Projet de système d'exploitation 2 : trions ensemble

Romane Castel

Pour le dimanche 15 avril 2018

1 Introduction

Un tri externe est un algorithme qui utilise la mémoire de masse pour stocker des résultats partiels, comme par exemple des sous-fichiers triés du fichier donné en entrée. L'intérêt pour les tris externes est grandissant avec la taille des données fournies en entrée d'un programme. En effet, dans l'exemple du tri dans un fichier, on peut être amené à manipuler des données qui peuvent ne pas être totalement contenues dans la mémoire vive. Cependant, travailler avec la mémoire de masse induit un temps plus long de traitement des données, car l'accès au disque dur est plus long que celui à la mémoire vive.

Ce projet a pour but de mettre en oeuvre différentes versions de tri externe (tri d'un fichier d'entiers) à partir d'un code de base fourni, et de comparer leurs efficacités. Ce code squelette est principalement composé de deux étapes : le tri des nb_split sous fichiers construit à partir du fichier donné en entrée, et les merges de tous les résultats qui sont des fichiers triés.

Nous verrons qu'une première amélioration apportée à cet algorithme sera la parallélisation des tris des sous-fichiers issus du fichier donné en entrée. Une seconde amélioration consistera à en plus paralléliser en deux cascades les merges des fichiers résultats.Une troisième amélioration sera de paralléliser totalement les merges des fichiers résultats avec n cascades (structure d'arbre binaire).Enfin nous jouerons avec la fonction de tri, lors du tri des nb_split sous-fichiers, afin de voir quel tri permet une meilleure efficacité temporelle de l'algorithme de tri externe.

Tout au long de cette étude, les tests seront réalisés sur un ordinateur possédant un processeur Intel Core i5, 4 coeurs, une mémoire de masse de 1To, et une mémoire vive de 8Go. Le système d'exploitation sur lequel le travail aura été effectué est : Debian stretch. Pour réaliser les tests d'efficacité (temporelle) des différentes versions de l'algorithme, j'ai réalisé un script python (il est forni dans le code) qui permet de récupérer le temps réel qu'a pris le tri en fonction de nb_split. On travaille sur un fichier en entrée d'une taille de 50.000 entiers.

2 Vocabulaire

Pour que cette étude soit claire, il y a un ensemble de conventions/notations à poser et de définitions à donner. Listons-les.

- Définitions
  - **fork**: cette commande permet, à partir d'un processus père, de créer un processus fils. A l'issu de la commande, le processus père et le processus fils exécutent du code de manière concurrente.
  - **merge**: on appelle un merge entre deux fichiers triés leur fusion. Pour cela, on compare d'abord les deux premiers entiers de chaque fichiers et on écrit le plus petit des deux (celui du fichier 1 par exemple) dans un troisième fichier. Ensuite on fait cela avec le premier entier du fichier 2 et le deuxième entier du fichier 1... et ainsi de suite. Ainsi on construit un troisième fichier qui est lui aussi trié, et qui possède les entiers du fichier 1 et ceux du fichier 2.
- Notations
  - **i_file** : fichier d'entiers donné en entrée. Il n'est pas trié.
  - **nb_split** : paramètre donné par l'utilisateur. Nombre de sous-fichiers issu de i_file.
  - **filenames** : i_file a été coupé en nb_split sous-fichiers (non triés, de tailles quasiment égales). filenames est un tableau contenant leurs noms.
  - **filenames_sort** : les nb_split fichiers d'entiers ont été triés. filenames_sort est un tableau qui contient le noms de ces derniers.
  - **o_file** : fichier rendu en sorti. Il est composé des mêmes entiers que i_file, mais ils sont triés par ordre croissant.

