Rapport final de projet

Implémentation et évaluation d'algorithmes de tri efficaces

UE : PSTL

Groupe

PINTO Bruno

DUTRA Enzo

Encadrants

JOURNAULT Matthieu

PEPIN Martin

Introduction 3
I - Rappel sur l'architecture des processeurs 4
A - Notion de cache 4
B - Prédicteur de branchement 4
II - Tri de données 6
A - L'entropie 6
B - Timsort et Algorithme K-aware 7
C - Algorithmes implémentés 9
1 - InsertionSort 9
2 - TimSort 9
3 - AdaptiveShiverSort 9
D – Implémentation 10
1 - Tri de petit tableau 10
2 - Détection et incrémentation des runs 10
3 - Structure de la pile 12
4 - Condition de fusion de l'AdaptativeShiverSort 12
5 - Fusion de run 12
6 - Structure du Code 12
III - Benchmark 13
A - Méthodes de benchmark 13
B - Génération de tableaux 14
1 - Description des différentes méthodes 14
i - Génération de tableaux complètement aléatoires 14
ii - Génération de tableaux avec runs de taille constante 14
iii - Génération de tableaux avec runs de taille aléatoire 15
iv - Génération de tableaux avec runs de taille semi aléatoire contrôlé par un paramètre delta 16
v - Génération de tableaux ayant une entropie donnée 16
2 - Génération d'un tableau à entropie donnée 17
C - Résultats des tests 18
Discussion sur les résultats 22
Conclusion 23
Bibliographie 24

Introduction

Le problème du tri de données est l'un des problèmes les plus anciens et les plus étudiés dans le domaine de l'informatique. L'utilisation d'algorithmes de tri étant bien souvent un des outils de base dans la conception de nombreux algorithmes. Depuis les années 1940, de nombreux algorithmes de tri ont été élaborés comme le tri rapide [4], le tri par fusion [3] ou bien encore le tri par tas [8].

Plus récemment des algorithmes basés sur l'exploitation de l'architecture moderne des processeurs ainsi que de l'entropie des données à trier tel le TimSort [7], PowerSort [6] ou le très récent AdaptiveShiversSort [5] ont vu le jour.

L'apparition du très célèbre TimSort [7] en 2002, qui sera adopté par la librairies standards python et java (pour les types non primitifs) a été un véritable renouveau pour l'élaboration d'algorithmes de tri, cet algorithme de tri est extrêmement efficace, mettant à profit l'architecture moderne des processeurs ainsi que les structures de données partiellement triées correspondant aux données triées en pratique. Plusieurs algorithmes basés sur le même principe ont vu le jour, notamment l'Adaptative ShiverSort proposé il y a peu (2018) par Vincent Jugé [5]. La preuve de complexité du TimSort en O(n log(n)) remonte à seulement 2015, en 2018 la complexité a été raffiné à O(n + nH) avec H l'entropie du tableau à trier (voir I-c), qui dans certain cas (l'extrême majorité) est meilleur que O(n log(n)).

Les algorithmes étant très récent, nous avons très peu de mesure précise en pratique de la relation performance / entropie. L'idée du projet est donc d'implémenter, dans un langage de bas niveau (C++), divers algorithmes de tri afin d'en évaluer l'efficacité et d'en analyser son comportement. L'objectif étant, dans nos benchmarks, de traiter un maximum de cas possibles pour en extraire des résultats pertinents.

Notre implémentation est disponible sur GitHub, ainsi qu'un script (whole_execution.bash) permettant de lancer tous les bench et afficher tous les résultats de test sous forme de graphiques, à cette adresse :

https://github.com/blackorbit1/PSTL-Tri-optimise

I - Rappel sur l'architecture des processeurs

Le TimSort est extrêmement efficace de par son utilisation très astucieuse de l'architecture moderne des processeurs, notamment la présence de mémoire cache ainsi que le prédicteur de branchement. C'est une des raisons pour laquelle le TimSort est récent, sur un processeur de 1970 n'ayant pas de mémoire cache les algorithmes types QuickSort restaient extrêmement pertinents.

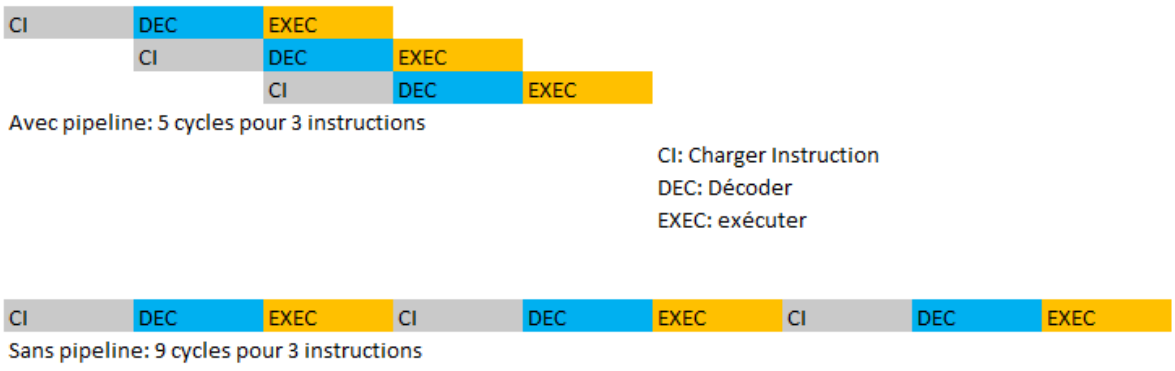
A - Notion de cache

Depuis le début des années 1970, la puissance de calcul des processeurs ainsi que la taille de la mémoire ont drastiquement augmenté, bien plus que la vitesse à laquelle ils sont capables de communiquer. À tel point que le temps de communication entre ces deux organes est devenu un facteur très important dans le temps d'exécution d'un programme.

À partir de ce postulat, les concepteurs de processeurs ont introduit la notion de mémoire cache. L'idée étant que le processeur va dorénavant avoir un peu de mémoire (au plus quelques ko) accessible très rapidement ce qui va lui permettre d'être très efficace sur des éléments très proches en mémoire, et va donc rendre les structures de données contiguës telles que les tableaux (structure statique) et vecteurs (structure dynamique) très efficaces lors de l'accès de cases adjacentes.

B - Prédicteur de branchement

Lors de la lecture d'une instruction assembleur, le processeur doit effectuer plusieurs opérations (nous nous limiterons à un schéma très simplifié) : charger l'instruction en mémoire, la décoder puis l'exécuter. Les processeurs modernes font ce que l'on appelle un *pipeline*, l'idée étant que chaque instruction ait un coup amorti d'un cycle élémentaire.



Or, un des cas posant grandement problème pour le pipeline est le saut conditionnel, en effet au moment où l'on décode l'action A_n (où A_n est un saut conditionnel), on ne sait pas encore quelle sera l'action A_{n+1}. Pour éviter d'avoir des moments où le processeur ne fait "rien", le processeur va commencer à décoder une des instructions suivant le saut.

Dans ce cas-là, il y a deux cas possibles : premièrement, l'instruction décodée correspondait au branchement, dans ce cas tout va bien et on continue le programme, sinon il faut annuler les calculs inutiles et décoder le bon branchement cette fois.

Pour éviter au maximum de se tromper, le processeur va se "souvenir" des dernières exécutions des sauts.

```
set i = 0
while i < 100 do
    // foo
    set i = i + 1
end
```

Dans ce cas, on voit très facilement que le saut engendré par *i < 100* va être en général pris et que donc il peut continuer de décoder *foo*.

Toutes ces notions d'architecture moderne des processeurs sont essentielles pour comprendre l'efficacité du Timsort.

II - Tri de données

A - L'entropie

Pour connaître le "taux de dégradation" dans des données, on utilise usuellement l'entropie de Shannon. Celle-ci permet de savoir combien de bits sont nécessaires pour chaque composante d'une information (lettre, pixels, etc.) afin de la restituer complètement en fonction de la "qualité" de la transmission.

Cette mesure a été formalisée en 1948 par Claude Shannon afin de mieux analyser la transmission de données avec perte (ou bruit parasite).

