Biau-Guilhembet Marie

Il existe différentes manières de trier des données. Parmi les algorithmes de tri existants, il y a le tri rapide (quicksort), le tri par tas (heapsort) et le tri par insertion (insertion sort). De plus, les performances d'un programme de tri dépendent de l'approche d'accès aux données choisie et de l'utilisation ou non du parallélisme. Cela nous amène à nous poser la question suivante : Quel est l'impact de la méthode de tri utilisée sur la vitesse du programme ? Pour répondre à cette question, nous allons réaliser une suite d'expériences sur l'ordinateur LIPSKY-b-06 situé en salle 101D à l'ENSSAT. Il s'agit d'un ordinateur HP équipé du système d'exploitation Ubuntu et possédant 4 processeurs Intel Core i5. Dans un premier temps, nous allons présenter les différents versions implémentées du programme de tri externe. Elles serviront de base à notre expérimentation. Ensuite, nous étudierons l'influence de l'algorithme de tri employé par le programme. Pour terminer nous comparerons les différentes méthodes et synthétiserons l'impact des différents paramètres.

I. Les différentes versions du programme de tri externe

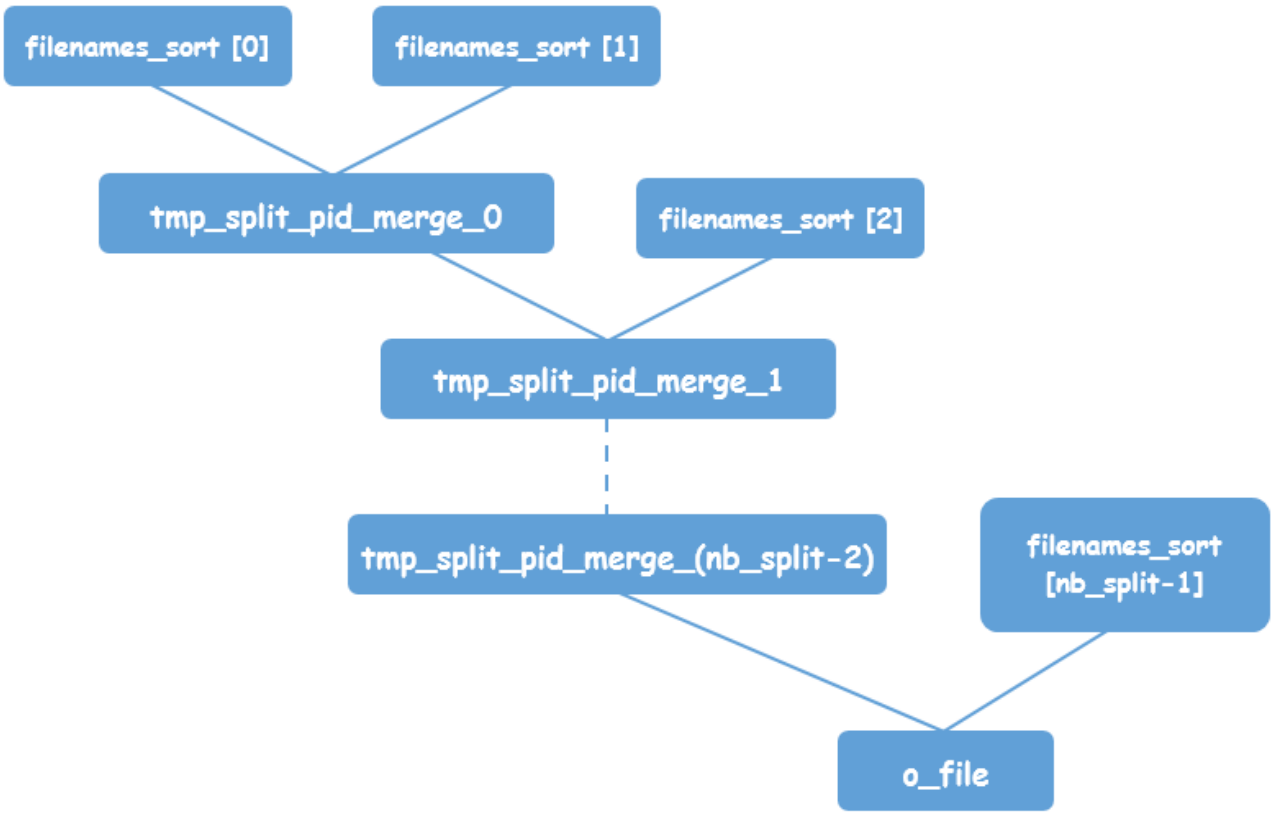
La primitive system() introduit des failles de sécurité. C'est pourquoi on lui préfère les primitives de la famille exec() qui emploient le recouvrement. Dans le fichier system_utilis_etu.c se trouve le code de la méthode SU_removeFile n'utilisant pas system mais execl. On utilise execl car elle prend en paramètre le chemin courant, la liste des arguments et utilise l'environnement courant. Le processus appelant est alors complètement remplacé, c'est pourquoi on doit utiliser un fork.