3 Gestion de fichiers temporaires

Dans le code de base fourni, on a besoin de gérer des fichiers temporaires, et donc de les supprimer.
Initialement, la primitive system se chargeait de supprimer les fichiers temporaires. Néanmoins, celle-ci présente des failles de sécurité. C'est pourquoi l'on peut utiliser une primitive exec pour effectuer cela. Or, cette primitive remplace le code d'un processus en cours (recouvrement). Donc le problème qui se présente est que, dès la première utilisation de la fonction qui supprime un fichier temporaire, toute la suite du code est compromis. C'est pourquoi, dans cette fonction de suppression, on doit isoler exec, et donc créer un processus fils qui se charge de cette primitive.

4 Une première amélioration: la parallélisation des tris des sous-fichiers

Dans le code initialement fourni pour le projet, la partie tri de fichiers ne se fait pas de manière parallèle. En effet, on parcours juste le tableau filenames et on trie chacun des fichiers à la suite. Quand on les trie, on met le nom du fichier dans filenames_sort. Dans une première version de l'algorithme amélioré, on a utilisé la commande fork pour paralléliser les nb_split fichiers.
On a un processus père qui crée nb_split processus fils à l'aide de la commande fork. Chacun des processus fils prend en charge un nom de fichier de **filenames**, c'est-à-dire que chaque fils trie le fichier qu'il lui est indiqué. Le tri appliqué sur les fichiers est le quick sort (tri rapide de complexité temporelle moyenne en nlog(n)).
Le processus père doit attendre ses nb_split fils, car sinon la seconde partie du programme (merge) sera exécutée avant que tous les sous-fichiers soient triés. Cela ne permettrait pas la correction de notre algorithme (on doit effectuer le merge sur des fichiers triés, pour obtenir un fichier finalement triés).
Illustrons notre propos grâce à un schéma :

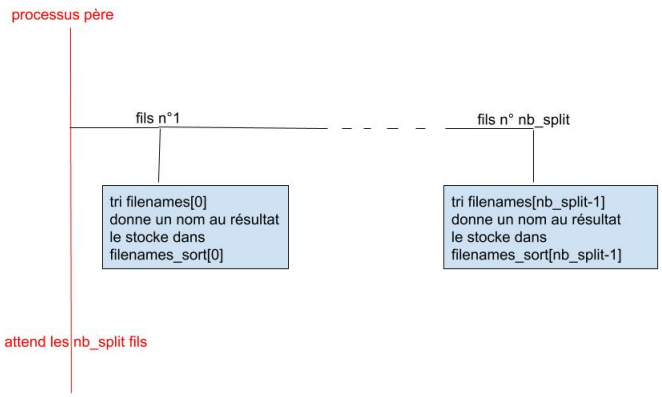

Figure 1: Parallélisation des tris des fichiers avec nb_split processus fils

A l'issu de cette première partie du programme, le tableau filenames_sort est rempli des noms de sous-fichiers de i_file qui sont triés. Les fichiers triés sont stockés dans la mémoire de masse.

5 Une deuxième amélioration: la parallélisation des merges des sous-fichiers triés en deux cascades

Dans le code de base fourni, le tableau **filenames_sort** contient les nb_split fichiers d'entiers qui sont triés. **filenames_sort** est issu de la première partie de l'algorithme (la partie tri interne), ceci est expliqué dans la section précédente.

La seconde amélioration s'intéresse à la seconde partie de l'algorithme : celle ou l'on merge les fichiers triés. En effet, dans le code de base, les merges sont effectués à la suite (en une seule cascade).

Nous avons donc amélioré cette partie du programme en parallélisant la fusion des fichiers en deux cascades au lieu d'une.
Le principe est le suivant : on a un processus père qui crée simplement deux fils. Chacun des fils s'occupe d'une moitié du tableau filesnames_sort. C'est-à-dire qu'ici, chacun des fils effectue des merges à la suite dans leur partie de tableau. En clair, on applique la partie merge de la version 1 du programme sur les deux moitiés du tableau.
Chaque fils aura donc au final un fichier trié à l'issu de sa série de merges. Pour obtenir le résultat final, à savoir un fichier d'entiers triés, il suffit de merge le fichier résultat du fils 1 et le fichier résultat du fils 2.