On peut utiliser cette notion d'entropie dans le cas de tableaux non triés où l'entropie qui permet de calculer le "taux de dégradation" du signal permettra de traduire un "taux de désordre" par rapport au même tableau complètement triée.

On va utiliser, pour ce calcul, la présence de ce qu'on appellera des "runs" qui correspondent à des portions de tableaux déjà triées (de manière strictement croissante ou décroissante).

Par exemple :

12, 7, 6, 5, 4, 3, 1, 0, 0, 7, 14, 36, 37, 42, 73, 3, 3, 5, 21, 21, 21, 24

1e run décroissant	*2e run croissant*	*3e run croissant*

La définition formelle de l'entropie de Shannon est la suivante :

$$H(X) = -\sum_{i=1}^{p} P_i \log_2(P_i)$$

Avec H l'entropie de X le tableau, et où P_i correspond à la probabilité de tomber sur le ième run pour un élément choisi uniformément au hasard parmi les n éléments du tableau.

Cette probabilité correspond à $\frac{taille\ i^{ème}\ run}{taille\ tableau}$

Cette formule correspond à un réel entre 0 et $\log_2(n)$ avec n la taille de le tableau à trier qui traduit donc à quel point le tableau est déjà triée (plus le réel est proche de zéro, plus le tableau est déjà triée).

B - Timsort et Algorithme K-aware

L'un des grands principes du TimSort (et des algorithmes qui s'en inspirent) est d'exploiter les parties déjà partiellement triées, pour cela, l'algorithme se base sur la notion de runs.

Les runs décroissants sont inversés puis tous les runs sont stockés dans une pile R.

À ce stade nous avons donc une pile de runs toutes croissants.

L'idée est maintenant de fusionner des runs deux à deux selon certaines règles. Vu que l'on fusionne deux à deux tous les runs triés jusqu'à ne plus avoir qu'un seul run, le coût en fusion correspond à la somme des membres fusionner, par exemple, si nous avons 3 runs de taille 7, 10 et 12. On peut d'abord fusionner 7 et 10 pour avoir 17 puis 17 et 12 et nous avons donc un coût de fusion de (7 + 10) + (17 + 12) = 46 mais on peut aussi fusionner 10 et 12 pour avoir 22 puis 22 et 7 et donc un coût de fusion de (10 + 12) + (22 + 7) = 51. On peut donc dire que le premier cas est bien meilleur. L'idée est donc de trouver un algorithme offrant un coût de fusion le plus faible possible à l'image de ce que l'on ferait avec Huffman.

Pseudo code du MinimalSort :

```
Entrée: T le tableau à trier
Résultat: T est trié
soit runs = la décomposition en run de T
tant que runs contient au moins deux éléments
    faire fusionner les deux plus petits run de runs dans runs
fin tant que
```

Pseudo code des algos types TimSort :

```
Entrée: T le tableau à trier
Résultat: T est trié
Note: Soit h la taille de S et Ri le ième élément en tête de S, on note Ri la
taille de ri.

soit runs = la décomposition en run de T
soit S = PileVide()
tant que runs n'est pas vide faire
    extraire une run R de runs et empiler R dans S
    tant que vrai faire
        // condition de fusion
    fin tant que
fin tant que
tant que h > 2 faire
    fusionner les runs R1 et R2
fin tant que
```

Vous pouvez remarquer que nous n'avons pas précisé sous quelles conditions les fusions entre runs sont effectuées, en effet, ces conditions vont varier entre chaque algorithme, par exemple ; la première implémentation du TimSort avait comme condition

```
si h >= 3 et r1 + r2 >= r3 faire fusionner R1 et R2
sinon si h >= 2 et r1 >= r2 faire fusionner R1 et R2
sinon sortir de la boucle
```

On dit que cette version du TimSort est 3-aware car l'algorithme s'autorise à observer les 3 éléments en tête de la pile S (R_1, R_2 et R_3). Le MinimalSort, lui, garantit un coût de fusion minimal mais en pratique n'est pas efficace. Notamment car les deux runs sont potentiellement très éloignés et donc la fusion ne peut pas être faite sur place (fait dans le même tableau que l'on souhaite trier) en plus ne me pas être optimale en termes de cache. Ce qui fait que nous avons un algorithme qui ne peut pas trier un tableau sans allouer un second tableau de même taille.

En général on dit qu'un algorithme est k-aware quand il va s'autoriser à observer les k premiers éléments de la pile S. L'AdaptiveShiverSort est aussi dit 3-aware car il va s'autoriser à observer les 3 éléments en tête de pile. Les algorithmes comme PowerSort et MinimalSort eux s'autorisent à observer toute la pile. Les algorithme k-aware ont pour avantage de faire très peu de parcours de la pile et peut garder la pile en mémoire cache mais ont des surcoûts en fusion qui peuvent être important par rapport au MinimalSort (jusqu'à 50% pour le TimSort par exemple). L'idée était donc de trouver un juste milieu entre le parcours de la pile et le coût de fusion. L'AdaptiveShiverSort est le premier algorithme k-aware à avoir un surcoût linéaire par rapport au MinimalSort.

Algorithme	Complexité	k-aware	Coût de fusion au pire cas
AdaptiveShiversSort	O(n + nH)	3	nH + O(n)
TimSort	O(n + nH)	4	(3/2)nH + O(n)
PowerSort	O(n + nH)	non	nH + O(n)
MinimalSort	O(n + nH)	non	nH + O(n) (parfait)
InsertionSort	$O(n^2)$	non	aucune fusion

C - Algorithmes implémentés

Nous avons donc, au cours de ce projet, implémenté divers algorithmes de tri, tout en les comparant au *std::sort* de la librairie standard C++ afin d'avoir un point de comparaison.

1 - InsertionSort

Le tri par insertion est extrêmement connu, de par sa simplicité d'implémentation. Il a aussi comme propriété d'être en $O(n^2)$, on pourra montrer via les algorithmes NaturalMergeSort et HybridMergeSort, que le tri par insertion est extrêmement efficace sur les petits tableaux, notamment grâce à l'exploitation de la mémoire cache, il profite aussi des parties déjà triées au début du tableau, notamment grâce au prédicteur de branchement.

2 - TimSort

Condition de fusion de la version actuelle de TimSort :

```
si h > 3 et r1 > r3 faire fusionner R2 et R3
sinon si h > 2 et r1 > r2 faire fusionner R1 et R2
sinon si h > 3 et r1 + r2 > r3 faire fusionner R1 et R2
sinon si h > 4 et r2 + r3 > r4 faire fusionner R1 et R2
sinon sortir de la boucle
```

La nouvelle version du TimSort possède 4 conditions de fusion pour éviter certains cas où l'algorithme ne fonctionnait pas (cas découverts grâce aux méthodes formelles). Cette version est 4-aware, c'est à dire que l'algorithme se permet d'aller observer les 4 éléments en tête de pile pour savoir s'il va effectuer les fusions intermédiaires.

3 - AdaptiveShiverSort

Note: Soit h la taille de S et Ri le ième élément en tête de S, on note Ri la taille de ri et soit li = ⌊log2(ri)⌋.

Condition de fusion de l'adaptiveShiversSort selon le papier de Vincent Jugé [5] *:*

```
si h > 3 et l1 > l3 faire fusionner R2 et R3
sinon if h > 3 et l2 > l3 faire fusionner R2 et R3
sinon if h > 2 et l1 > l2 faire fusionner R1 et R2
sinon sortir de la boucle
```

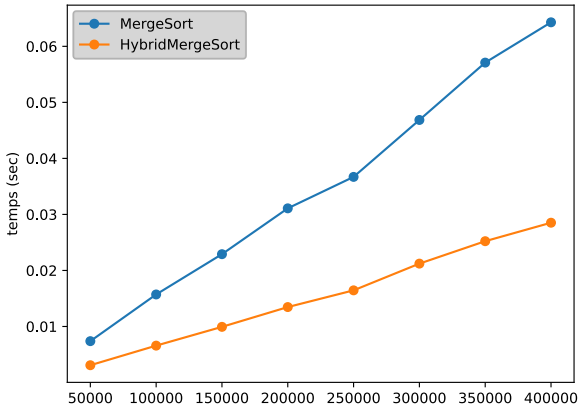
Le code de l'adaptive shiver sort est très similaire au TimSort à l'exception des conditions de fusion. Selon Vincent Jugé (créateur de l'algorithme) il y aurait moins de 10 lignes de modification dans le TimSort de la librairie standard java pour passer à l'adaptive shiver sort. On peut aussi noter que l'algorithme est 3-aware.

