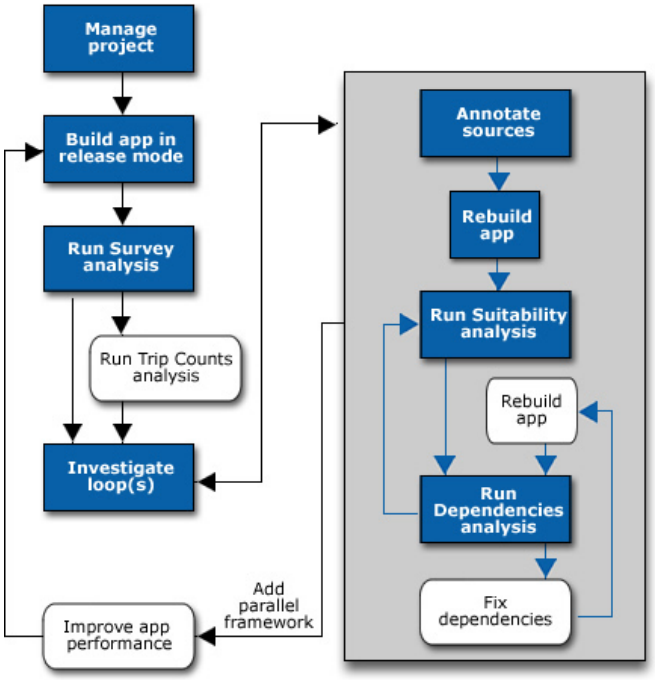

FIGURE 18: Processus itératif d'amélioration des performances.

Source : Documentation Intel.

a. Fonctions chaudes et optimisations D'abord, nous analysons le programme en cherchant les fonctions chaudes et les différents problèmes pouvant réduire les performances du programme.

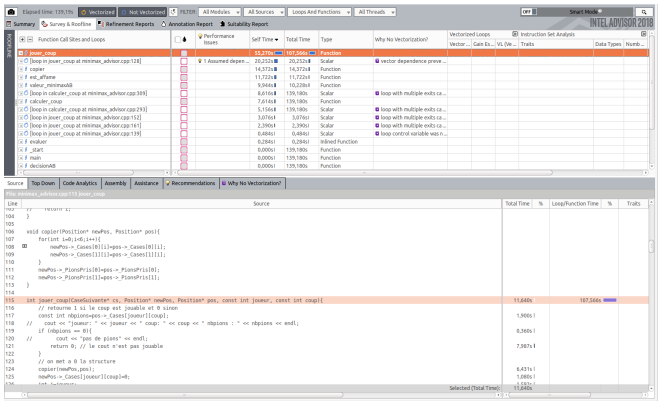

FIGURE 19: Résultat d'Advisor "Survey".

Contrairement à vTune6, Advisor donne aussi des indications sur ce qui nuit aux performances du programme. Par exemple nous avons ici une dépendance de donnée qui empêche la vectorisation. Malheureusement nous ne pouvons pas la supprimer, car c'est une dépendance Read after Write (dite "true dependancy").

1. *Test opportunité ajout de parallélisme*

Il faut donc chercher où il convient d'ajouter du parallélisme afin que cela augmente les performances du programme.

Pour cela nous visons les boucles ou le programme passe plus de 1.5% du temps afin que l'ajout de parallélisme soit un minimum pertinent, idéalement on choisirait les boucles où l'on passe au moins 5% du temps, mais il n'y en a aucune dans ce programme, car il effectue beaucoup d'appel récursif.

Nous ajoutons des annotations autour des zones que nous souhaiterions idéalement paralléliser et nous exécutons une analyse dite de "Suitability".

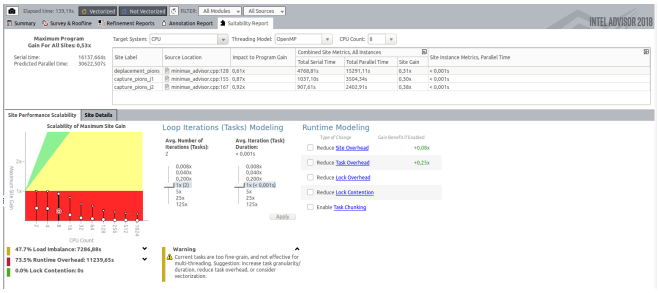

FIGURE 20: Résultat d'Advisor "Suitability".

Malheureusement nous pouvons observer que l'ajout de parallélisme à chaque boucle ne fera que ralentir le programme.

Le logiciel nous indique que les taches effectuées sont d'une granularitée trop fine ce qui induit un "Task Overhead" (le temps passé à créer une tâche et à l'assigner à un thread ainsi qu'à mettre en pause ou arrêter le thread une fois fini) qui est trop important, ce qui pénalise le programme parallèle.