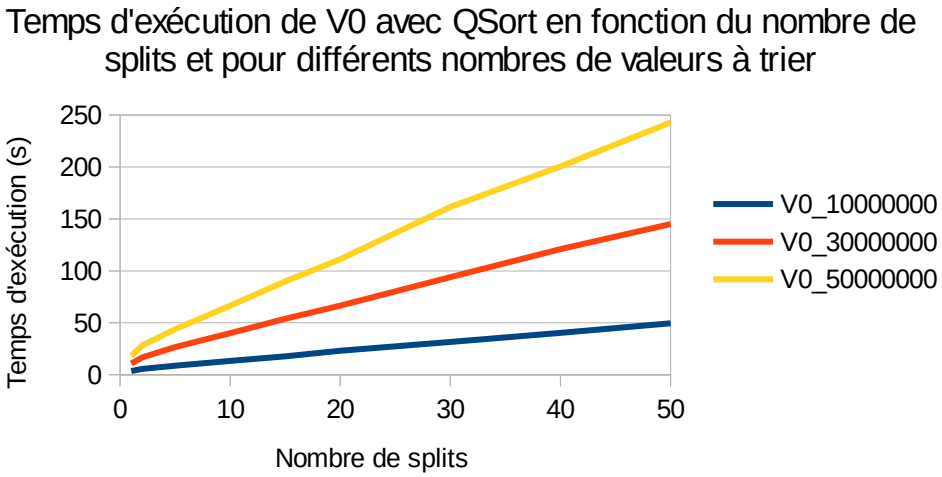
1. ProjectV0

ProjectV0 prend en paramètre le fichier d'entrée, le fichier de sortie et le nombre de splits souhaités. Le fichier d'entrée est divisé puis chaque sous-fichier est trié l'un après l'autre grâce à la procédure projectV0_sortFiles puis les fichiers sont fusionnés les uns après les autres grâce à la procédure projectV0_combMerge.

Schéma de fonctionnement de projectV0_sortFiles

Schéma de fonctionnement de projectV0_combMerge



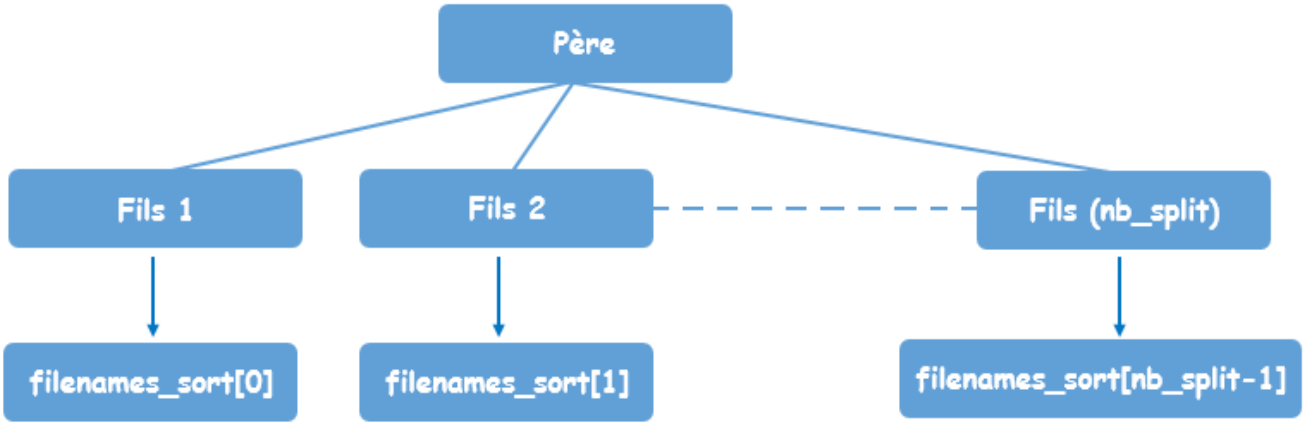


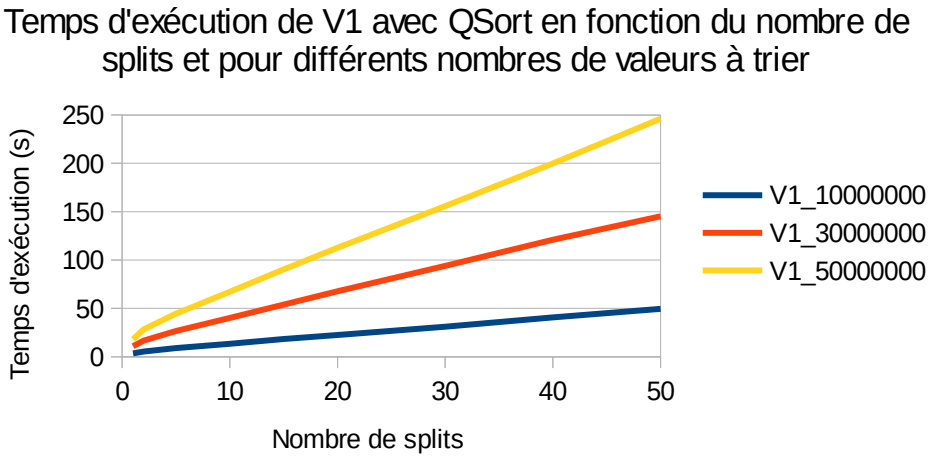
Le graphique ci-dessus montre que le temps d'exécution augmente linéairement avec le nombre de splits. Plus le fichier en entrée est grand, plus le temps d'exécution est élevé.

2. ProjectV1

Le mode projectV1 fonctionne similairement à projectV0 mais parallélise le tri des différents sous-fichiers grâce à la commande fork.