D – Implémentation

1 - Tri de petit tableau

Beaucoup d'algorithmes de tri moderne utilisent le tri par insertion pour trier des petits sous-tableau (IntroSort, TimSort, MergeSort …).

Nous avons voulu mesurer à quelle point le gain de performance était marqué. Pour cela, on définit le NaturalMergeSort par l'implémentation la plus classique du tri fusion, et l'HybridMergeSort, un tri fusion qui va effectuer un tri par insertion sur les sous-tableaux de taille K ou moins. Pour avoir testé diverse valeurs et en regardant ce qui se fait globalement on a pris K = 32.



La courbe MergeSort correspond au NaturalMergeSort

Nos benchmarks ont pu montrer que les deux variantes de MergeSort étaient très loin d'être des algorithmes capables de rivaliser avec les algorithmes modernes de tri, cependant c'est un excellent moyen de montrer le gain d'efficacité apporté par l'utilisation du tri par insertion sur les tableaux de petite taille.

2 - Détection et incrémentation des runs

Les runs sont détectés par un algorithme glouton (on parcourt le tableau case par case en ayant seulement de la visibilité sur l'élément courant et précédent) et stockés dans un module étant une surcouche au *std::vector* de la librairie standard c++. Lorsque l'on commence un nouveau run : s'il reste au moins 2 éléments on regarde les deux premier élément A_0 et A_1.

• Si $A_0 \leq A_1$

• alors on va continuer à parcourir le tableau tant que le run est croissant.

• Sinon on va continuer à parcourir le tableau tant que le run est strictement décroissant, lorsque l'on arrive à la fin du run décroissant on l'inverse afin d'avoir un run croissant. Pour éviter d'avoir énormément de runs de petite taille, lorsque l'on trouve un run de moins de 32 éléments on effectue un tri par insertion afin d'avoir un run de 32 éléments. Pour avoir testé avec diverses valeurs (puissance de 2 pour être plus cohérent vis-à-vis de la mémoire cache) en dessous de 8 et au-dessus de 32 il y a une perte d'efficacité. Dans un cas il y a trop de runs et dans l'autre le tri par insertion commence à être vraiment coûteux.

Dans les piles de run nous stockons les runs strictement décroissants ou croissants (pour être plus exact, nous stockons les indices des débuts/fins de run pour des raisons évidentes de performance), dans l'absolu, stocker les runs croissants et décroissants (qui seront inversés) ne changeraient pas grand-chose au moment de la fusion, cependant, cela permet de gagner du temps au moment de la fusion.

Notre détecteur de run garde en mémoire si nous sommes actuellement dans un run décroissant ou non via un *booléen*, pour savoir si l'on ajoute au run courant, il y a deux cas :
Soit A_n la case courante du tableau, A_{n+1} la case suivante et bool le booléen qui est vrai si le run courant est croissant et faux sinon.

On incrémente le run courant si :

((An <= An+1) et (bool)) ou ((An > An+1) et (non bool)) <=>
((An <= An+1) et non (non bool)) ou (non (An <= An+1) et (not bool)) <=>
(An <= An+1) XOR (non bool) <=>
(An > An+1) XOR (bool)

On arrive donc à une condition qui une fois traduite en assembleur donne deux instructions assembleur dénuées de branchement qui est bien plus efficace qu'un *((A_n <= A_{n+1}) et (bool)) ou ((A_n >= A_{n+1}) et (non bool))* qui une fois compilé prendrait beaucoup plus d'instructions assembleur.

Exemple de code asm généré par gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0 (en O2) pour (A_n > A_{n+1}) XOR (bool)

```
_Z6comp_iiib:

.LFB0:
    cmpl    %esi, %edi  ; inst 1
    setg    %al         ; inst 2
    xorl    %edx, %eax  ; inst 3
    ret                 ; inst 4
```

Exemple de code asm généré par gcc (en O2) pour

((A_n <= A_{n+1}) et (bool)) ou ((A_n >= A_{n+1}) et (non bool))

```
_Z7comp_i2iib:
.LFB1:
    cmpl    %esi, %edi  ; inst 1
    setle   %al         ; inst 2
    andb    %dl, %al    ; inst 3
    jne .L3
; inst 4 et jump (donc pas bon pour la prédiction de branchement)
    cmpl    %esi, %edi  ; inst 5
    setl    %al         ; inst 6
    orl %edx, %eax      ; inst 7
    xorl    $1, %eax    ; inst 8
.L3:
    rep ret
; inst 5 et 6 (si viens du jump) ou 9 et 10 (si viens de inst 8)
; rep ret = ret mais avec 2 instructions pour l'alignement mémoire
```

Le but n'étant pas de décortiquer le code assembleur généré par gcc, on peut cependant facilement voir que le code généré pour la première condition est beaucoup plus compact et ne possède pas de branchement conditionnel.

3 - Structure de la pile

Comme dit précédemment la pile est une surcouche au *std::vector* qui stock les indices des runs, par exemple avec le tableau (nous considérons sur cet exemple que nous n'agrandissons pas les runs):

12, 7, 6, 5, 4, 3, 1, 0, 0, 7, 14, 36, 37, 42, 73, 3, 3, 5, 21, 21, 21, 24.

Le premier run (qui sera inversé) va de la case 0 à la case 7, le second run de la case 8 à 14 et le dernier run de la case 15 à 21. On va donc stocker dans la pile 0, 8, 15 et 22. On a donc une pile de *nombre de run* + 1 élément. Lorsque l'on souhaite connaître le run en tête de pile, on extrait l'élément en taille de pile X_1 puis on regarde l'élément en tête de pile X_0, on a donc un run de X_0 à X_1, X_1 exclue ce qui correspond au standard du C++.

4 - Condition de fusion de l'AdaptativeShiverSort

(On utilise ici les variables définies avec l'AdaptiveShiverSort)

Soit C1 la première condition de fusion (*h > 3 et l1 ≥ l3*).

Soit C2 la seconde condition de fusion (*h > 3 et l2 ≥ l3*).

Soit C3 la troisième condition de fusion (*h > 2 et l1 ≥ l2*).

Or l1 ≥ l2 correspond à une comparaison des bits de poids fort que l'on peut implémenter de manière assez astucieuse et efficace via de l'arithmétique bit à bit.

l1 ≥ ((non l1) et r1)

On peut aussi fusionner C1 et C2 car il fusionne les mêmes runs.

On a donc *h > 3 et ((l1 ou l2) ≥ l3).*

Après une discussion avec Vincent Jugé, celui-ci nous a expliqué que la troisième condition est juste utile pour la démonstration d'optimalité en coût de fusion de l'adaptive ShiverSort mais en pratique on peut s'en passer et réduire les cas de fusion à *((l1 ou l2) ≥ l3)*.

5 - Fusion de run

La fusion entre run est assuré par la librairie standard c++ avec le std::inplace_merge. Notre pile contenant les indices de nos runs, on utilise les primitives c++ sur les itérateurs pour faire nôtre fusion de manière efficace.

6 - Structure du Code

Nous avons implémenté notre code avec une template c++ prenant en paramètre la "fonction" permettant de trouver les runs *RunFinder*, la "fonction" permettant de dire comment on fusionne deux rules *MergeStrat* ainsi que les conditions de fusion *Rules*. Petite particularité à l'implémentation, *RunFinder*, *MergeStrat* et *Rules* sont implémentés par des objets ayant l'opérateur *()* surchargé, ce qui permet d'avoir des "fonctions" qui ont une mémoire. Ce qui nous à parti d'implémenter le TimSort et l'AdaptiveShiverSort avec un minimum de duplication de code.

III - Benchmark

A - Méthodes de benchmark

Nous étions à l'origine parti sur l'utilisation de la librairie de microbench développée par Google. Cependant la librairie n'étant pas vraiment adapter pour des benchmark long et très peu souple nous avons opté pour une librairie plus artisanale.

