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

Pierre TASSEL

(Dated: 11 mai 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 sur l'une des variantes du jeu Awale avec 6 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 exécutions ainsi que de tenter de l'optimiser par différent moyen.

I. INTRODUCTION

A. Programme à analyser

Le code à analyser étant dépendant de Windows (import de la bibliothèque windows.h), il faut le réécrire pour permettre de le rendre non dépendant d'un os en particulier. Cette bibliothèque étant utiliser pour les compteurs de temps, il est donc facile de réécrire le programme afin qu'il soit indépendant de l'OS.
Ce programme utilise un MinMax avec coupes alpha bêta pour choisir le coup à jouer, le principe d'un algorithme MinMax est de parcours 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 utilisant une fonction qui notera les différentes situations. Les coupes alpha bêta sont juste 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.

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 service possible installé. Nous utiliserons donc Arch Linux, basé sur le noyau Linux 4.14.13-1 cette distribution possède le minimum d'applications et de services possible installé.

2. Matériel

Notre environnement expérimental est un ordinateur portable utilisant un processeur Intel I7-4710HQ, 8Go de RAM en DDR3, un SSD et le processeurs scaling a été désactivé dans le BIOS, cela permet de faire tourner le processeur à vitesse constante et ainsi ne faussant pas les résultats.

II. ANALYSE DES OPTIONS DE COMPILATIONS OPTIMAUX

A. Démarche expérimentale

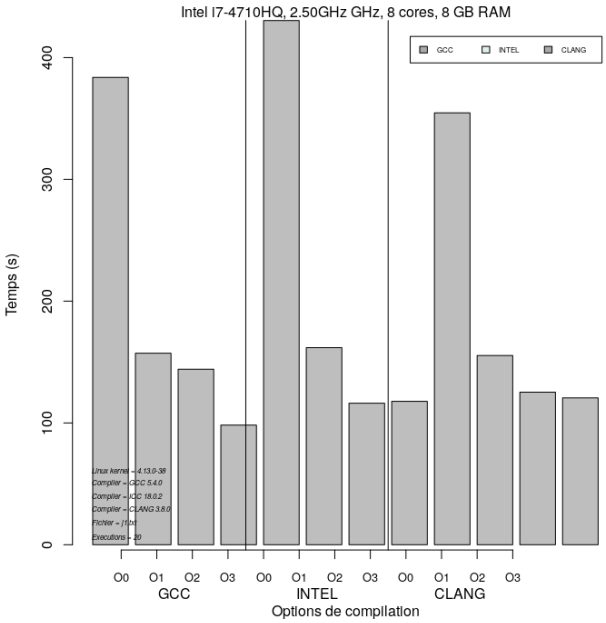
Il faut tout d'abord modifier le programme pour qu'il n'affiche en sortie uniquement le temps d'exécution de la partie et non plus les informations utiles aux joueurs.
Ensuite, il faut obtenir un ensemble de données. Sachant que le programme est déterministe (pas de modification de la profondeur en fonction 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 utilisé) et nous obtiendrons le temps d'exécution de chacun.
Afin d'obtenir les entrées du programme, nous avons fait tourner deux instance 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 une script exécutant les différentes compilations et exécuteras les programmes 20 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 lent à être totalement exécuter (il y a 480 exécutions différentes au total) nous l'exécuterons au travers de la commande nohup afin d'éviter toute interruption possible.
Les inputs sont disponibles dans le dossier input.

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 configurations.
Les résultats sont disponibles dans le dossier resultats.