2. *Test de la viabilitée ajout parallélisme*

Malgré le fait que l'ajout de parallélisme ne soit pas pertinent, nous allons tout de même tester le mode "Dependencies" d'Advisor qui sert à détecter si les hotspots dont l'ajout de parallélisme est pertinent découvert précédemment ne possèdent pas des dépendances de données qui empêchent la parallélisation.

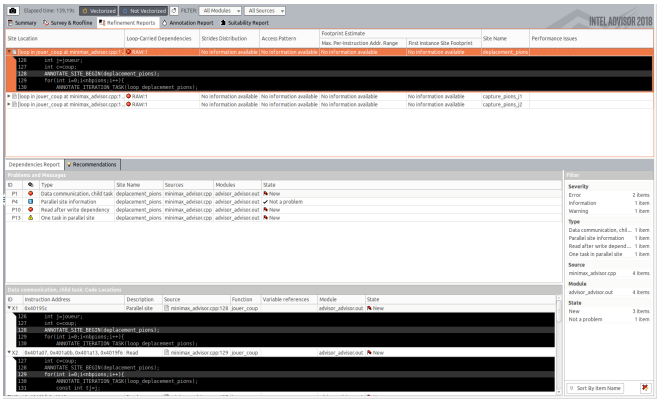

FIGURE 21: Résultat d'Advisor "Dependencies".

En plus de ne pas être pertinent, l'ajout de parallélisme dans les hotspots n'est pas possible à cause des dépendances de données, Read After Write donc réelles dépendances que l'on ne peut enlever. Et il n'est pas non plus possible d'ajouter du parallélisme dans la fonction **decisionAB** à cause du fait qu'il faille remonter alpha et bêta1 lors de l'évaluation des sous-noeuds afin d'effectuer des coupes.

V. MODIFICATION ALGORITHMIQUE DU CODE

Dans cette partie nous allons essayer d'améliorer les performances de notre application en modifiant son algorithme.

Ce programme utilise un algorithme de min-max avec coupes alpha bêta1, il y a quelques voies d'optimisation possible :

1. Nous pourrions réordonner l'ordre d'évaluation des coups afin de faire jouer les coups statistiquement meilleurs en premier (exemple : un puits avec beaucoup de cailloux à plus de chance d'être un meilleur coup qu'un puits avec un unique cailloux). Cela peut améliorer l'efficacité des coupes alpha bêta1.

2. Nous pourrions créer une table de transposition qui associe une position à un score afin de ne pas avoir à recalculer certaines positions. Vu que l'on doit utiliser une fonction de hachage, il y a potentiellement des collisions qui peuvent changer les coups joués par rapport à la version sans table de transposition, mais il est très facile de tester si tel est le cas.

3. On peut créer un fichier d'ouvertures, avec les X premiers coups pré-calculés à l'avance avec une meilleure profondeur. L'inconvénient de cette solution est qu'elle peut changer les coups joués par la nouvelle version du programme par rapport à la version de base, ce qui empêche la comparaison.

A. Trie des coups à visiter

Trier à chaque profondeur à un coup trop important, mais trier à la profondeur zéro est pertinent, car cela peut aider grandement à améliorer la valeur de alpha1 et donc de générer plus de coupe dans les sous noeuds, et ce pour un coût négligeable en temps (trie sur les indices d'un tableau de six cases).

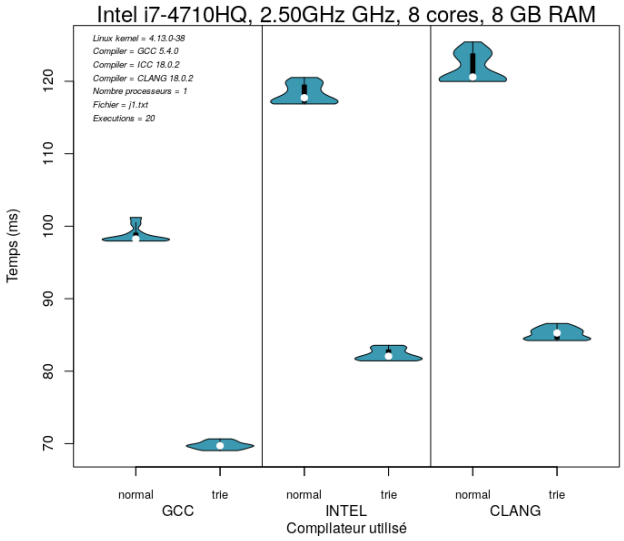

FIGURE 22: Trie coup au premier niveau.

On remarque qu'il y a eu une amélioration des performances assez nette, d'environ 30%. C'est donc une optimisation à garder.