Nous sommes finalement parti sur un module de benchmark basé sur la librairie std::chrono, basiquement on exécute le code 3 fois sans mesurer le temps afin de charger certains éléments en cache, puis on mesure k exécutions du code avec k = 33. Pour éviter d'être parasité par du bruit au moment du bench, nous prenons la médiane et non la moyenne des temps d'exécution.

Nous avons testé diverses valeurs pour k. Pour cela nous avons pris en compte deux éléments, premièrement, un k trop grand ralentissait le temps d'exécution, nous avons aussi mesuré le rapport de taille entre l'écart interquartile et la médiane. Avec le calcul (écart-interquartile / médiane) * 100.

On obtient le tableau suivant :

k/algo	TimSort	AdaptiveShiverSort	stdSort	mergeSort	hybridMergeSort
13	1.12%	1.12%	1.70%	2.70%	0.77%
33	0.29%	0.18%	0.20%	0.12%	0.22%
97	0.07%	0.08%	0.16%	0.14%	0.16%

Mesure effectuée sur une seule mesure avec un processeur intel i3-6006U

La première chose à noter est l'utilisation des médianes et écart interquartile au lieu de la moyenne et l'écart type avec d'éviter les artefacts par exemple si sur 5 mesures de temps (du même code exactement) on a : 99, 100, 100, 101 et 180. La valeur 100 (médiane) semble plus représentative du comportement habituel que 116.2 (moyenne). Mais afin de mesurer à quel point la médiane est représentative, nous avons donc mesuré la différence entre la médiane et l'écart interquartile.

On peut voir par exemple que sur cette mesure il y avait 2.7% de différence entre la médiane et l'écart interquartile pour le mergeSort avec k = 13, ce qui est conséquent. Après plusieurs mesures, nous avons opté pour k = 33 car il offrait un bon compromis entre rapidité d'exécution et fiabilité.

B - Génération de tableaux

L'utilisation de données "de la vie réelle" serait en théorie l'idéal mais en pratique peu exploitables car le choix de celles-ci serait arbitraire et ne correspondrait qu'à un champ plus ou moins restreint de cas pratiques.

Pour tester les différents algorithmes implémentés, nous avons donc eu besoin d'avoir la capacité de générer des tableaux de tailles et d'entropies différentes qui nous permettraient de voir l'évolution de la performance de ces algorithmes en fonction de l'évolution des caractéristiques de ces tableaux.

Nous avons utilisé le langage python car il permet d'avoir une syntaxe simple et offre des outils de construction de graphiques très performants (on a donc une homogénéité entre la construction des tableaux de test et des graphiques de résultats des tests).

1 - Description des différentes méthodes

Pour cette génération de tableaux, nous avons donc pensé à plusieurs méthodes (nous ne les avons finalement pas toutes utilisées car certaines se sont révélées peu pertinentes en pratique).

i - Génération de tableaux complètement aléatoires

Cette génération des tableaux est la plus basique, il s'agit simplement, pour un tableau demandé de n éléments avec une borne supérieure et inférieure, de générer n entiers aléatoires entre ces deux bornes.

On aurait aussi pu penser à la permutation d'éléments d'un tableau déjà triée, ce qui pourrait permettre de n'avoir aucun nombre en double, mais l'incidence sur les performances des différents algorithmes serait négligeable.

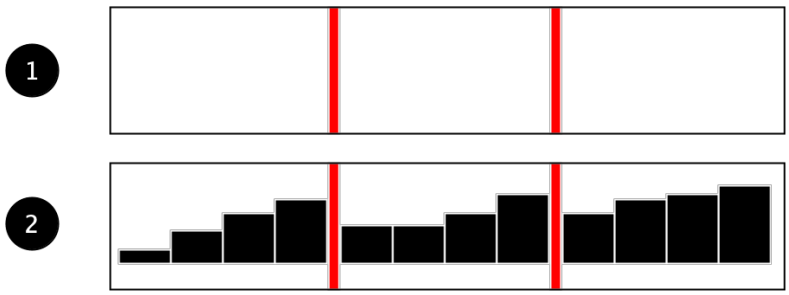
Il est possible d'obtenir le même résultat via la méthode de génération d'un tableau à entropie donnée, que l'on verra plus tard, en choisissant l'entropie la plus haute possible.

Nous ne l'avons donc pas utilisée, car elle ne correspond statistiquement qu'à une entropie maximale qui, de plus, ne permettrait pas de remarquer le gain en performances qu'apporte les méthodes basées sur l'utilisation de runs dont l'Adaptative ShiverSort fait partie.

ii - Génération de tableaux avec runs de taille constante

Cette méthode permet de générer un tableau, de la même façon que la première méthode mais ayant un nombre donné de runs, qui auront tous la même taille (à 1 près).

Pour R runs et un tableau de taille n :



(1) On place les délimitations des ième runs à la position i * (n / R)

(2) On remplit les runs avec des tableaux triées

Schéma 1 - Représentation de la construction des runs

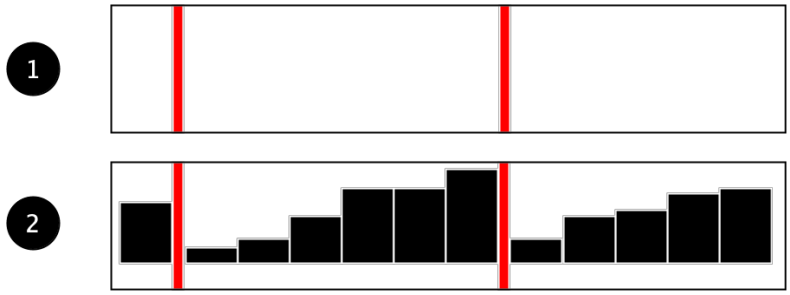
Cette méthode permet de choisir “à quelle point le tableau est triée” de manière assez instinctive mais rend la comparaison de différentes méthodes de tri à des tailles de tableaux relativement compliquée et d’autant plus dans le cadre de tests automatisés.

Si l’on veut que les tableaux soient « à moitié » triées, il faudra en effet un nombre de runs différents entre un tableau de taille 3 et de taille 3 millions. Pour choisir combien de runs on souhaite avoir en fonction de la taille du tableau, il nous faudrait définir une relation entre taille du tableau et nombre de runs qui nous semblerait quelque peu arbitraire (pour un tableau de taille n, faut-il n/2 runs ? ou alors des runs de taille log(n) ? ou autre ?).

iii - Génération de tableaux avec runs de taille aléatoire

Avec cette méthode, nous avons tenté d’obtenir des tableaux plus proches de “la vraie vie” qu’avec la méthode précédente. Ici, on génère un tableau avec un nombre de runs donné mais de taille totalement aléatoire.

Pour R runs et un tableau de taille n :



(1) On tire de manière uniformément aléatoire R-1 délimitations entre 1 et n

(2) On remplit les runs avec des tableaux triées

Schéma 2 - Représentation de la construction des runs

Cette méthode souffre cependant des mêmes défauts que la méthode précédente.

De plus, étant donné que les limites sont tirées de manière uniformément aléatoire, il est possible que deux tirages donnent la même position, cela résulterait en un run de taille 0, c’est à dire un run en moins.

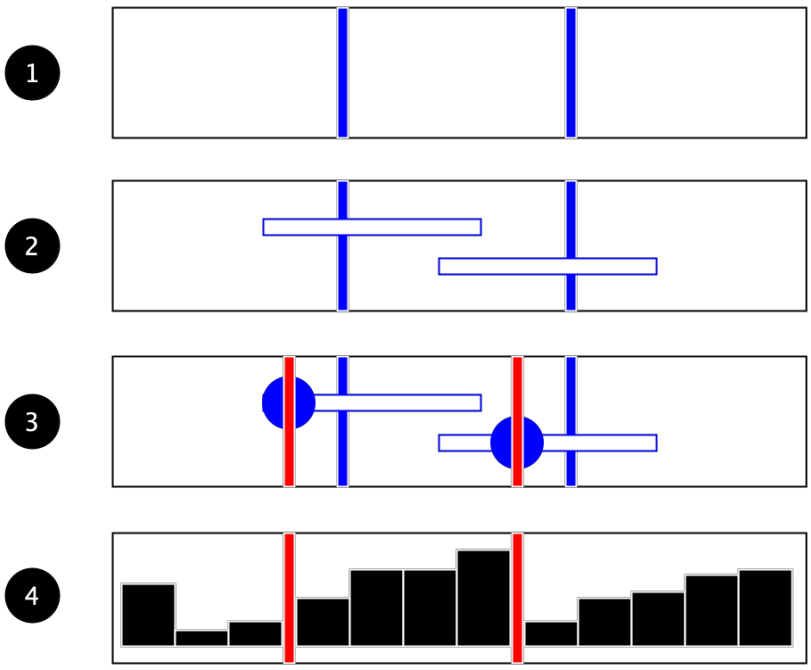
iv - Génération de tableaux avec runs de taille semi aléatoire contrôlé par un paramètre delta

Cette méthode permet de générer un tableau avec des runs de taille semi-aléatoire.

