Performances des programmes multi-threadés

François Robert et Nicholas John Loughran Cree

November 11, 2019

1 Utilisation de gettimeofday

On doit placer les appels à gettimeofday au debut et au fin de l'algorithme qui fait la recherche dans le vecteur. Dans la version parallèle du programme on doit inclure la creation de threads et la synchronisation parce que c'est du temps quand le cpu est utilisé aussi.

L'ordonnanceur et les autres utilisateurs jouent un role, l'ordonanceur va endormir les threads pour partager les ressources de la machine, prenons l'exemple d'un thread commencant son execution a un temps t1, endormi a un temps t2, reveillé en t3, et finissant son éxecution en t4. Le temps réel d'execution de ce thread serait (t2-t1) + (t4-t3) cependant, avec gettimeofday() on prendrait aussi la mesure du temps durant lequel le thread est endormi, et comme c'est l'ordonanceur qui gère les ressources en fonctions des utilisations du processeur, ces deux paramètres jouent un role.

2 Utilisation de getrussage

Le nombre de thread impact sur la valeur retourné : le temps utilisateur rensigné peut ne pas changer, car il représente la somme des teps d'executions des threads du processus cependant le temps superviseur augmente si le nombre de thread augmente par rapport au nombre de processeur, il y a un nombre de thread optimal a choisir pour executer son programme dans es meilleurs conditions possible (nous le verrons dans les experiences plus tard) et si nous dépassons ce nombre l'ordonanceur va passer plus de temps à gérer les threads.

Le mesure du temps avec gettimeofday mesure tout le temps passé mais getrusage mesure seulement le temps du CPU. Getrusage est inférieur à gettimeofday parce que il n'inclut pas le temps de creation de threads et le temps de synchro, seulement le temps de CPU. C'est mieux utiliser gettimeofday quand on fait la mesure d'execution d'un algorithme parallèle.

3 Performance d'un programme

3.1 Tracez les courbes d'accelération et d'efficacité correspondantes.

Expérimentation effectuées : nous avons choisi de faire deux collections de graphe une collection représentant l'evolution de l'efficacité et de l'accélération de notre algorithme de tri en version sequentiel par rapport a la version mlti threadé, et ce pour une taille fixe, avec le nombre de thread qui évolue. La collection est composée de 9 graphes, donc 9 valeurs de taille de vecteur à trier qui sont fixés, puis on fait evoluer le nombre de threads. De plus, il y a 100 execution du tri pour chaque taille donnée et nombre de thread donné. Le script R que nous avons utilisé pour obtenir ces graphes est un copier/coller du script exemple fourni avec le TP, nous lui fournissons l'ensemble des données récoltés par les 100 executions pour chaque nombre de thread pour tracer un graphe.

Une deuxième collection représentant pour un nombre de thread fixe, l'évolution de l'accélération et de l'éfficacité en fonction de la taille du vecteur. Il y a 4 graphes dans cette collection, le nombre de thread fixé étant soit 2, 4, 8, ou 16. Le script R que nous avons utilisé pour obtenir ces graphes est un copier/coller du script exemple fourni avec le TP, nous lui fournissons l'ensemble des données récoltés par les 100 executions pour chaque taille pour tracer un graphe.