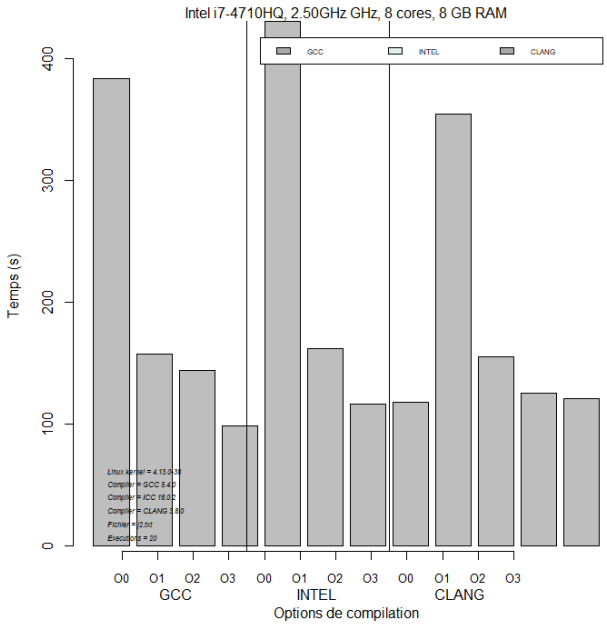
1. Analyse globale

Temps d'exécutions quand le joueur commence



a. Joueur Commence

Temps d'exécutions quand le programme commence

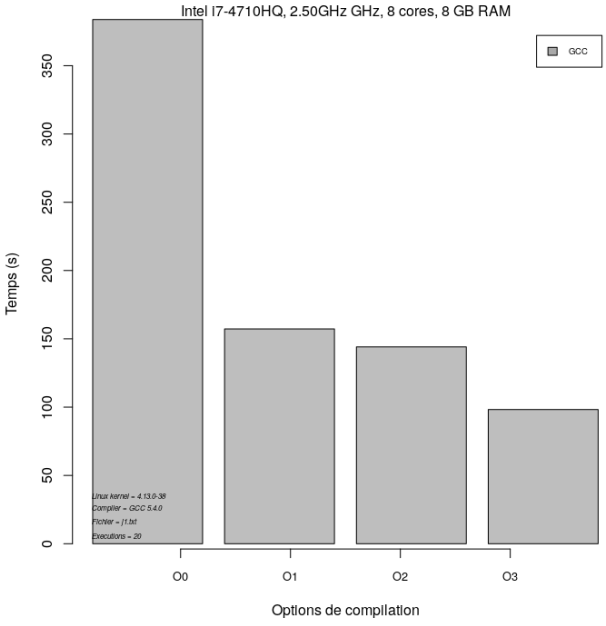


b. Programme Commence

2. Analyse compilateur par compilateur

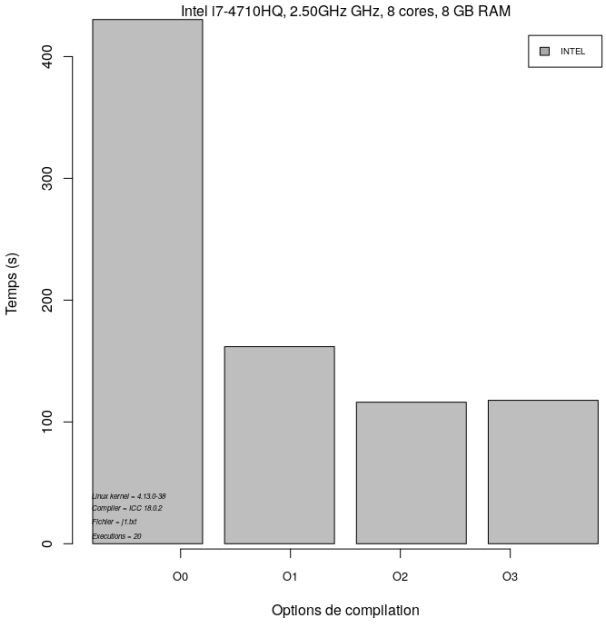
Comme nous l'avons vu précédemment, le fait que le joueur commence ou non n'a pas d'influence sur le choix du compilateur. Nous nous concentrerons dons sur les parties où le programme commence pour analyser plus finement les écarts de temps d'exécutions entre les diverses configurations.

Temps d'exécutions diverses configurations GCC



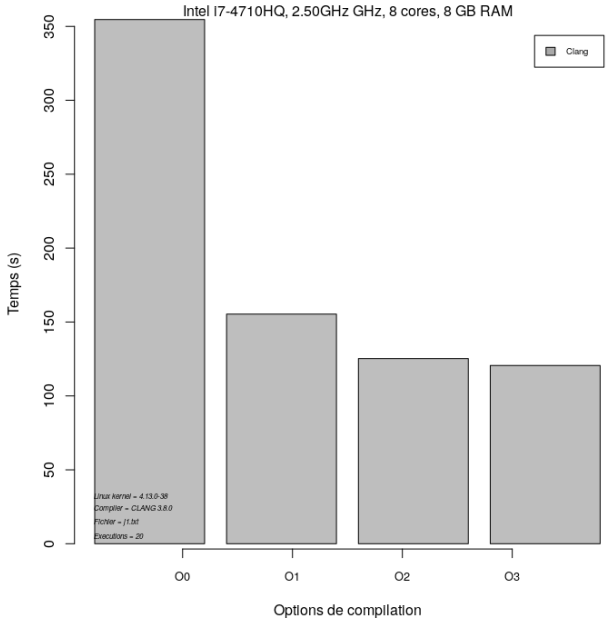
a. GCC

Temps d'exécutions diverses configurations Intel



b. Intel

Temps d'exécutions diverses configurations Clang



c. Clang

C. Analyse des résultats

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

1. GCC

Chaque niveau d'optimisation réduit grandement le temps d'exécution, il y a un rapport de 4 entre la compilation sans optimisation (O0) et la compilation avec le plus d'optimisations (O3).