On entend ici par “semi-aléatoire” un tirage uniforme des délimitations de runs entre des bornes qui sont choisies par un paramètre qu’on appellera delta.

Avec delta = 0, on arrive au même résultat qu’avec la méthode ii.
Avec delta = 1, on arrive au même résultat qu’avec la méthode iii.

On procède, donc pour R runs, un tableau de taille n et un paramètre delta, aux étapes suivantes (ce schéma suivant représente visuellement l’utilisation de cette méthode pour 3 runs et delta = 0.3):



(1) On place les délimitations des ième runs à la position i * (n / R)

(2) Pour chaque délimitation, la borne inférieure se situe à la position (distance entre 0 et position délimitation) * delta et la borne supérieure à la position (distance entre position délimitation et n) * (1 - delta)

(3) On tire une délimitation de run entre chaque borne

(4) On remplit les runs avec des tableaux triées (on peut choisir la proportion de runs décroissants)

Schéma 3 - Représentation de la construction des runs

Cette méthode avait pour objectif de pouvoir contrôler plus finement le “réalisme” des tableaux mais elle souffre malheureusement toujours des mêmes problèmes que les méthodes ii et iii.

v - Génération de tableaux ayant une entropie donnée

Cette méthode permet de générer un tableau approximant une entropie demandée (généralement avec une précision de l’ordre du dixième pour des tableaux de taille > 1000).

Nous allons développer cette méthode dans la partie suivante.

2 - Génération d'un tableau à entropie donnée

Nous allons revenir ici sur une méthode de génération de tableaux permettant d'approximer de manière relativement précise une entropie donnée.

On peut inverser la formule de l'entropie pour obtenir la taille des runs nécessaires (où le tableau serait une succession de runs de même longueur) afin d'avoir un tableau d'une entropie donnée.

$$H(X) = -\sum_{i=1}^{p} P_i \log_2(P_i)$$

$$P_i = \frac{taille\ ième\ run}{p}$$

On va traiter, ici, le cas où tous les runs ont la même taille.

On fixe $taille\ ième\ run$ donc $P_i = \frac{taille\ ième\ run}{n} = \frac{1}{nombre\ de\ runs}$

On pose $p = nombre\ de\ runs$

$$H(X) = -\sum_{i=1}^{p} \frac{1}{p} \log_2\left(\frac{1}{p}\right)$$

$$H(X) = -\frac{p}{p} \log_2\left(\frac{1}{p}\right)$$

$$H(X) = -\log_2\left(\frac{1}{p}\right) = \log_2(p)$$

$$\boxed{H(X) = \log_2(nb\ runs)}$$

Donc on a également :

$$\boxed{2^{H(X)} = nb\ runs}$$

On prend donc ici le cas le plus simple où tous les runs font la même taille, mais il aurait pu être intéressant d'avoir une formule permettant de construire des tableaux avec des runs de plusieurs tailles différentes. La taille des runs est en effet un élément ayant un rôle dans la stratégie des différents algorithmes testés.

Pour des raisons pratiques, nous avons décidé de choisir l'entropie indirectement, via un pourcentage où 0% serait l'entropie minimale et 100% l'entropie maximale pour une taille de tableau donnée.

Cette manière de faire simplifie les tests ainsi que la lecture des résultats, car elle permet de comparer des tableaux de tailles très différentes de la même manière en normalisant l'entropie maximale de toutes les tableaux (un tableau de taille 3 a en effet en entropie maximum très faible, contrairement à un tableau de taille 3 milliards).

Une fois qu'on connaît le nombre de runs que le tableau doit contenir pour arriver à une entropie donnée, on construit le tableau de la même manière qu'avec la méthode (ii).

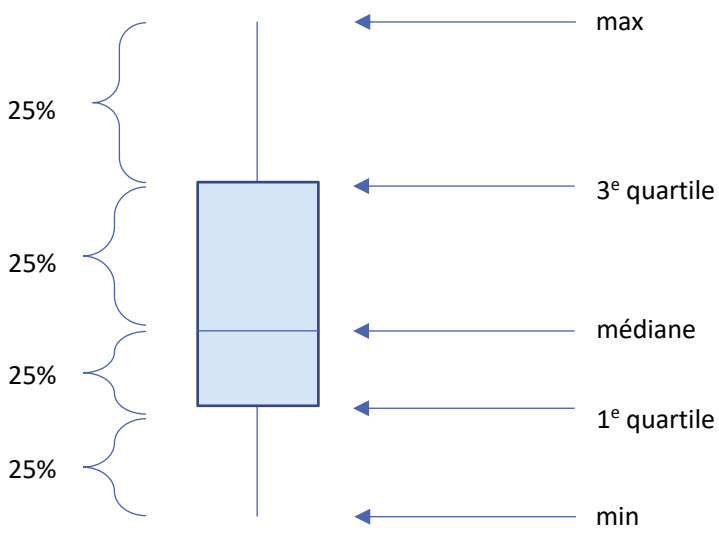
Parmi toutes les méthodes développées, nous avons choisi cette méthode de génération de tableaux car elle permet, comme vu plus haut, de gérer l'entropie de la même manière quelle que soit la taille du tableau ce qui rend l'automatisation des tests bien plus aisée.

C - Résultats des tests

Le bench des différents algorithmes a été fait sur les serveurs du laboratoire du LIP6 dont la configuration sera détaillée plus tard. Il a duré environ 8h et a pris 20 Go d'espace disque.

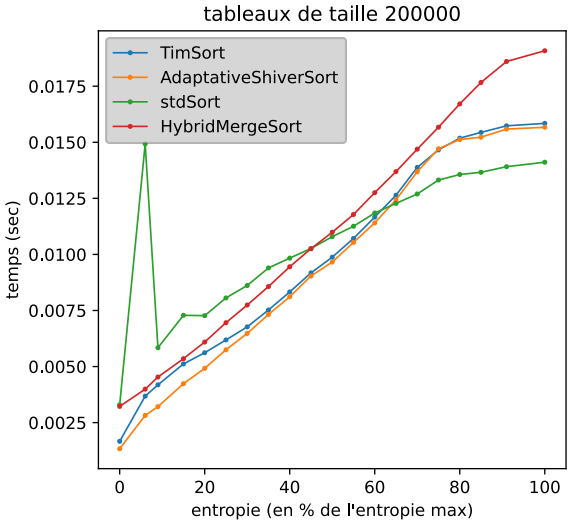
Pour apprécier au mieux les différences entre les différents algorithmes, nous avons produit 3 séries de graphique sur ces données : le temps d'exécution en fonction de l'entropie, le temps d'exécution en fonction de la taille du tableau et, enfin, une heatmap permettant d'avoir une vue sur les données dans leur globalité.

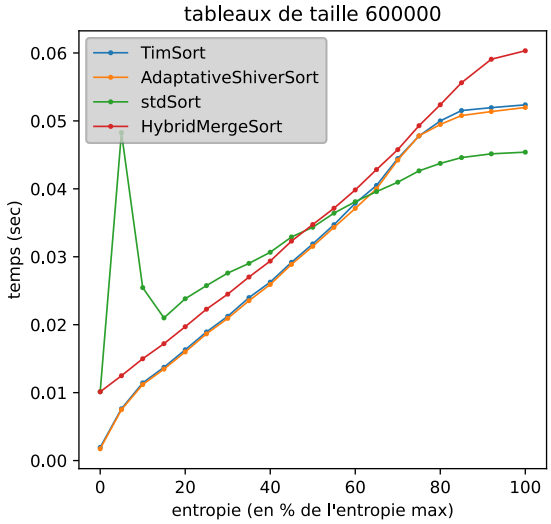
Dans le deuxième graphique, nous utiliseront la représentation en « boite à moustache » qui permet, pour un ensemble de mesures, de voir le 1er quartile, la médiane, le 3^{e} quartile, la valeur min et max.