Les vecteurs à trier ont été crées à l'avance, ils ont une taille variant de 1.000.000 à 5.000.000, les valeurs de chaque éléments du vecteur ont une valeur entre 0 et 1000 (valeur par défaut) Nous n'avons pas fait varier l'ensemble des valeurs possible pour ne pas se mélanger dans toutes les variantes possible et imaginable de tests, cependant nous savons que plus l'ensemble de valeur est grand, plus le vecteur sera long a trier (car plus grande probabilités de devoir déplacer un élément et donc le temps d'éxecution augmente) Nous n'avons pas non plus fait varier l'algorithme de tri, qui selon comment il est implémenté peut être optimisé pour être utilisé en multi threadé, ou bien en séquentiel, en l'occurence la version proposée semble repartir le vecteur en sous vecteurs en fonction du nombre de threads, il semblerait qu'il soit parti pour être plus performant en multi threadé qu'en séquentiel et c'est ce que nous verrons dans l'études de nos graphes. Pour calculer l'accélération et l'efficacité pour chaque collection de donnée il faut également d'autres données, les ressources utilisées (nombre de processeurs notamment) aisni que le temps minimum que l'algorithme en sequentiel met à trier le vecteur d'une taille donnée, nous avons éxécuté le programme sur mandelbrot qui possède 8 processeurs milti coeurs, mais qui doit partager ses ressources entre tout les utilisateur connectés, ne connaissant pas cet inconnu nous ne pouvons pas estimé des ressources réelles associé à nos éxécution, et ne pouvons pas garantir que chaque execution ait eu autant de ressource que les autres, cependant le fait de faire une moyenne sur 100 éxecution permet d'obtenir une valeur représentative, la marge d'erreur apparaissant sur les graphes étant faible.

3.1.1 Graphs avec le nombre des threads fixée.

Etudes de la collection 1 : les graphes ou le nombre de thread evolue pour une taille de vecteur fixé à l'avance Plusieurs informations importantes ressortent de ces graphes : Le fait que quand on choisit un nombre de thread qui ne coincide pas avec les ressources de la machine, les performances chutent, on peut déduire de nos 9 graphes que le nombre optimal sur cette machine à ce moment la semble être 4 threads. Le fait que le nombre de thread optimal dépend également de la taille du vecteur à trier, en effet si le vecteur est petit et qu'on donne un nombre de thread grand les performances chutent, on perd plus de temps que nécéssaire a gérer les threads, au contraire si on ne met pas un nombre de thread suffisant, on tend vers le temps d'éxecution de la version séquentielle (valeur d'accélération de 1). Nous le voyons pour le nombre de thread = 8 : la valeur de l'accélération est plus faible que pour 4 threads, cependant la valeur augmente un peu quand la taille augmente