2. Intel

Il y a un gros écart entre la compilation sans optimisation et celle avec un minimum d'optimisation (rapport de 3). Mais étonnement, la version avec le plus d'optimisation (O3) est très légèrement plus lente que celle avec un peu moins d'optimisations (O2).

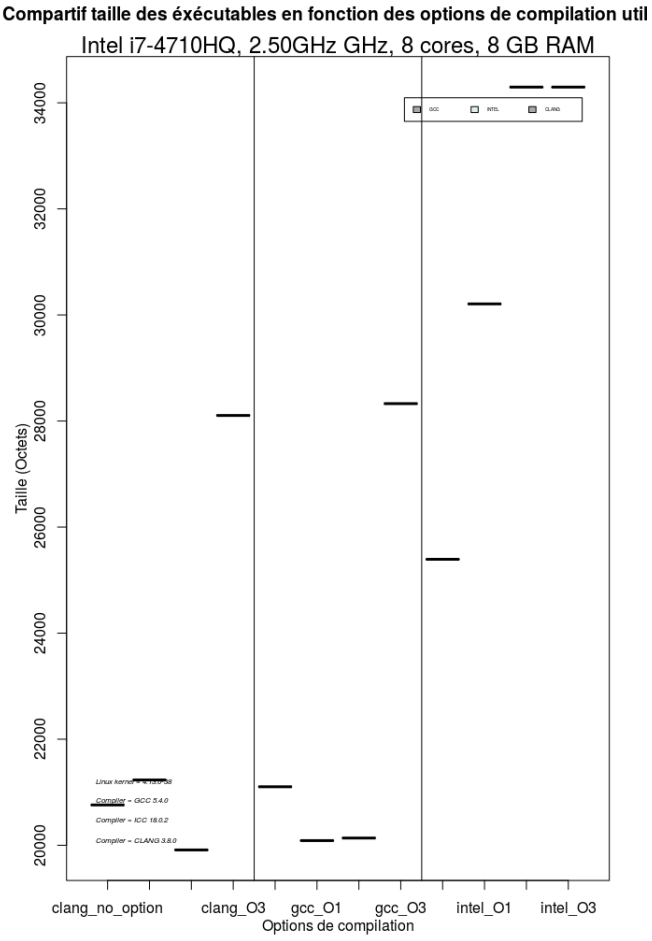
3. Clang

Sans optimisations particulières, Clang est plus rapide que les autres compilateurs sans options de compilations. Il n'y a par contre pas beaucoup d'améliorations entre la

version la plus optimiser (O3) et celle avec un peu moins d'optimisations (O2).

4. *Taille des éxécutables*

Comparatif des tailles d'éxécutables produits par les diverses compilations



La aussi GCC avec l'option O3 semble être la aussi la meuilleure option pour ce programme.

III. PROFILAGE DU CODE

A. Gprof

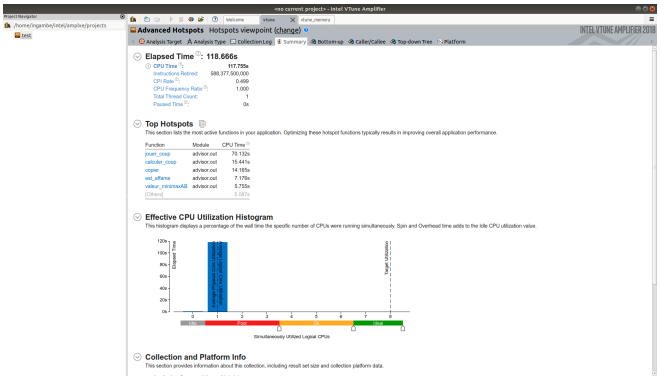
Each sample counts as 0.01 seconds.