Arbre des processus et schéma de fonctionnement de projectV1_sortFiles



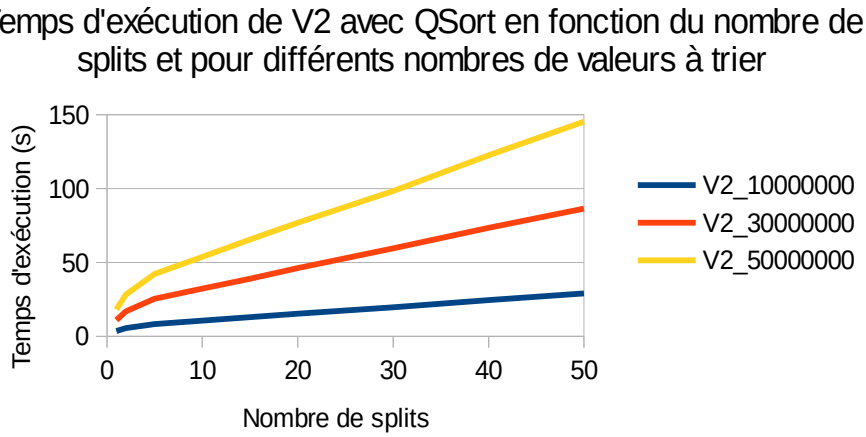


Le graphique ci-dessus montre que le temps d'exécution augmente linéairement avec le nombre de splits. Plus le fichier en entrée est grand, plus le temps d'exécution est élevé. Malgré la parallélisation du tri des sous-fichiers, on ne note pas de différence significative avec V0. Cela peut être dû au nombre de splits ainsi qu'à la taille du fichier d'entrée et varier d'un ordinateur à l'autre.

3. ProjectV2

Le mode projectV2, en plus de ce qui a été réalisé dans projectV1, parallélise la fusion des sous-fichiers en deux cascades au lieu d'une grâce à la commande fork.

Arbre des processus et schéma de fonctionnement de projectV2_combMerge

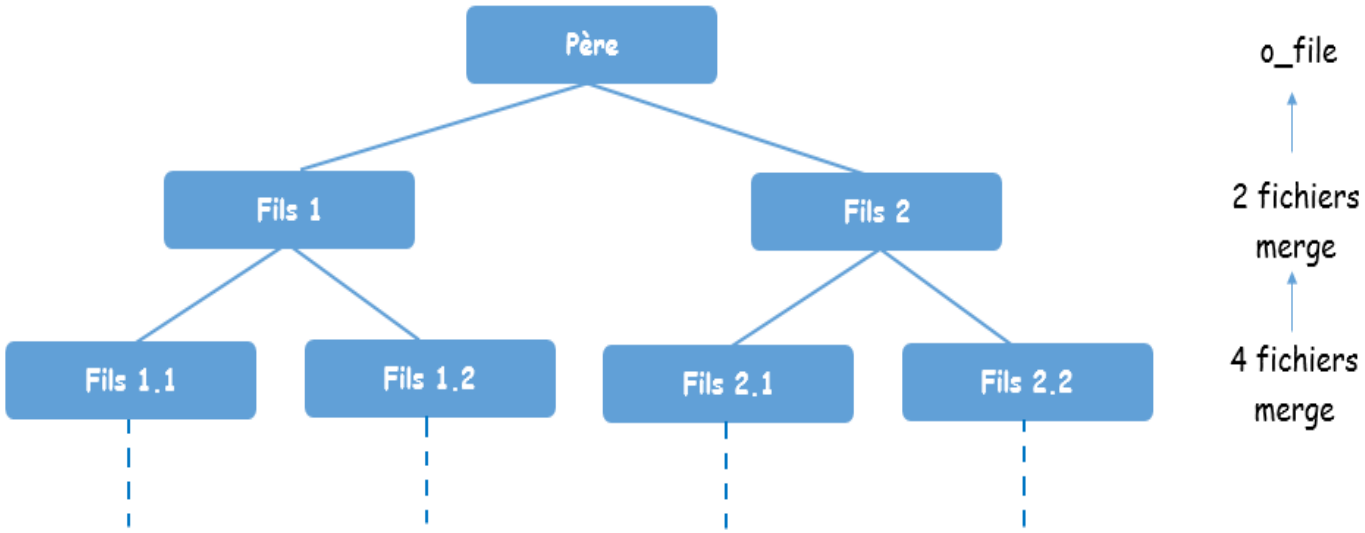


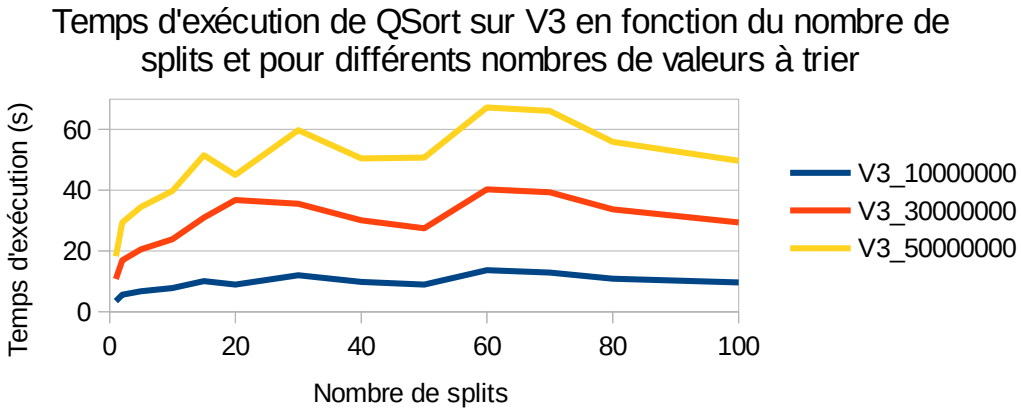
Le graphique ci-dessus montre que le temps d'exécution augmente linéairement avec le nombre de splits. Plus le fichier en entrée est grand, plus le temps d'exécution est élevé. La parallélisation de la fusion des sous-fichiers en deux cascades au lieu d'une permet de réduire notablement le temps d'exécution pour un nombre de splits suffisant. Cette différence s'accroît lorsque le nombre de splits augmente.