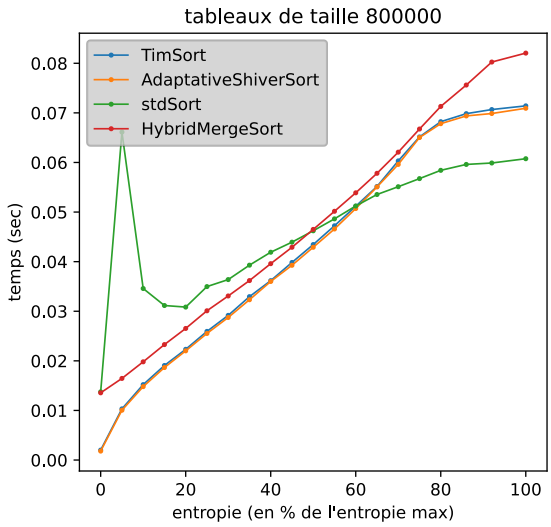


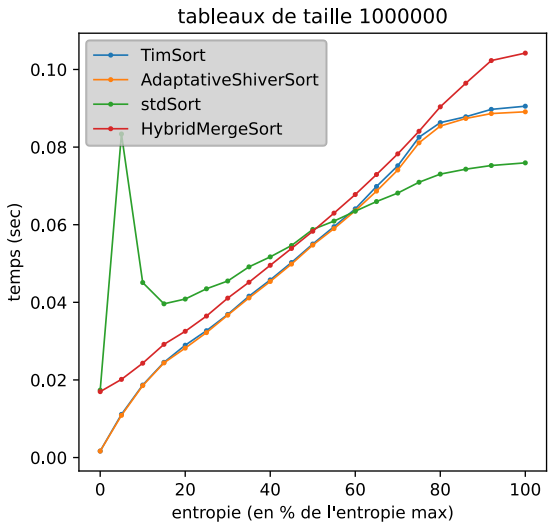
Nous verrons donc, dans un premier temps, un graphique montrant le temps d'exécution des algorithmes en fonction de l'entropie pour 4 tailles de tableaux différentes. Chaque point est la moyenne de 50 benchs sur des tableaux différentes (et comme vu plus haut, chaque bench est la médiane de 33 exécutions différentes sur le même tableau).

Nous n'avons pas opté pour une représentation en boite à moustache car la variance était tellement faible qu'elles auraient presque toutes été complètement plates.





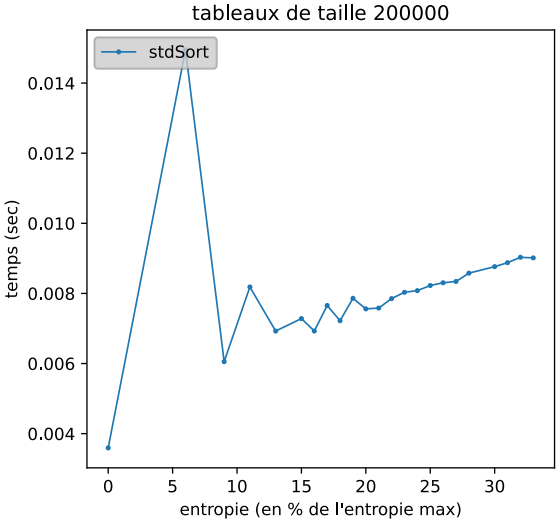




On remarque que l'algorithme AdaptiveShiverSort suit de très près les performances de l'algorithme Tim Sort (avec un léger avantage pour l'AdaptiveShiverSort) ce qui peut s'expliquer par leur grande similarité en termes de stratégie de fusion des runs.

Il serait difficile de tirer des conclusions quant à la supériorité de l'un sur l'autre entre ces deux derniers algorithmes.

On remarque en revanche un pic assez impressionnant dans le temps d'exécution du stdSort au niveau des basses entropies. Nous avons effectué des mesures supplémentaires sur des tableaux de taille 200000.



Il est possible que cette anomalie soit causée par le QuickSort qui serait surutilisé lorsque le tableau est « presque » triée mais nous n'avons pas réussi à trouver d'explication précise.

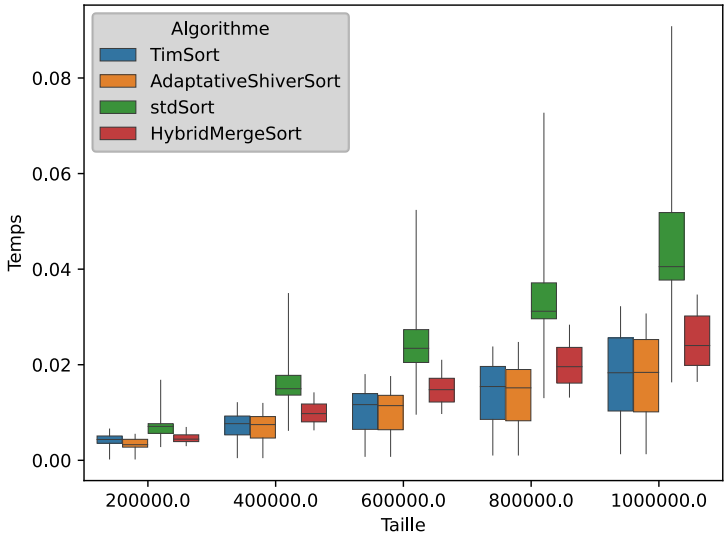
On ne peut pas, avec nos méthodes de génération de tableaux à entropie fixée avoir plus de précision dans les basses entropies. Tous les runs étant en effet de la même taille, il y a un bond entre l'entropie d'un tableau à 1 run (totalement trié) et un tableau à 2 runs. A mesure qu'on ajoute des runs, le bond se réduit ensuite.

En dehors de cette anomalie, on peut également remarquer que l'algorithme stdSort de la bibliothèque standard semble être, dans une moindre mesure, sensible à l'entropie. Ceci peut est dû au fait qu'il utilise dans la plupart des cas (car l'implémentation est libre) l'algorithme Intro Sort [9] où l'on parcourt le tableau d'un bout à l'autre en appliquant différentes méthodes lorsqu'on tombe sur une valeur non triée en fonction de la situation (Intro Sort fera un nombre donné de récursions en QuickSort, puis du HeapSort une fois ce nombre atteint en arrivant finalement à l'InsertionSort lorsque le tableau a une taille inférieure à 32 qui, lui, est en O(n) sur un tableau triée).

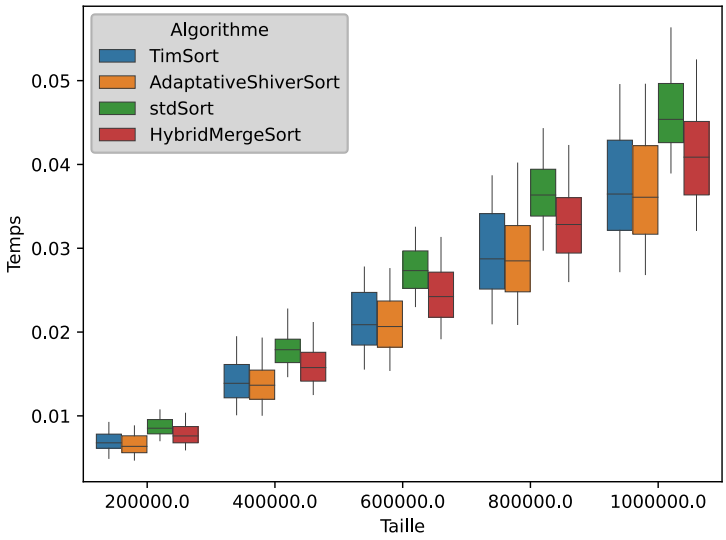
On peut, de plus, relever un infléchissement de toutes les courbes dans les entropies les plus élevées. Celui-ci s'explique également par l'InsertionSort qui crée des runs artificiels de 32 éléments, lorsqu'il n'y en a pas ou qu'ils sont trop petits, de manière très performante (d'une part par la nature même de l'algorithme mais aussi grâce aux prédictions de branchements du processeur qui pourra profiter des mini runs qu'il traite s'il y en a). A partir d'une certaine entropie donc, tous les runs fusionnés feront la même taille.