Figure 2: Illustration sur un exemple du merge à 2 cascades

6 Une troisième amélioration : la parallélisation des merges des sous-fichiers triés en n cascades

Dans la version 2 de l'algorithme, nous avons parallélisé les merges en deux cascades. C'est pourquoi il est naturel de se demander si l'on pourrait pas paralléliser la partie merge du programme en n cascades. Car on peut imaginer que plus l'on parallélise le travail, plus l'on réduit le temps d'exécution. Pour cela, on va utiliser une autre structure de données : l'arbre binaire.

Pour effectuer ce travail, il faut d'abord définir une procédure auxiliaire, que l'on appellera merge_rec. Etudions cette procédure auxiliaire récursive :
Cas de base : on considère une partie du tableau filenames_sort qui est composé de deux ou trois cases.
Cas général : on considère une partie du tableau filenames_sort qui est composé de plus de trois cases.
Hypothèse de récurrence : on sait produire un fichier d'entiers triés sur un tableau de taille inférieure à nb_split.
Étude plus précise :
Cas de base : la partie du tableau filenames_sort traitée par un processus fils ne comporte que deux cases, alors on les merge. Ceci produit un fichier d'entiers trié.
La partie du tableau filenames_sort traité par un processus fils ne comporte que trois cases, alors on les merge en série. Ceci produit un fichier d'entiers trié.
Cas général : le processus courant crée deux processus fils, l'un applique la procédure sur la première moitié de la partie considérée de filenames_sort, et l'autre l'applique sur la seconde partie de cette partie du tableau. Ensuite, on merge les deux fichiers triés produits.

Il était nécessaire d'introduire une procédure auxiliaire, car le résultat du dernier merge que l'on doit effectuer doit être redirigé dans o_file. Donc le caractère récursif du problème proposait de traiter tous les merges (sans le dernier) dans une fonction auxiliaire et le dernier merge dans la fonction principale.

Donc dans la fonction principale nommée projectV3_combMerge, on a un processus père qui créé deux processus fils. Le premier d'entre eux s'occupe de la première moitié du tableau, et le second de la seconde partie de filenames_sort. Les fils vont produire en résultat chacun un fichier d'entiers triés, le processus père s'occupera donc de merger les deux résultats. En sortie, on aura produit un fichier d'entiers triés.