4. ProjectV3

Le mode projectV3, en plus de ce qui a été réalisé dans projectV1, parallélise la fusion des sous-fichiers en utilisant un arbre binaire grâce à la commande fork et à l'utilisation de la récursivité.

Arbre des processus et schéma de fonctionnement de projectV3_combMerge



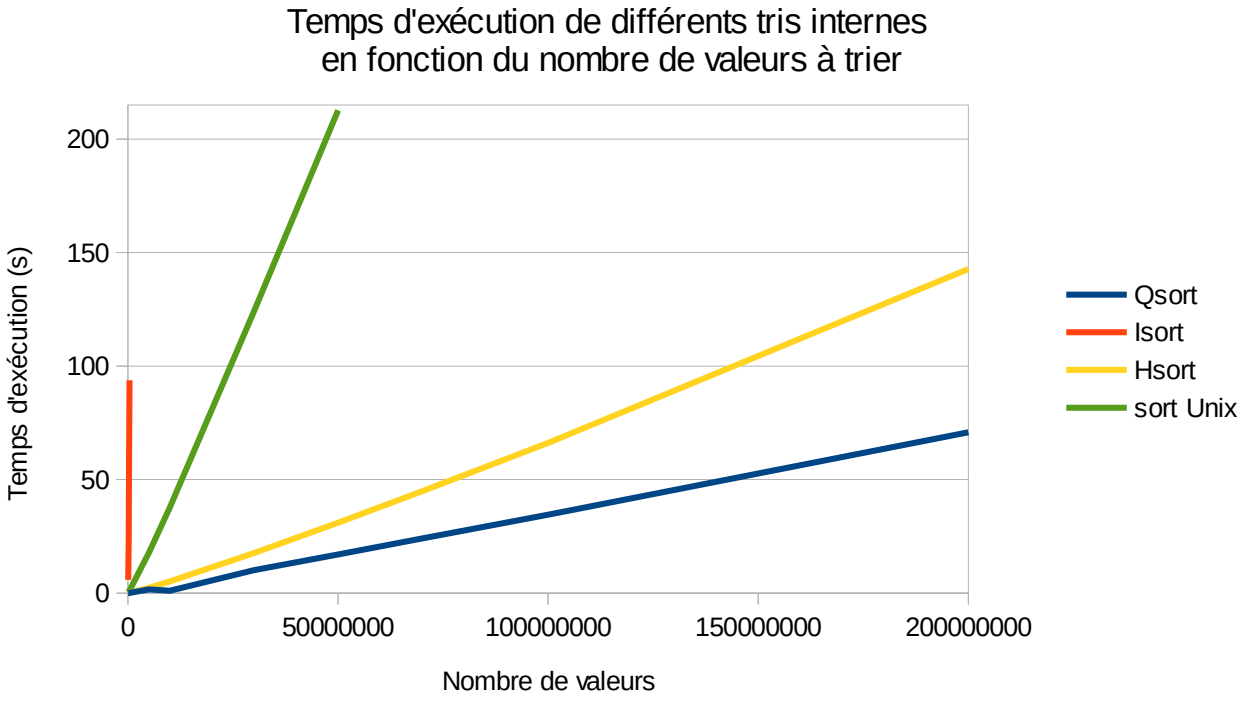


Le graphique ci-dessus montre que plus le fichier en entrée est grand, plus le temps d'exécution est élevé. Le temps d'exécution est quasi-proportionnel à la taille du fichier d'entrée. Pour un nombre de splits inférieur à 15, il augmente avec le nombre de splits. Ensuite, il fluctue tout en gardant des valeurs faibles. Cela s'explique par le fait que la hauteur et l'équilibre de l'arbre binaire créé dépendent du nombre de splits. Ainsi, si une branche possède plus de sous-fichiers à trier que sa branche sœur, alors le père devra attendre la branche la plus longue, même si la branche la plus rapide a déjà terminé son travail. La parallélisation de la fusion des sous-fichiers en arbre binaire permet de limiter le temps d'exécution de l'algorithme.

II. Les différents algorithmes de tri

1. Tri interne

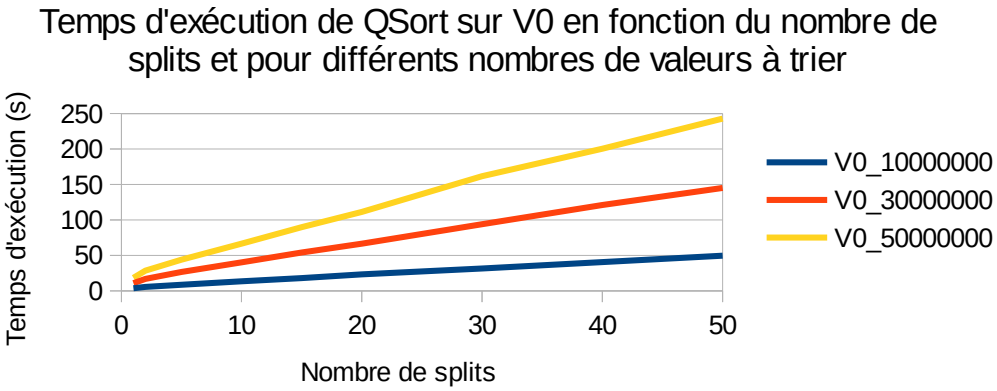
Le tri interne est un tri qui s'effectue entièrement en mémoire centrale. Le temps d'exécution d'un tri interne dépend de l'algorithme de tri utilisé. Pour tester les tris rapide, par tas et par insertion, on utilise le mode demoSort proposé par le projet.