On va maintenant s'intéresser à l'évolution du temps d'exécution des différents algorithmes en fonction de la taille du tableau.

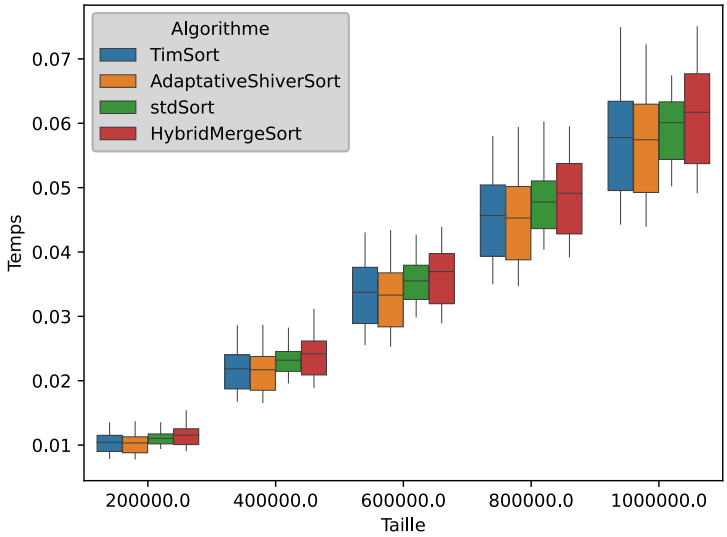
Tableaux d'entropie de 0% à 20%



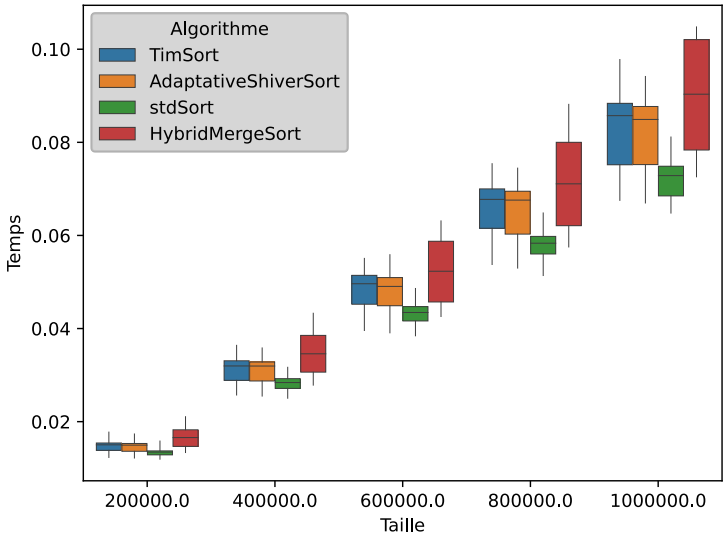
Tableaux d'entropie de 20% à 40%



Tableaux d'entropie de 40% à 65%



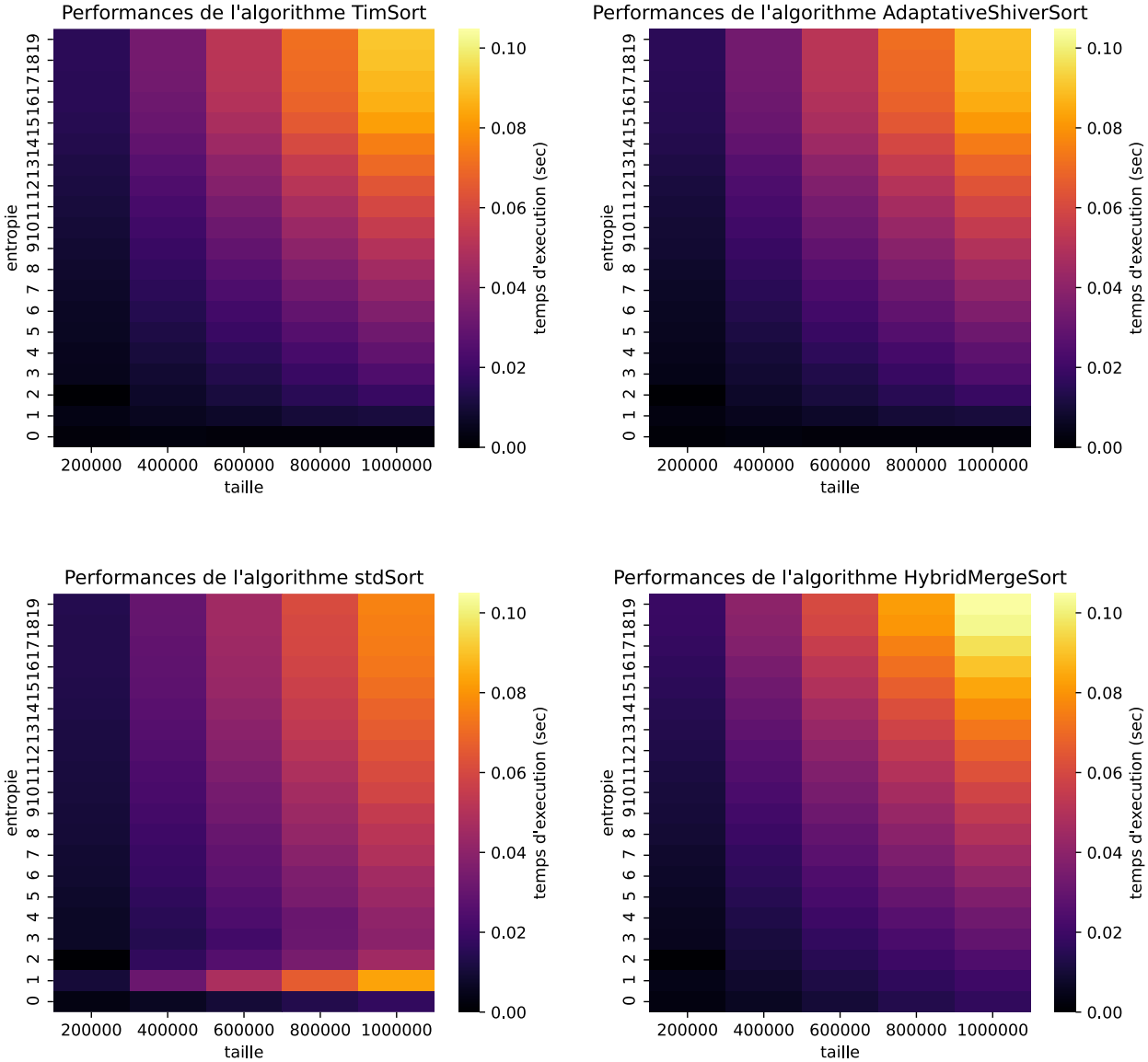
Tableaux d'entropie de 65% à 100%



On remarque, sur les basses entropies (de 0% à 20%), le temps d'exécution du stdSort qui suit une courbe de type n log(n) alors que les temps d'exécution des algorithmes de TimSort et AdaptiveShiverSort suivent une courbe plus linéaire. Cela semble donc confirmer la complexité vu plus haut de ces deux algorithmes qui est de O(n + nH). Avec H(X) petit, on s'approche en effet de O(n) avec donc une évolution du temps d'exécution linéaire en fonction de la taille des tableaux.

On voit donc, comme sur le graphique précédent, que les algorithmes TimSort et AdaptiveShiverSort sont bien plus performants que stdSort lorsque l'entropie des tableaux n'est pas très élevée.

Enfin, on va terminer avec une représentation des résultats en heatmap. Comme pour les deux séries de graphiques précédents, chaque point correspond à 50 bench. On a ici choisi d'afficher la moyenne des temps d'exécution de ces benchs.