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*
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_positionP3
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_

Il nous apprend que la fonction la plus utilisé est jouer_coup, et ce très largement. On note aussi que la fonction decisionAB est moins souvent appelé mais chaque appel est d'une plus longue durée.
Ce sont donc les deux fonctions qu'il faut tenter d'optimiser en priorité pour améliorer les performances.

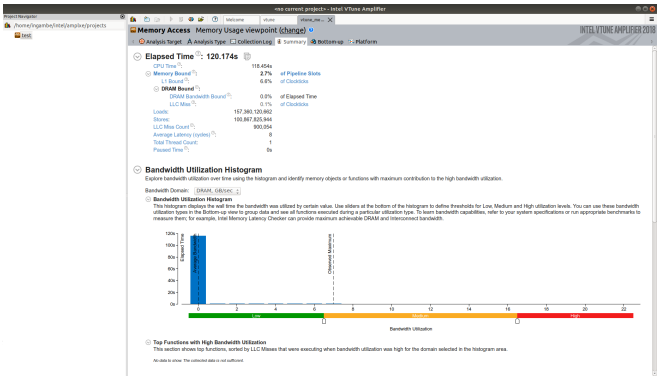
B. vTune

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 8 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.

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).
Il y a très peu d'erreurs de cache et les accès mémoire sont petits.

IV. AMÉLIORATION DU CODE

Nous allons ensuite tenter d'améliorer le code avec les outils de la suite Parallel Studio XE d'Intel

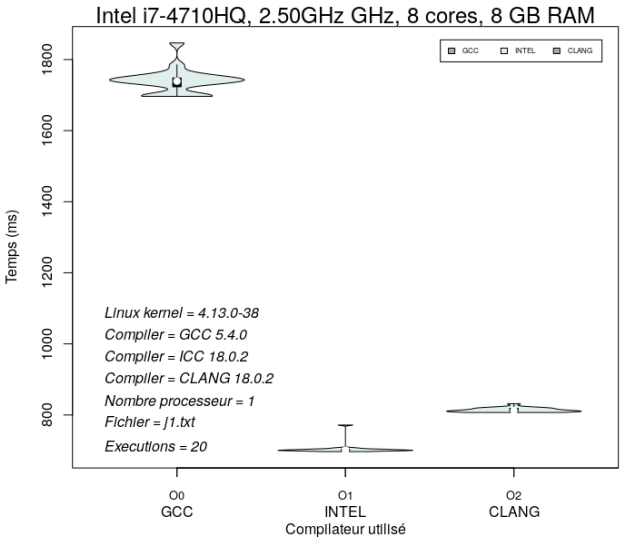
A. Parallélisme

1. Naïf

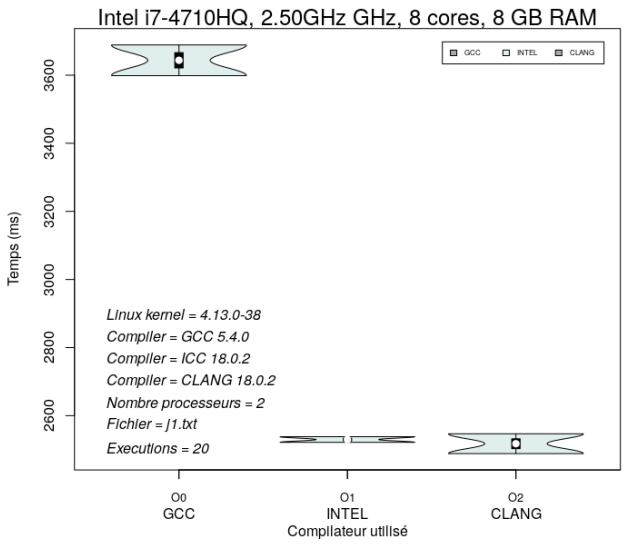
On a ajouté du parallélisme un peu partout où il n'y a pas de dépendance de données en utilisant la librairie OpenMP.
Ensuite, nous réitérons ce que nous avons fait pour la version sans parallélisme en utilisant chaque compilateur avec l'option O3 activé mais, en faisant varier le nombre de processeurs.