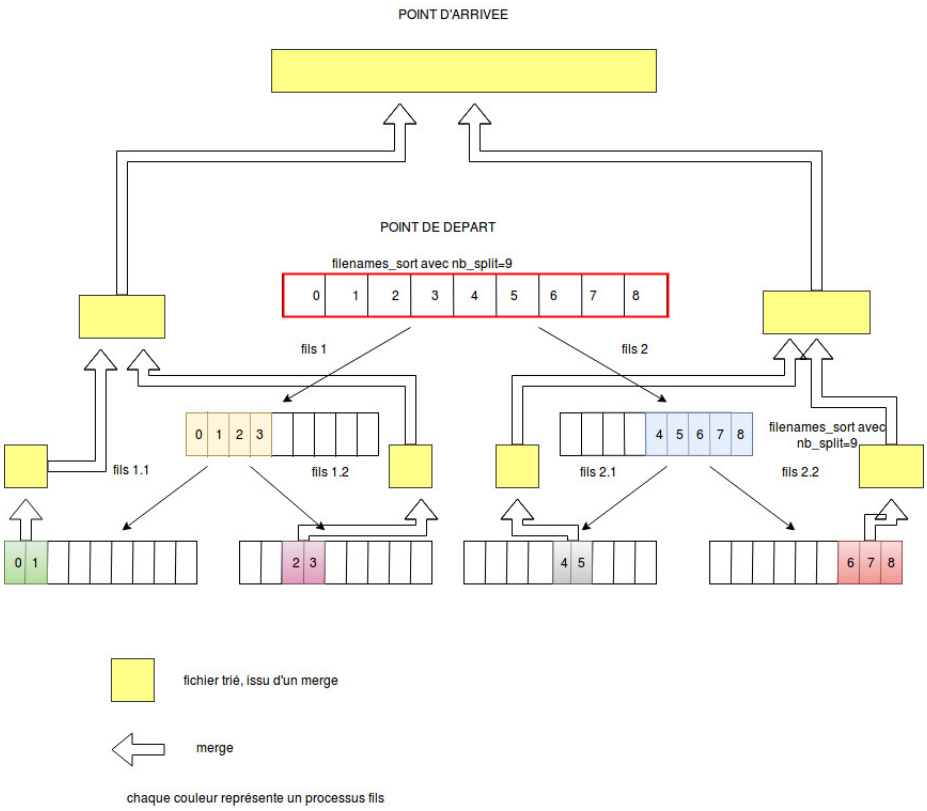

Figure 3: Illustration sur un exemple du merge à n cascades

7 Mesures : comparaison des différentes versions de l'algorithme

Le script Python qui a permis d'automatiser (en partie) les tests effectués sur les différentes versions de l'algorithme (il y a quatre versions, en comptant celle implémentée de base) est fourni avec le code.

Pour réaliser les tests, nous avons d'abord généré un fichier de 50.000 entiers. Sur chacune des versions, nous calculons le temps que cela a pris pour résoudre le problème du tri en fonction du paramètre nb_split. Remarquez sur le script Python, que nous faisons varier nb_split seulement de 1 à 1000. En effet, il serait aberrant de vouloir faire des tests pour nb_split=50.000, parce qu'il y a 32.767 valeurs de PID différentes possibles, or si l'on va au niveau des feuilles de l'arbre binaire pour la version 3 de l'algorithme, on a environ 16.000 processus qui tournent en même temps. En remontant l'arbre, il y a alors environ 32.000 processus qui tournent en même temps. On voit clairement le problème qu'il se

pose pour l'ordinateur (sachant que l'on ne peut déjà sûrement pas faire tourner 32.767 processus en même temps, d'autant plus qu'il y a des tâches de fond qui s'exécutent aussi). De plus, au delà de 1000 splits, les temps nécessaires pour répondre au problème étaient très longs.

Dans un premier temps, traçons un graphe permettant de comparer les différentes versions de l'algorithme de tri externe. Ces différentes améliorations ont été détaillés dans les sections ci-dessus.