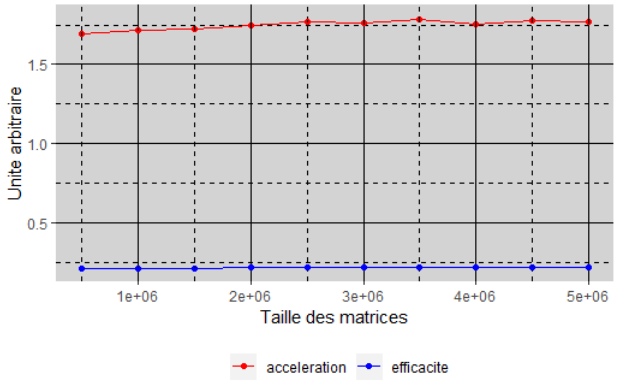

Figure 1: Nombre threads fixée à 2

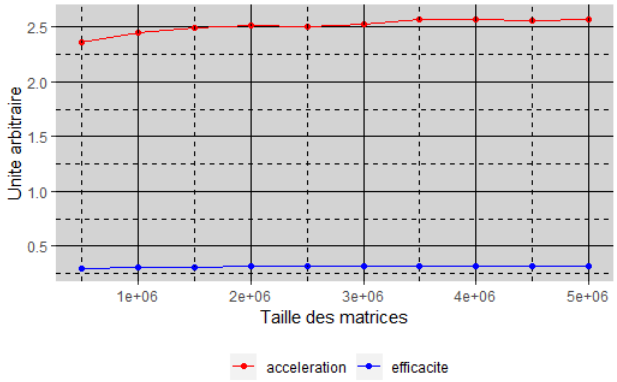

Figure 2: Nombre threads fixée à 4

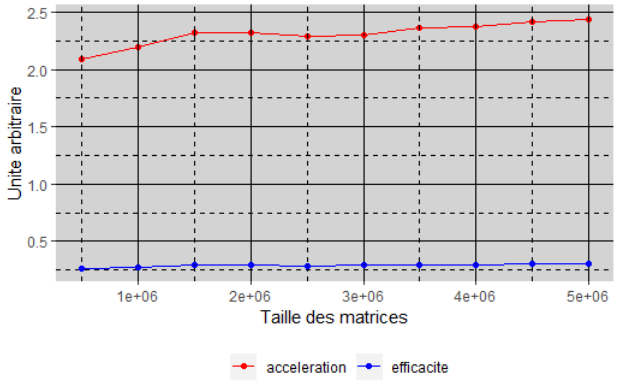

Figure 3: Nombre threads fixée à 8

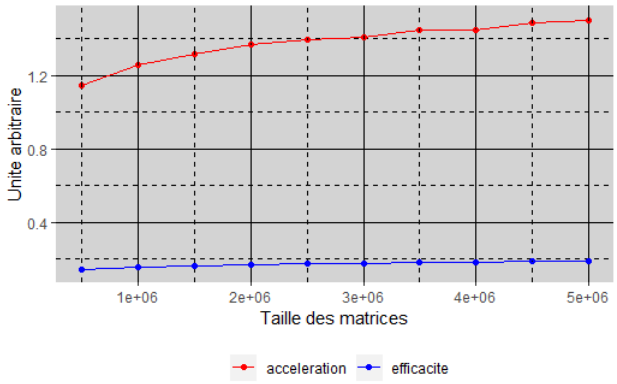

Figure 4: Nombre threads fixée à 16

3.1.2 Graphs avec la taille de vecteur fixée.

Etudes de la collection 2 : les graphes ou le nombre de thread evolue pour une taille de vecteur fixé à l'avance nous pouvons confirmer des informations que nous avosn vu dans la précédente collection avec ces graphes : Le fait qu'il faut bien choisir le nombre de thread, d'une part en fonction de la machine mais aussi en fonction de ce qu'il faut éxéctuer, le premier graphe nous montre une valeur de l'accélération et efficacité plus faible que les autres, c'est normale étant donné qu'on ne donne pas assez de ressources au programme, on pourrait lui donner un nombre de threads plus grand qui se rapprocherait du nombre otpimal, qui semblait être 4. Comme on l'avait déduis précédemment le graphe pour 4 threads démontre les meilleurs performances, tandis que pour plus de thread les performances chutent. Par ailleurs nous remarquons encore une fois que quand on donne un nombre trop grand de thread (ici 16 mais on aurait pu choisir 32 pour pousser un peu plus loin le résultat), les performances sont plus basses que pour un nombre plus petit certe, mais la performance augmente quand la taille du vecteur augmente, donc cela prouve que la taille du vecteur est bien en lien avec le choix à faire

pour le nombre de thread.