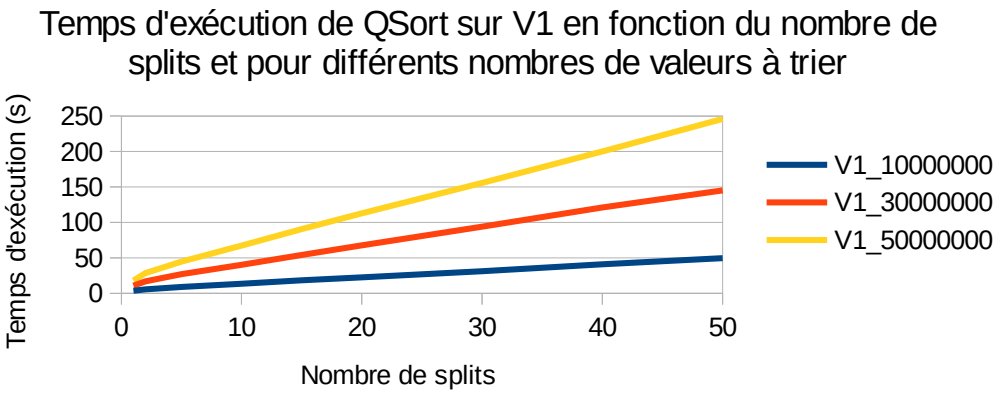


Le tri rapide est le plus rapide des tris testés, sa complexité théorique moyenne est en O(nlog(n)). Vient ensuite le tri par tas qui possède également une complexité théorique moyenne en O(nlog(n)). La commande sort fournie par Unix, présente des temps d'exécution supérieurs à ceux des deux algorithmes de tri précédents. En effet, cette commande permet de trier toute sorte de données, ce qui explique qu'elle mette plus de temps à s'effectuer. Le plus lent des tris testés est le tri par insertion, sa complexité théorique moyenne est en $O(n^2)$.

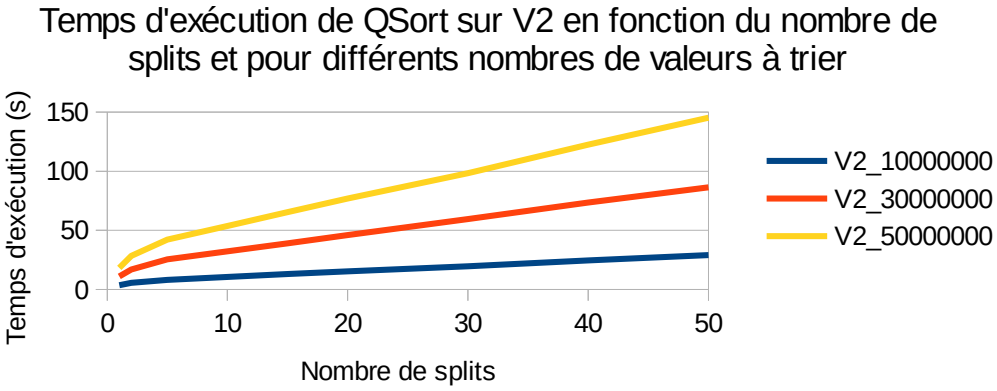
2. Tri rapide

Le tri rapide est une méthode qui repose sur le principe « diviser pour régner ». Une valeur est choisie comme pivot et les éléments plus petits que le pivot sont dissociés, par échanges successifs, des éléments plus grands que le pivot. Chacun de ces deux sous-ensembles est ensuite trié de la même manière.

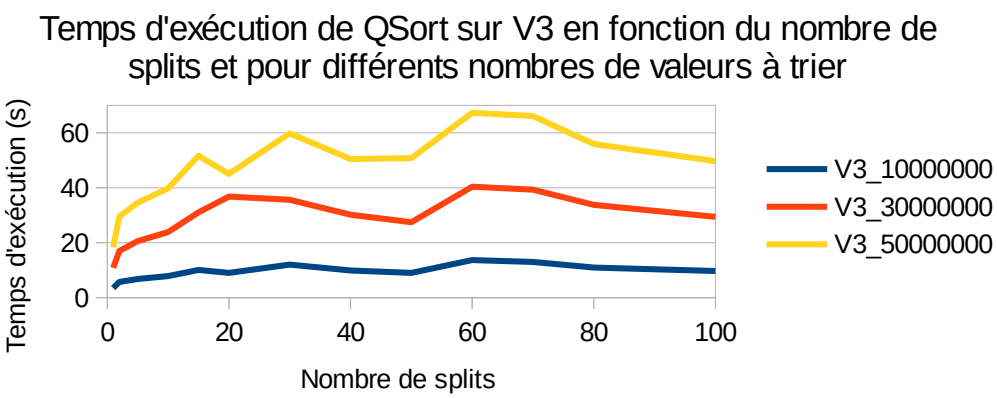




Les graphiques montrent que le temps d'exécution augmente linéairement avec le nombre de valeurs à trier et linéarithmiquement (O(nlog(n))) avec le nombre de splits pour V0 et V1.



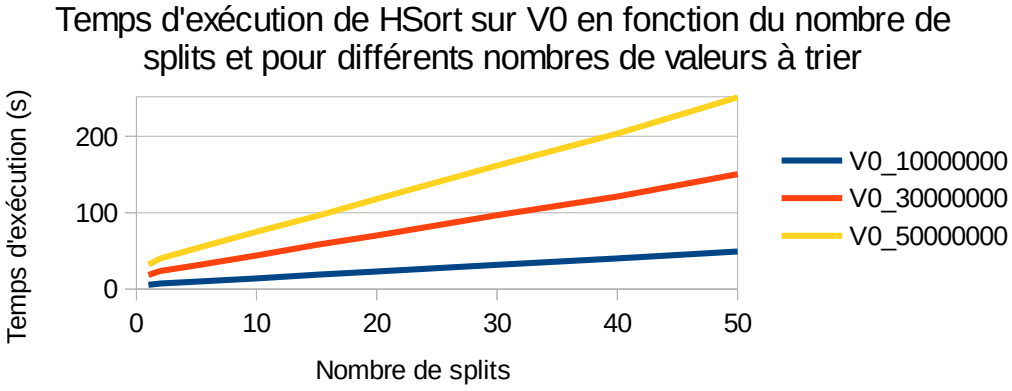
V2 propose des temps d'exécution similaires à V0 et V1 pour un nombre de splits faible. La différence s'accroît lorsque le nombre de splits augmente : V2 est plus rapide.