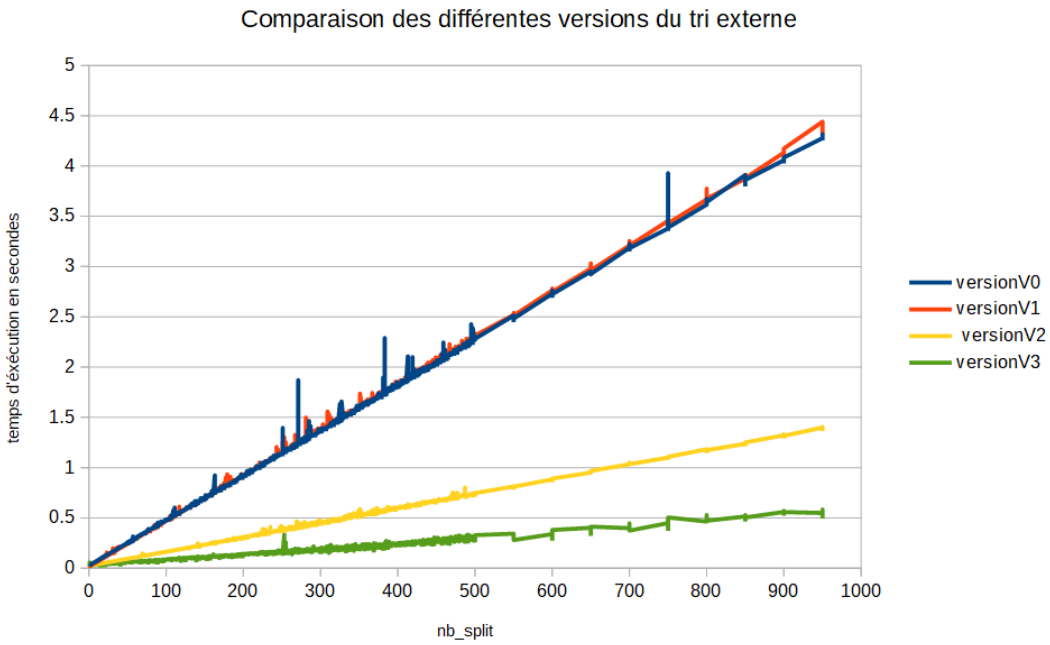

Figure 4: Comparaison des différentes versions du tri externe : temps d'exécution en fonction de nb_split

Étude des données fournies :

On remarque clairement que la version 3 de l'algorithme (celle avec les merges à n cascades) est plus efficace en terme de temps d'exécution quel que soit nb_split. En fait, toutes les améliorations développées semblent, d'après les courbes, meilleures. Sauf la version 1 semble quasiment équivalente à la version 0 implémentée de base. Pour la version 0, on a juste changé la primitive system en exec (cf paragraphe 3).
En observant la figure 4, on remarque que les deux premières versions sont très similaires, on peut donc se dire que paralléliser les tris des sous-fichiers n'est pas ce qui influe le plus sur le résultat. Ce serait donc la fusion de deux fichiers qui

serait la plus coûteuse. Si l'on pense à la fusion de deux fichiers de n entiers, on peut alors se dire que cette opération a une complexité temporelle en O(n), car on parcourt chacun des fichiers. Cependant en disant que la fusion est en O(n), on ne connaît pas la constante tn=Constante*n (tn la complexité temporelle), donc peut être que sur des fichiers de taille inférieure à 50.000 entiers, la constante est trop forte pour que la complexité de la fusion soit inférieure à celle du tri rapide qui est en O(nlog(n)).
En traitant les données numériques issues des tests, on a calculé que la version 3 est en moyenne 16 fois plus rapide que la version 0, avec un écart-type de 7%. Alors que la version 1 est en moyenne

8 Modification du tri interne sur les différentes versions, et mesures

Dans le code de base fourni pour ce projet, différentes fonctions de tri ont été données. Pour les tests effectués dans la section 6, le tri interne utilisé était le quick sort (qsort). On va tester l'efficacité, en terme de temps d'exécution, de toutes les versions avec deux autres tri : le tri par tas (hsort) et le tri par insertion (isort).

Traçons dans un premier temps, les graphes correspondants aux résultats expérimentaux.

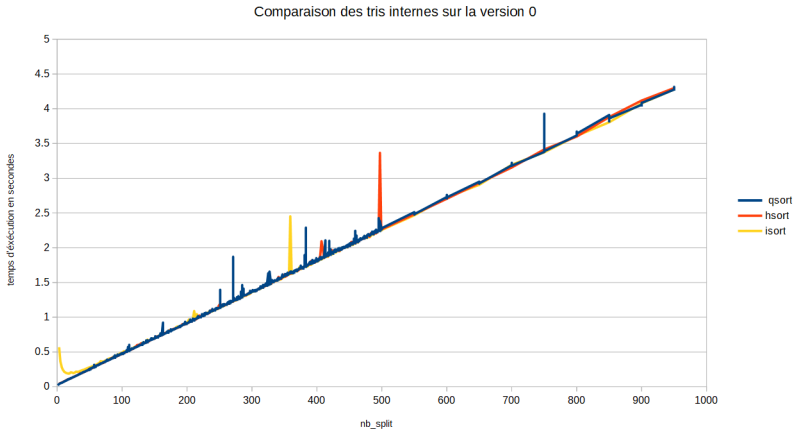

Figure 5: Comparaison des différents tris internes sur la version 0 (celle fournie de base) : temps d'exécution en fonction de nb_split