B. Table de transposition

On ne peut pas avoir une table de hashage globale car il y a trop de positions possibles à stocker. Cependant, on peut avoir une table de hashage pour le tour courant, qui nous permet de regarder pour chaque noeud, si il a déjà été évalué et cela peut, peut-être nous éviter certains calculs sans pour autant consommer trop de mémoire.

1. *Ordered Map*

En C++ les **std : :map** sont des arbres binaires de recherche. Pour pouvoir implémenter les maps il faut d'abord changer légèrement la structure de données contenant la position du plateau, passant d'un tableau deux dimensions à un objet de type **std : :array** à une dimension (on linéarise simplement le tableau à deux dimensions).

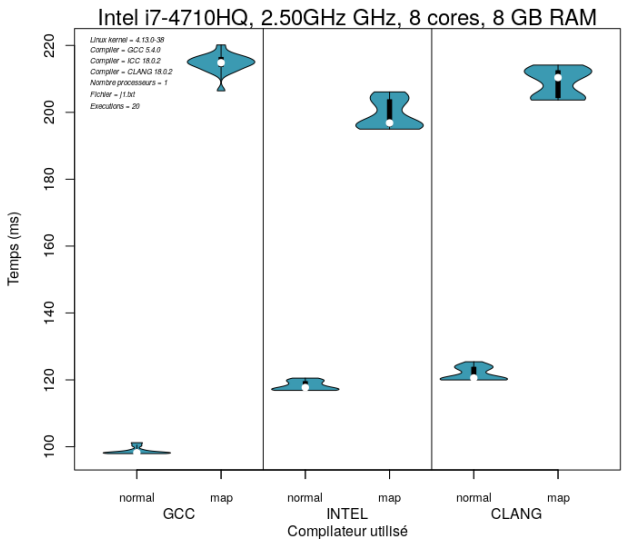

FIGURE 23: Ordered Map.

2. *Unordered Map*

Nous utilisons une fonction de hachage basique trouvé sur StackOverflow, c'est une fonction qui comme beaucoup de fonction de hashage est basé sur des XOR successifs. Elle effectue des XOR entre le hash de chacun des éléments du tableau auquel on ajoute un nombre fixe, l'élément décalé de six bits vers la gauche et de deux bits vers la droite.

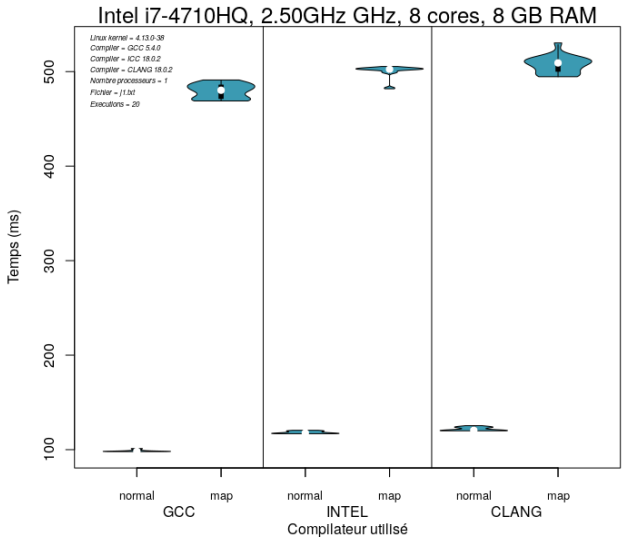

FIGURE 24: Unordered Map.

Dans les deux cas les performances ne sont pas améliorés, cependant la table de hashage ordonnée donne de meilleurs résultats que la table de hashage non ordonnée, cela semble montrer que la fonction de hashage prend beaucoup de temps à être calculée par rapport au coût de notre fonction d'évaluation (simple différence entre le nombre de cailloux prix par l'ordinateur et l'adversaire).

Il est aussi possible que la fonction de hashage ainsi que la structure de donné ne soit pas optimale pour ce jeu, une possibilité serait d'implémenter la fonction de hashage de Zobrist et d'utiliser un grand tableau de liste contenant un couple <Hash,Valeur>, pour retrouver la valeur de notre position, il faudrait faire **Hash%TAILLE_TABLEAU** pour obtenir l'indice du tableau et parcourir jusqu'à trouver notre couple <**Hash,Valeur**>, si le tableau est très grand cela pourrait peut-être accélérer le calcul, cela n'a pas été fait par manque de temps. Néanmoins, il est fort probable que l'ajout d'une table de hashage soit beaucoup moins pertinent que pour des jeux tel que les échecs, car il y a beaucoup moins de positions répétitives que les échecs.

Aux échecs excepté les pions, les pièces peuvent revenir au coup d'après sur leur case d'origine si elles n'ont pas été capturées, si la voix n'est pas bloquée ou si elle ne met pas en échec le roi.