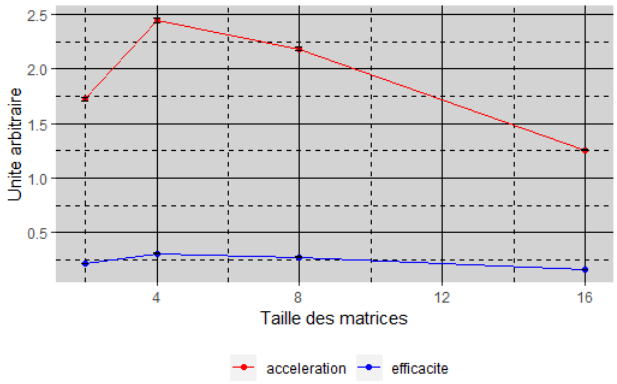

Figure 5: Taille fixée: 1000000 elements

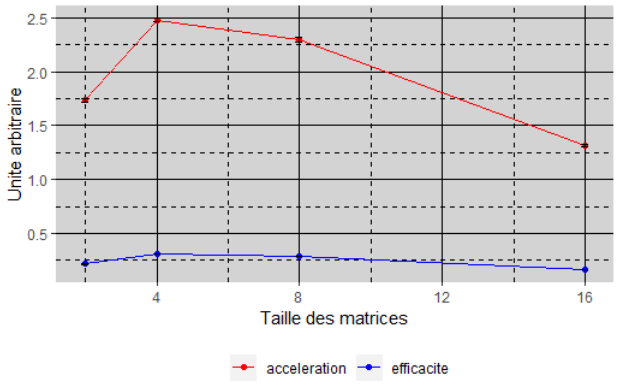

Figure 6: Taille fixée: 1500000 elements

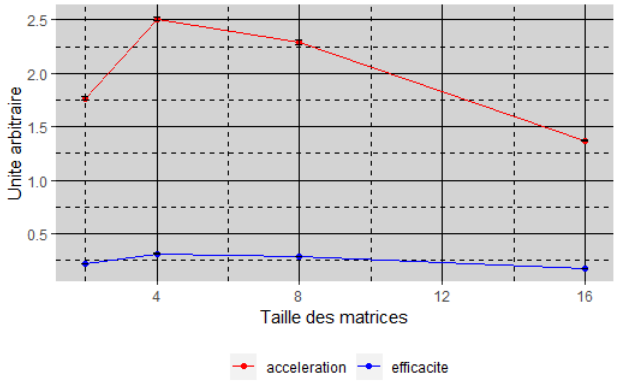

Figure 7: Taille fixée: 2000000 elements

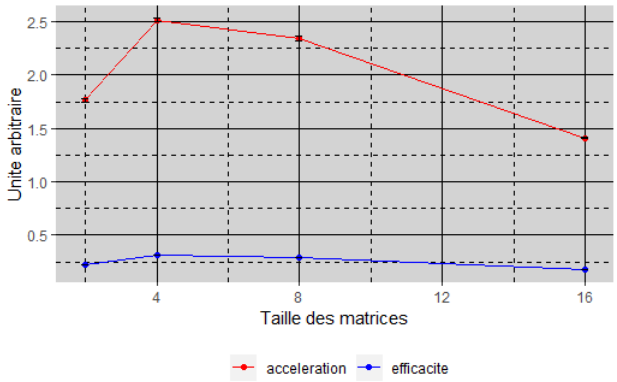

Figure 8: Taille fixée: 2500000 elements

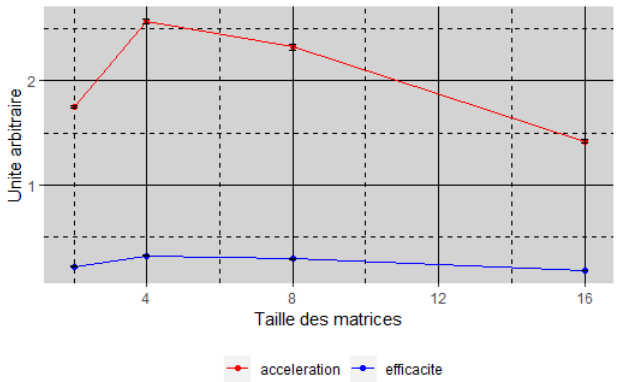

Figure 9: Taille fixée: 3000000 elements

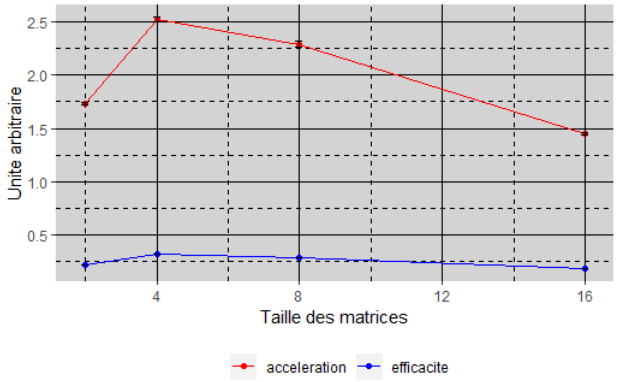

Figure 10: Taille fixée: 3500000 elements

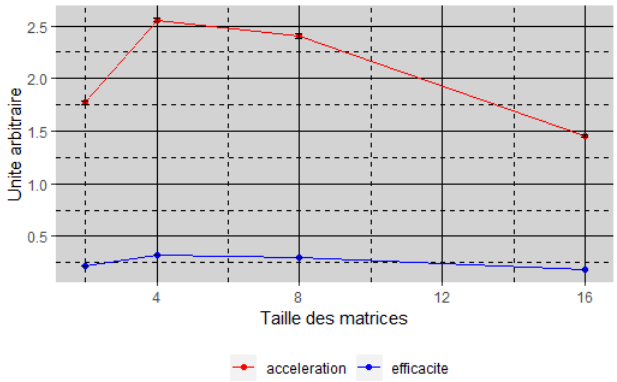

Figure 11: Taille fixée: 4000000 elements

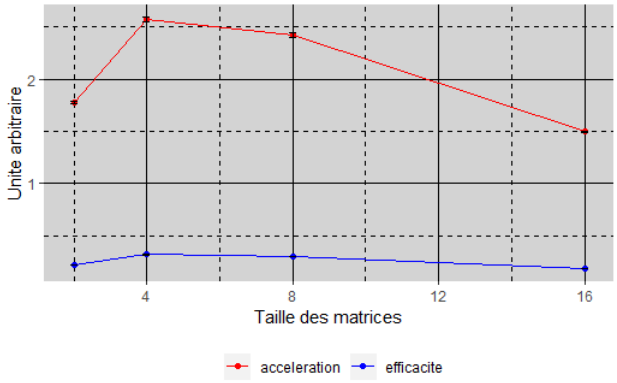

Figure 12: Taille fixée: 4500000 elements

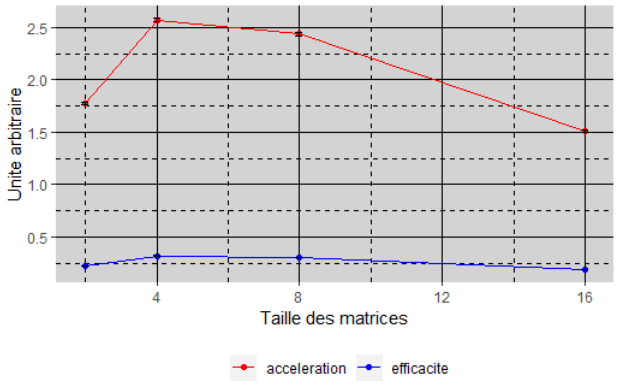

Figure 13: Taille fixée: 5000000 elements

3.2 Quels facteurs influencent l'accélération et l'efficacité d'un programme parallèle?

D'après nos résultats d'experimentation nous avons pu observer certains des facteurs qui sont listé et les avons déjà explicité, nous allons parler plus en details de ceux qui n'ont pas été traité Comme nous l'avons vu l'algorithme utilisé peut influencer l'execution d'un programme parallèle, selon s'il est codé de sorte à être fait en parallèle de manière efficace ou nous le nombre de thread utilisé, si le nombre est ben chosit comme nous l'avons vu dans les expérimentation, permet d'avoir de meilleurs performances, des moins bonnes performances si le nombre est vraiment mal choisit (plusieurs facteurs à prendre en compte) la bibliothèque de thread peut également jouer, d'une bibliothèque à une autre peut être qu'en fonction de l'architecture de la machine une sera plus performante qu'une autre (nous ne l'avons pas testé) le système d'exploitation joue égalment un rôle, l'ordonnanceur étant un acteur clé dans l'éxécution d'un programme parallèle le nombre de processeur de la machine joue également, pour un nombre de thread otpimal choisit, plus il y a de puissance de calcul dans une machine plus l'éxécution serait performante l'architecture mémoire de la machine jouerait également un role dans le cas d'accès mémoire dans le programme, plus rapide seraient les accès mémoire, meilleures seraient les performances S'il y a des utilisateurs connectés sur la machine, la machine doit alors partagé ses ressources, les performances sont alors forcément moins bonne que si on avait toutes les ressources pour nous seuls.