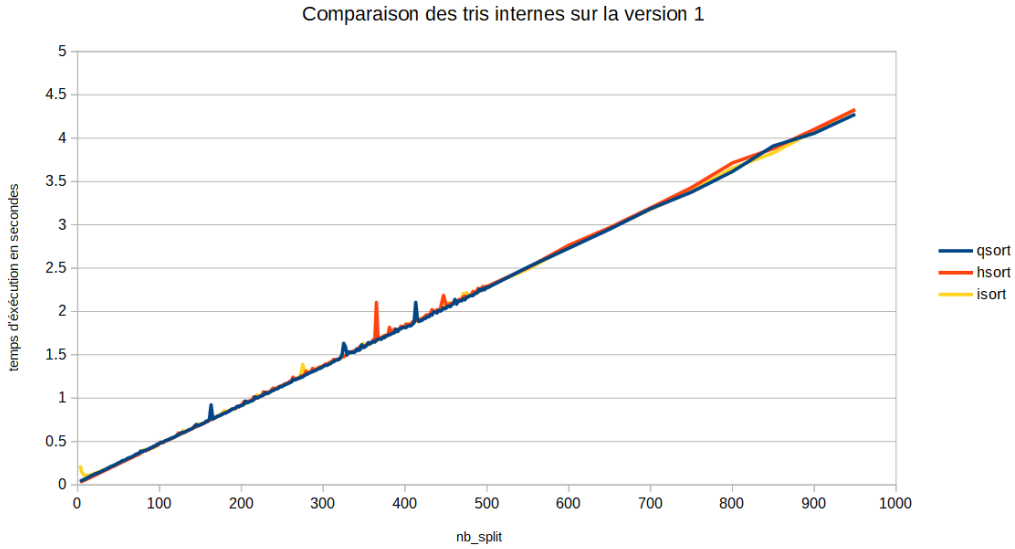

Figure 6: Comparaison des différents tris internes sur la version 1 (parallélisation des tris) : temps d'exécution en fonction de nb_split

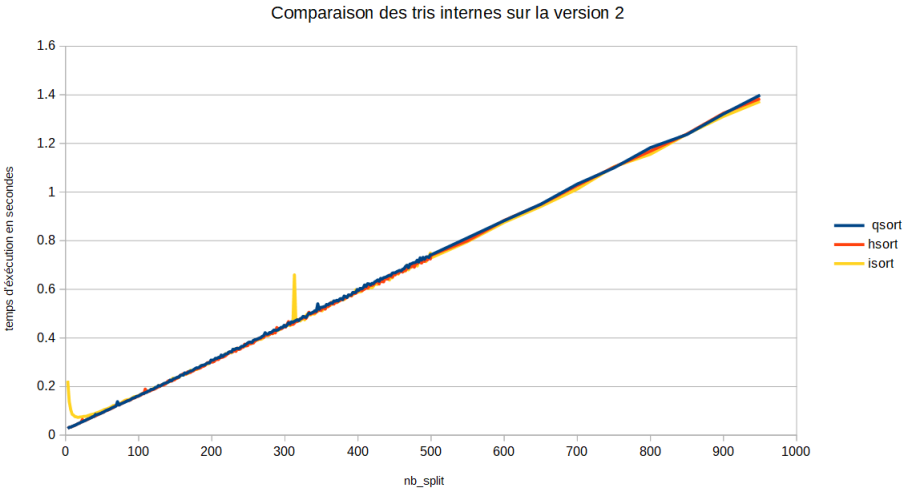

Figure 7: Comparaison des différents tris internes sur la version 2 (paréllélisation des merges à 2 cascades) : temps d'éxécution en fonction de nb_split