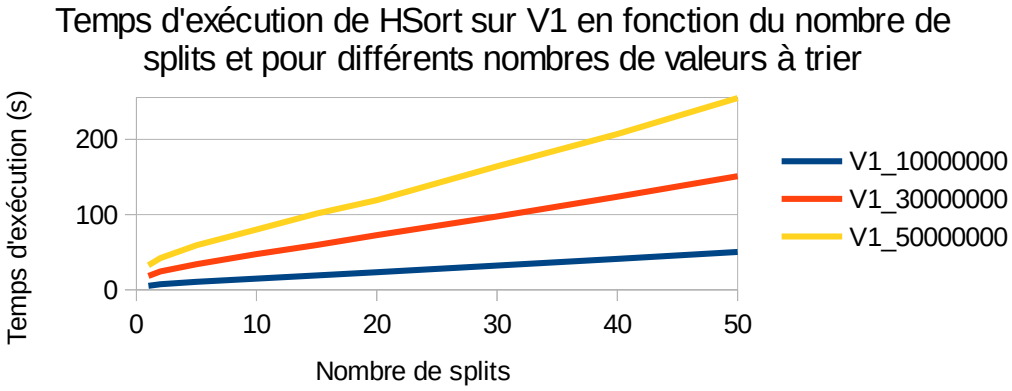
Pour V3, le temps d'exécution est quasi-proportionnel au nombre de valeurs à trier. Pour un nombre de splits inférieur à 15, il augmente avec le nombre de splits. Ensuite, il fluctue tout en gardant des valeurs faibles.

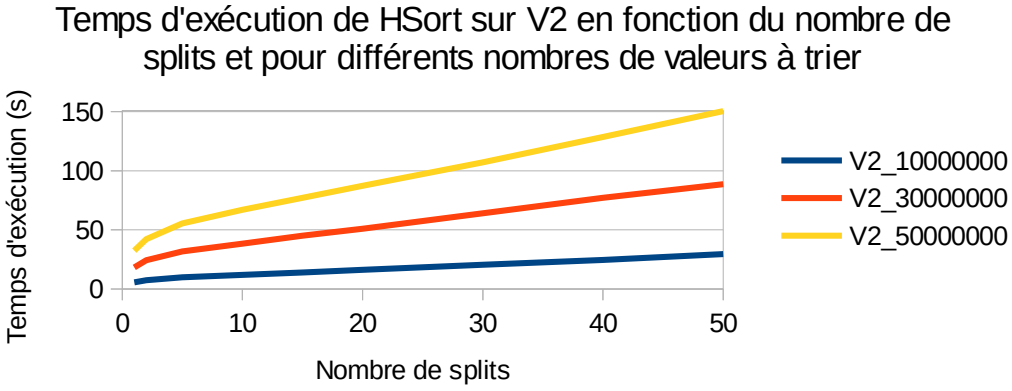
3. Tri par tas

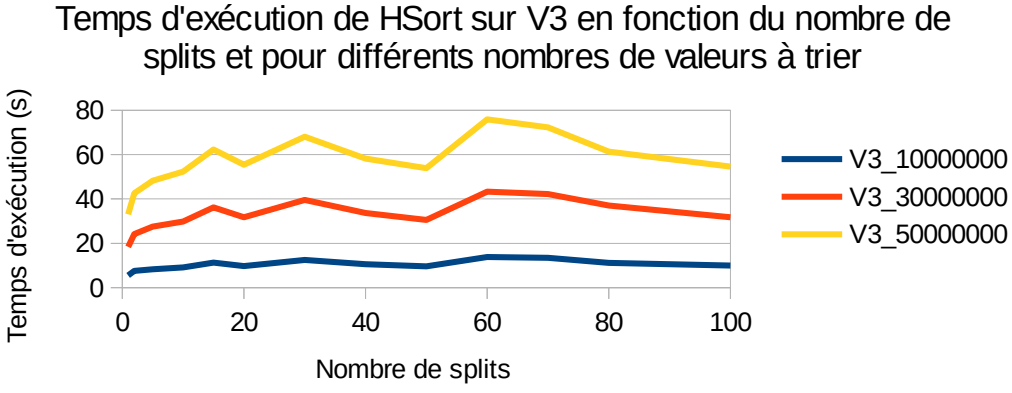
Le tri par tas est une méthode qui consiste, à chaque itération, à identifier le plus petit des éléments qui ne sont pas encore triés, et à l'échanger avec le premier de ceux-ci. L'algorithme utilise une structure de tas, souvent implémentée au moyen d'un tableau. Il est intéressant d'utiliser ce tri si l'on soupçonne que les données à trier seront souvent des cas quadratiques pour le tri rapide.



On observe les mêmes phénomènes que pour le tri rapide mais avec des temps d'exécution légèrement plus élevés.