On peut voir, avec ces heatmaps, que le stdSort a une vitesse moins variable relativement à l'entropie que les autres méthodes de tri qui sont basées sur l'utilisation de runs. On remarque également l'anomalie du stdSort dans les basses entropies.

Comme on s'y attendait, les algorithmes de TimSort et AdaptiveShiverSort ont tous deux montré une très grande efficacité pour les entropies basses.

Discussion sur les résultats

Le fait que les tableaux produits aient tous les runs de la même taille (à 1 élément près) aurait également pu avoir une incidence sur les résultats, notamment car lors de la recherche de run, les runs de moins de 32 éléments vont “manger” les runs suivants jusqu’à avoir 32 éléments, un très mauvais cas que l’on pourrait imaginer est un tableau avec seulement des runs de 31 éléments. En pratique, sur un tableau avec des runs moins homogènes, on pourrait voir des résultats meilleurs que ceux mesurés, notamment car nous ne “casserons” pas autant de petit run.

Les benchmarks ont été exécutés sur un serveur du laboratoire LIP6 avec la configuration suivante :

```
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
ldd (Ubuntu GLIBC 2.27-3ubuntu1) 2.27
Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz 64bit
       L1d cache:           32K
       L1i cache:           32K
       L2 cache:            256K
       L3 cache:            30720K
RAM memory block size:        2G
Total online RAM memory:     130G
```

L’utilisation d’un serveur dédié permet de réduire de manière considérable la variance dans les temps d'exécution des algorithmes, ce qui rend les résultats finaux plus fiables et permet une meilleure interprétation de ceux-ci.

On peut noter un très léger avantage de l’AdaptiveShiverSort sur le TimSort, cependant la configuration de la machine semble jouer un rôle relativement important dans les performances des différents algorithmes testés. Après avoir lancé des tests sur un ordinateur dont la configuration est détaillée ci-après, on arrive à des résultats très différents :

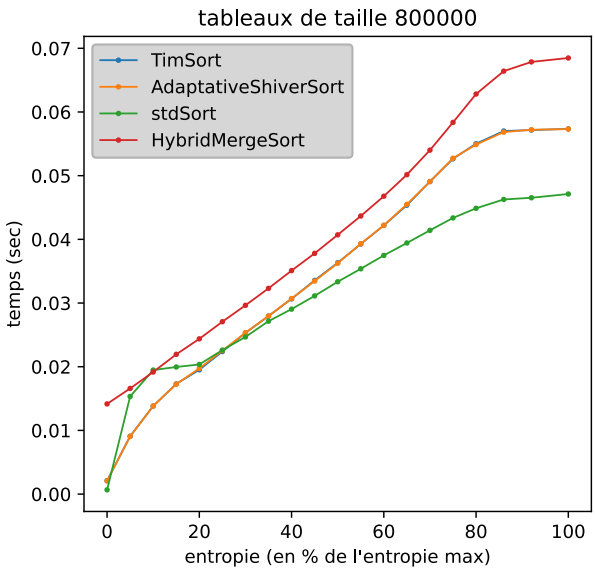


Figure 1 - Tests effectués sur un MacBook Pro
Les courbes de TimSort et MergeSort se confondent

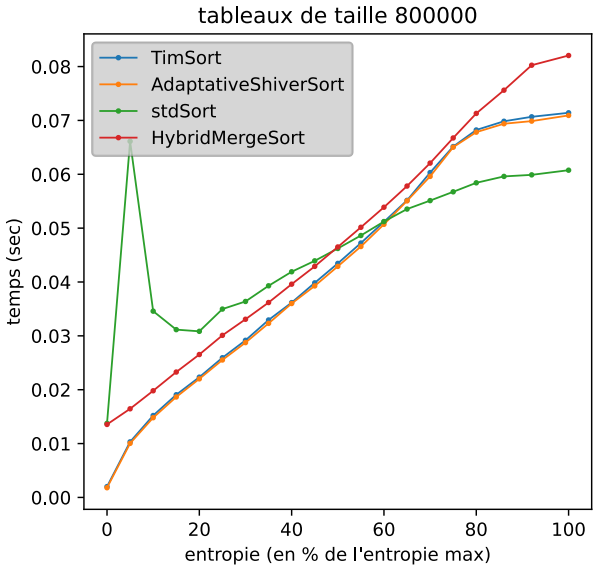


Figure 2 - Tests effectués sur les serveurs du LIP6

```
Nom du modèle :                     MacBook Pro (13-inch, 2018, Four Thunderbolt 3 Ports)
Système d'exploitation :            MacOS 10.14.1 (18B75)
C++ version :                       4.2.1
Nom du processeur :                 Intel(R) Core(TM) i5-8259U CPU @ 2.30GHz 64bit
       L1d cache:           32K
       L1i cache:           32K
       L2 cache:            262K
       L3 cache:            6291K
Nombre de processeurs :             1
Nombre total de cœurs :             4
Mémoire (RAM) :                     8 Go
```

Au vu du peu d'écart observé entre les algorithmes de TimSort et d'AdaptiveShiverSort, et de la variabilité importante des résultats en fonction des configurations sur lesquels les tests sont effectués, les résultats ne semblent pas assez significatifs pour conclure, en moyenne, si l'un ou l'autre est le meilleur.

Conclusion

Nous n'avons pas eu d'énorme blocage au cours de ce projet, cependant, l'implémentation des différents algorithmes ainsi que leurs tests et leurs comparaisons étaient non triviaux.

Nous avons eu la chance de pouvoir exécuter nos benchmarks sur les serveurs du lip6, ainsi que de rencontrer Vincent Jugé (le créateur de l'AdaptiveShiverSort) avec qui nous avons pu discuter.

Initialement l'idée du projet était de comparer finement divers algorithmes de tris afin d'en évaluer la pertinence. Étant donnée le stade d'avancement du projet, il serait parfaitement envisageable (mais trop chronophage pour être intégré au projet) d'aller encore plus loin et d'étendre notre implémentation de l'AdaptiveShiverSort afin qu'elle puisse devenir une alternative viable au *std::sort* : implémentation utilisant la norme des *RandomAccessIterator*, utilisation d'algorithmes alternatifs pour des tableaux plus petit (l'AdaptiveShiverSort montrant son efficacité qu'à partir d'une certains tailles de tableau). Il aurait également été envisageable d'aller plus loin dans le "réalisme" des tableaux générées, cependant l'impact sur les résultats finaux aurait probablement été négligeables par rapport au travail que cela aurait demandé.

Ce projet nous a donc permis, d'une part, de mettre en pratique nos acquis théoriques concernant les notions moderne d'architecture des processeurs. Cela nous a aussi permis et appris à mettre en place un protocole expérimental précis, faire des tests fiables, reproductible et représentatifs afin de montrer l'intérêt (ou le non intérêt) de tel ou tel algorithme.

Bibliographie

[1] Auger, N., Nicaud, C. and Pivoteau, C., 2020. Merge Strategies: From Merge Sort To Timsort. [online] Hal.inria.fr. Available at: <https://hal.inria.fr/hal-01212839v2> [Accessed 16 March 2020].

[2] Buss, S. and Knop, A.. Strategies for stable merge sorting. In Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 1272–1290. Society for Industrial and Applied Mathematics, 2019.

[3] Goldstine, H. von Neumann, J. Planning and coding of problems for an electronic computing instrument. 1947.

[4] Hoare, T.. Algorithm 64 : Quicksort. Communications of the ACM, vol. 4, issue 7, page 321, 1961

[5] Jugé, V., 2019. Adaptive Shivers Sort: An Alternative Sorting Algorithm. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1809.08411> [Accessed 16 March 2020].

[6] Munro, J. and Wild, S., 2020. *Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt To Existing Runs*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1805.04154> [Accessed 16 March 2020].

[7] Peters, T., 2020. [online] Svn.python.org. Available at: <https://svn.python.org/projects/python/trunk/Objects/listsort.txt> [Accessed 16 March 2020].

[8] Williams, J.. Algorithm 232 : Heapsort. Communications of the ACM, vol. 7, pages 347–348, 1964.

[9] [Software Excerpt] “__sort” from “stl_algo.h”, 2017-09-13 ⟨https://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/a00521_source.html#l01963⟩