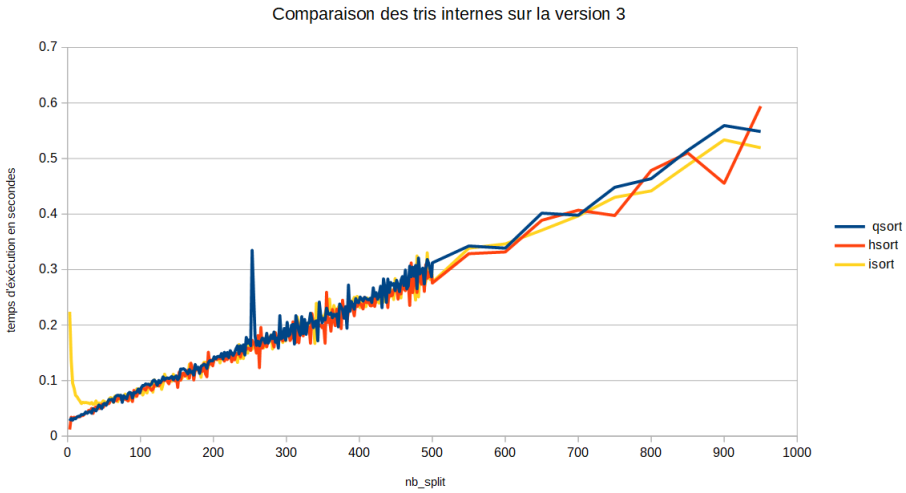

Figure 8: Comparaison des différents tris internes sur la version 3 (paréllélisation des merges en arbre binaire : temps d'éxécution en fonction de nb_split

Étude des graphiques fournis:
Premièrement, on peut remarquer que pour les versions 0, 1 et 2, quelque soit le tri interne utilisé, les résultats sont similaires (pour nb_split supérieur à 20). Les temps d'exécution en fonction du nb_split sont équivalents. En ce qui concerne la version 3, il y a plus de différences, au delà de 500 splits, il y a quelques divergences. Mais les résultats restent proches.

On remarque aussi, que dans chacune des versions, pour environ nb_split inférieur à 20, le tri par insertion est systématiquement moins bon que le tri par tas et que le tri rapide. Cela peut s'expliquer parce que le tri par insertion a une complexité temporelle en O(n^2), ou n est le nombre d'entiers à trier, alors que les deux autres tris la complexité moyenne est en O(nlog(n)). Donc pour un faible nb_split, il y a de grands fichiers à trier, donc ce tri est vraiment moins bon que les deux autres algorithmes de tri.
On remarque aussi, que vers 10 splits, la courbe correspondant au tri par insertion atteint un minimum, c'est à dire, que 10 splits est la configuration optimale pour ce tri. Les autres tris évoluent de manière quasiment linéaire avec nb_split. Le tri par insertion évolue de manière linéaire aussi après avoir passé son minimum.

9 Conclusion

Tout au long de ce projet, nous avons étudié et implémenté des tris externes, car ces derniers sont intéressants lorsque nous avons besoin de trier un grand nombre de données (par exemple un fichier avec beaucoup d'entiers), en effet il permet de ne pas dépasser les capacités de la mémoire vive. Cependant, travailler avec la mémoire de masse diminue l'efficacité en termes de temps d'exécution, car accéder à la mémoire en dur est plus long. C'est pourquoi nous avons utilisé la primitive Unix fork pour paralléliser les tâches. Premièrement, nous avons simplement parallélisé les tris des sous-fichiers, mais ça n'a pas apporté une réelle amélioration. Ensuite, nous avons parallélisé les merges des sous-fichiers triés, d'abord en deux cascades, puis en n cascades (structure d'arbre binaire). On remarque une très nette amélioration dans ces deux cas, ce qui est normal, car la fusion est plus coûteuse. Enfin, nous avons tenté de connaître l'influence qu'avait le choix du tri sur l'efficacité de note algorithme. De manière générale, on peut dire que ce choix n'a pas de réelle influence, pour un nb_split supérieur à 10 et nb_split inférieur à 1000.