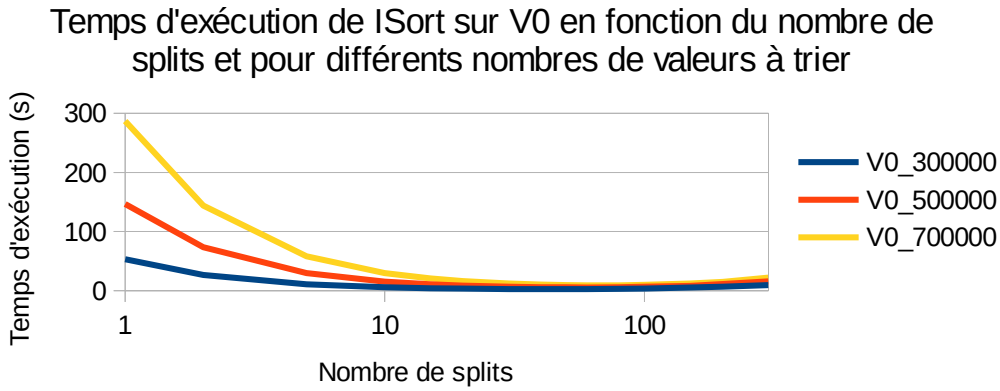


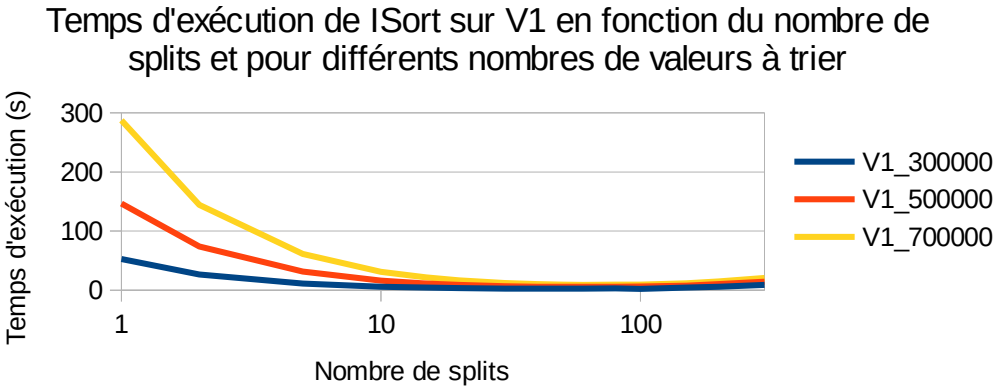




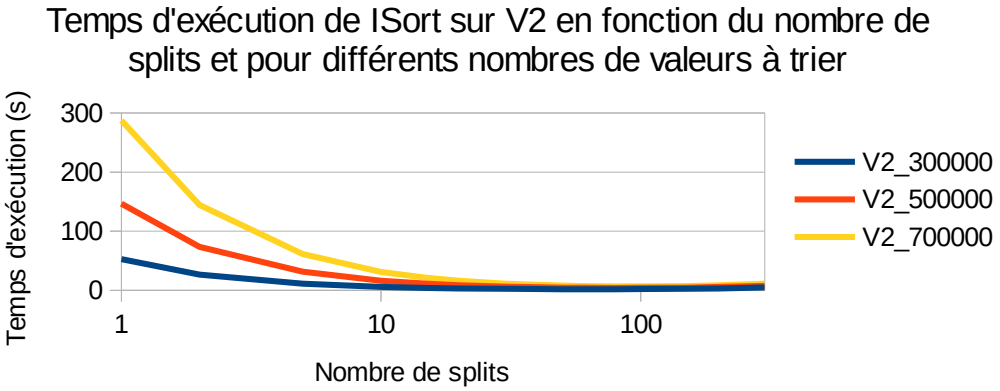
4. Tri par insertion

Le tri par insertion est une méthode reposant sur le principe suivant. Les valeurs sont insérées les unes après les autres dans une liste triée (initialement vide). C'est souvent le plus rapide et le plus utilisé pour trier des entrées de petite taille. Il est également efficace pour des entrées déjà presque triées.

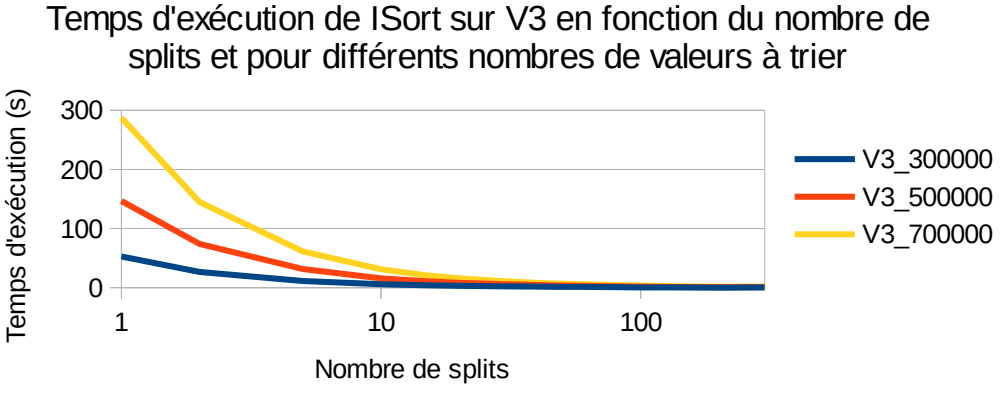




Les graphiques montrent que le temps d'exécution augmente linéarithmiquement avec le nombre de valeurs à trier et diminue selon une exponentielle décroissante avec le nombre de splits pour V0 et V1. Pour un nombre de splits supérieur à 50, le temps d'exécution augmente à nouveau.