a. Résultats

Parallélisme naïf avec un processeur



Parallélisme naïf avec deux processeurs

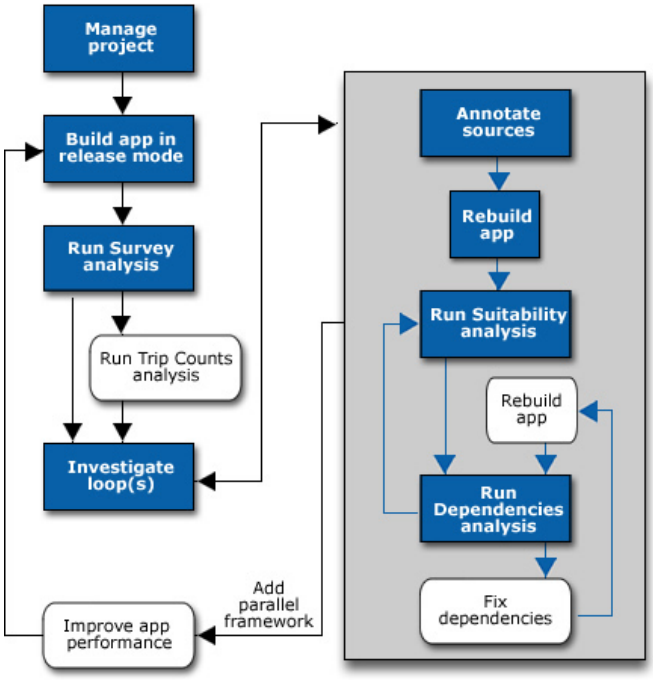


Nous devons arrêter l'évaluation ici, car l'analyse prend malheureusement trop de temps, il a fallu plus de deux jours et demi complet pour obtenir les données pour deux processeurs.
Seulement, nous pouvons déjà observer que l'ajout de parallélisme ralentit grandement la vitesse d'exécution du programme.

2. Advisor

Advisor 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.

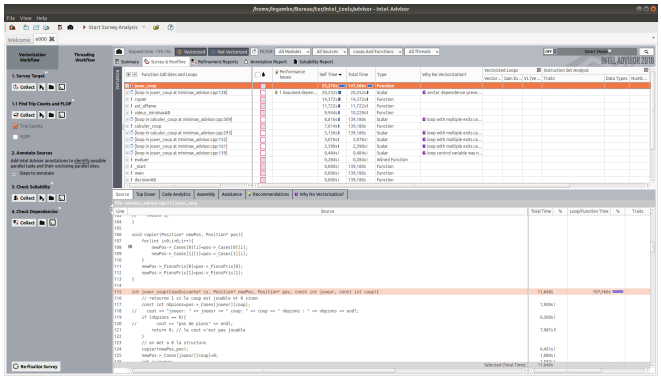
Processus itératif d'amélioration des performances



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.

Résultat d'Advisor "Survey"



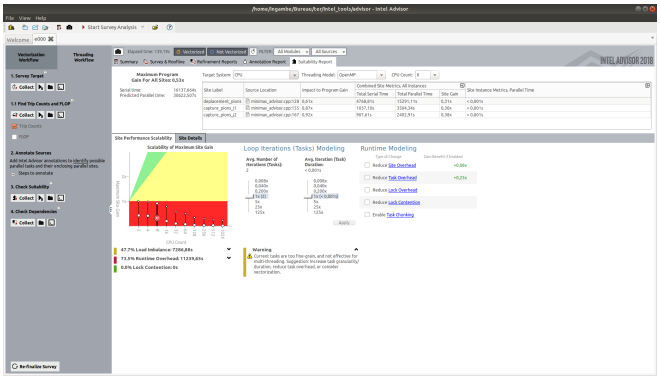
Contrairement à vTune, Advisor donne aussi des indications sur ce qui nuit aux performances du programme. Par exemple nous avons ici une dépendance de donnée qui empêche la vectorisation. Malheureusement nous ne pouvons pas la supprimer, car c'est une dépendance Read after Write (dite "true dependancy").

b. Test opportunité de parallélisme

Il faut donc chercher où 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 pertinent.
Nous ajoutons des annotations autour des zones que nous souhaiterions idéalement paralléliser et nous exécutons une analyse dite de "Suitability".

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 l'on peut un peu améliorer les performances du parallélisme en augmentant la taille des zones parallel, mais ce n'est pas suffisant pour rendre le speed up (ndlr : différence de vitesse entre le code séquentiel et le code parallel) pertinent.

c. Test de la viabilité ajout parallélisme

Mal grès le fait que l'ajout de parallélisme ne soit pas pertinent, nous allons mal grès tout utiliser le mode de détection de dépendance d'Intel afin de savoir si les hotspots sont parallélisables.

Résultat d'Advisor "Dependencies"