VI. GESTION DU PROJET

Je me suis servi de Github pour héberger mon projet. Je me suis servi de mon ordinateur portable car malheureusement il fallait désactiver le processeur scaling pour éviter de fausser les résultats et cela nécessitait un accès au BIOS, ce qui n'était pas possible avec les serveurs de l'INRIA.

Cela était parfois contraignant, car certains programmes ont tourné quasiment quatre jours, mais aussi parce que la distribution installée étant en CLI, je ne pouvais pas l'installer sur mon disque dur principal (car inutilisable pour des tâches quotidiennes), j'ai donc du l'installer sur un autre SSD et changer les disques quand je devais lancer l'expérience.

Les résultats ont été obtenus en utilisant des scripts que j'ai écrits en bash, toujours lancé avec la commande nohup (afin d'éviter les interruptions systèmes) et env -i afin d'éviter d'ajouter des variables d'environnements non propre au script qui viendrait possiblement polluer l'expérience, excepté pour le script de compilation qui a besoin de variables d'environnement pour faire fonctionner le compilateur d'Intel.

Les courbes ont été tracées avec le langage R et le rapport rédigé avec LaTeXen utilisant le style REVteX.

VII. CONCLUSION

A. Taches effectuées

Nous avons d'abord rendu le programme indépendant d'un système d'exploitation particulier en remplaçant les

appels à la bibliothèque de Windows par des appels à la librairie standard.

Nous avons ensuite comparé les performances des diverses options de compilations et compilateurs possible. Suite à quoi nous avons essayé de déterminer ce qui ralentissait le programme, nous avons déterminé que le programme était limité par l'usage du CPU.
Nous avons donc tenté d'ajouter du parallélisme d'abord de façon naïve en en ajoutant partout où cela était possible, puis ensuite en utilisant la suite de logiciel d'aide à l'optimisation d'Intel.

Les performances parallèles n'étant pas satisfaisantes, nous avons essayé de modifier le placement des threads, cela a amélioré les performances, mais les performances restaient inférieures aux performances du programme séquentiel à cause du mécanisme de False Sharing6 que nous avons mit en évidence.

Nous avons ensuite modifié l'algorithme, tout d'abord en triant les coups à évaluer à la profondeur 0 en évaluant les coups statistiquement les plus probables d'être des bon coups (les puits avec le plus de cailloux) et cela à permit d'améliorer les performances de 30%.

Comme autre modification algorithmique, nous avons essayé d'ajouter une table de hashage, mais cela a dégradé les performances à cause du coût d'insertion et du grand nombre de positions qui contrairement à beaucoup de jeux à deux joueurs peuvent rarement se répéter au cours d'une même partie.

B. Tâches restantes

Il y a encore beaucoup de voie d'optimisations possibles et d'expérimentations à mener. Je n'ai pas eu le temps de mener à terme mon expérience qui constituait à pré-calculer toutes les positions que l'on peut jouer en parallèle à chaque niveau du MinMax1, l'expérience a été mené avec une seule répétition par configuration, le résultat n'était pas concluant, là aussi le mécanisme de False Sharing6 intervient, mais cela n'est pas assez significatif pour être utilisé, mais si cette tendance viendrait à se confirmer, on pourrait essayer de supprimer le False Sharing6 en ajoutant un tableau de la taille d'une ligne cache dans la structure position afin d'éviter ce phénomène.

On pourrait aussi essayer d'améliorer le trie des coups en propageant un tableau des puits avec le plus de cailloux dans la fonction **jouer_coup** afin de maintenir l'ordre lorsqu'on joue un coup sans avoir à re-trier à chaque fois.

Et enfin on pourrait aussi implémenter la fonction de hashage de Zobrist vu plus haut, même si je suis plutôt dubitatif par rapport au potentiel de cette modification dans le cas du jeu de l'Awalé2.

C. Problèmes rencontrés

Ne voulant pas toucher au code source originel, j'ai commencé mes expérimentations sur Windows et il m'a fallu recommencer sur Linux une fois que M.Touati m'a expliqué pourquoi c'était une erreur et heureusement, car j'aurais très vite été bloqué dans l'analyse des performances.

J'ai eu du mal à prendre en main le langage R, ainsi que l'installation de certaines librairies comme wvioplot qui étaient un peu complexe à installer.

La prise en main du logiciel Advisor d'Intel ne fut pas facile, car il fallait lancer le programme depuis la ligne de commande pour pouvoir re-diriger l'entrée standard depuis un fichier d'entrée et la documentation est surtout faite pour la version graphique du logiciel. De plus, les expérimentations avec Advisor étaient très longues, environ une journée pour chaque type d'analyse.