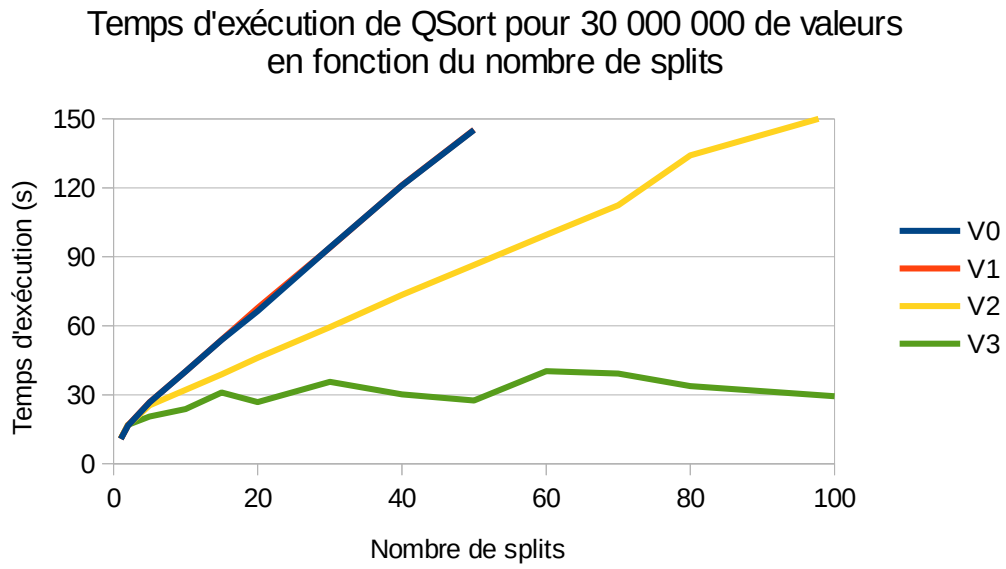
V2 propose des temps d'exécution similaires à V0 et V1 pour un nombre de splits faible. La différence s'accroît lorsque le nombre de splits augmente : V2 est plus rapide.



V3 est similaire à V2 mais avec des temps d'exécution de plus en plus faibles lorsque le nombre de splits augmente. Le temps d'exécution ne recommence à augmenter qu'à partir de 200 splits.

III. Comparaison des différentes méthodes

1. Impact des stratégies de tri et de fusion des sous-fichiers



La comparaison des quatre versions du projet, pour 30 000 000 de valeurs et en utilisant le tri rapide, permet de conclure sur l'efficacité des implémentations. Les versions V0 et V1 possèdent des temps d'exécution similaires, le tri des sous-fichiers présentant un temps faible devant celui de la fusion des sous-fichiers triés. La version V2 est plus rapide que les précédentes, grâce à la fusion des sous-fichiers en deux cascades au lieu d'une. La version V3 du projet est de loin la plus rapide car elle maximise le parallélisme en employant un arbre binaire. Cependant, elle présente des irrégularités selon le nombre de splits employés.

2. Impact de l'algorithme de tri

L'algorithme de tri le plus efficace est le tri rapide, que ce soit en l'utilisant simplement dans le cadre d'un tri interne ou bien en l'utilisant dans les différents projets de tri externe. Vient ensuite le tri par tas puis le tri par insertion. Le tri par insertion, pour pouvoir être exécuté dans un las de temps convenable, ne doit prendre en entrée qu'un faible nombre de valeurs comparé aux deux autres algorithmes de tri.

Le tri externe, contrairement au tri interne, est un algorithme qui utilise la mémoire de masse pour stocker les résultats partiels de l'algorithme de tri. Certains algorithmes de tri, à la complexité non linéaire, empêchent d'utiliser des algorithmes travaillant en flux. Le tri externe est utile pour trier des volumes de données trop importants pour pouvoir tenir en mémoire centrale, ou lorsque la mémoire centrale de l'utilisateur est limitée ou bien lorsque l'on veut minimiser l'empreinte en mémoire vive.

Effectuer un tri interne s'avère plus rapide pour le tri rapide et le tri par tas, quelque soit le nombre de splits employé. Le tri externe, pour un nombre de splits supérieur à 1, est plus rapide pour le tri par insertion. En effet, cet algorithme est adapté aux entrées de petite taille et presque triées.

Le tri fusion, dont la complexité temporelle est en O(nlog(n)), nécessite un tableau intermédiaire de même taille que le nombre d'éléments à trier. Lorsque l'espace mémoire est restreinte, la complexité spatiale en O(n) de ce tri peut poser problème au-delà d'une certaine taille de sous-fichiers. Il faut donc tenir compte à la fois de la complexité temporelle et de la complexité spatiale des algorithmes de tri utilisés.

L'utilisation de la primitive fork permet de réduire considérablement le temps d'exécution de certains algorithmes. Lorsque plusieurs tâches doivent être réalisées par celui-ci, l'implémentation la plus intéressante parmi celles testées (si elle se prête au cas étudié) est l'arbre binaire. En effet, il faut essayer de maximiser le parallélisme lorsqu'on fait le choix de l'utiliser. Cela permet également de limiter l'impact en mémoire vive, afin de pouvoir effectuer plusieurs processus simultanément. Beaucoup d'algorithmes de tri existent, mais certains sont bien plus utilisés que d'autres en pratique. Le tri par insertion est souvent plébiscité pour des données de petite taille, tandis que des algorithmes asymptotiquement efficaces, comme le tri rapide ou le tri par tas, seront utilisés pour des données de plus grande taille. Le parallélisme est d'autant plus efficace que l'algorithme de tri est optimisé pour des données de petite taille et déjà triées. Cependant, si l'utilisation du parallélisme n'est pas adaptée, elle peut ralentir l'algorithme. La primitive fork n'est donc pas à employer dans toutes les situations.