Evaluation énergétique du QuickSort

Auteurs :
Maxime CLEMENT
Jordan PIORUN

18 Janvier 2016

Table des matières

Introduction **4**

1 QuickSort **5**

1.1 Algorithme 5

1.2 Complexité 6

2 Évaluation **7**

2.1 Consomation énergétique en fonction du temps 7

2.2 Résultat 8

Conclusion **13**

Introduction

L'essor du monde de l'embarqué demande une autonomie de plus en plus grande. Si la solution la plus explorée est l'amélioration des batteries, une autre possibilité est la diminution de la consommation. Il est donc important de pouvoir quantifier cette consommation, afin de mesurer l'impact des optimisations apportées.

Lorsque nous devons écrire un programme, il est donc important de choisir le langage le plus adapté, c'est à dire celui qui a la meilleure consommation énergétique en fonction du temps d'exécution.

Afin d'apporter une comparaison entre différents langages de programmation, nous nous sommes fixé comme objectif d'étudier les performances d'un algorithme, implémenté dans différents langages.

QuickSort

Le QuickSort, appelé *tri rapide* en français est un algorithme de tri très performant et communément utilisé dans le monde de l'informatique. Les sections 1.1 et 1.2 détails son fonctionnement ainsi que ses propriétés.

1.1 Algorithme

Le QuickSort est fondé sur le pattern *diviser pour régner*. La figure 1.1 est un pseudo-code permettant de comprendre son fonctionnement.

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] <= pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Figure 1.1: Pseudo algorithme du QuickSort

1.2 Complexité

La complexité moyenne du QuickSort est de $n \log n$, cependant cette complexité est de n^2 dans le pire des cas.

Évaluation

2.1 Consomation énergétique en fonction du temps

Notre but était de comparer les consommations énergétiques du même programme écrit dans différents langages. Nous avons pour cela choisi le QuickSort, que nous implémentons dans les langages suivants : C, C optimisé, Java, Ruby, Perl, Python.

Afin que les comparaisons soient pertinentes, nous avons gardé le même principes lors des différentes implémentations. Nous avons donc le même fichier contenant 10 millions de nombres dans un ordre identique. Chaque programme charge le fichier en mémoire, puis il effectue ensuite le tri. Ces différentes implémentations ainsi que le fichier contenant les nombres aléatoires peuvent être trouvées sur notre page Github[1].

Les différentes exécution ont été lancées dans les même condition, sur une machine sous Debian Jessie 64 bits, 4Go de Ram, possédant un processeur Intel i5-4570 3.2 GHz.

Nous avons obtenu les mesures relatives aux différentes implémentations à l'aide de la librairie PowerApi[2].

[1] `https://github.com/maxcleme/GreenComputing`
[2] `http://www.powerapi.org/`

2.2 Résultat

L'exécution des différentes versions du QuickSort a permis de tracer les courbes de résultat des figures 2.1 à 2.6. Ces dernières représentent la valeur de la consommation énergétique en fonction en temps lors de l'exécution des différents QuickSort.

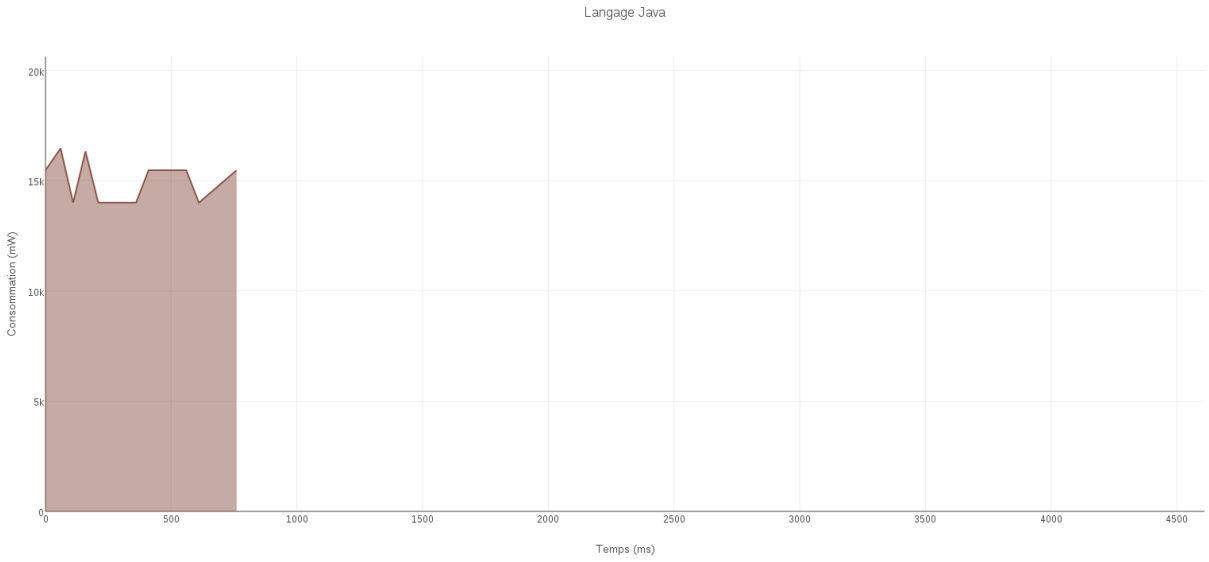

Figure 2.1: Consommation du QuickSort en Java

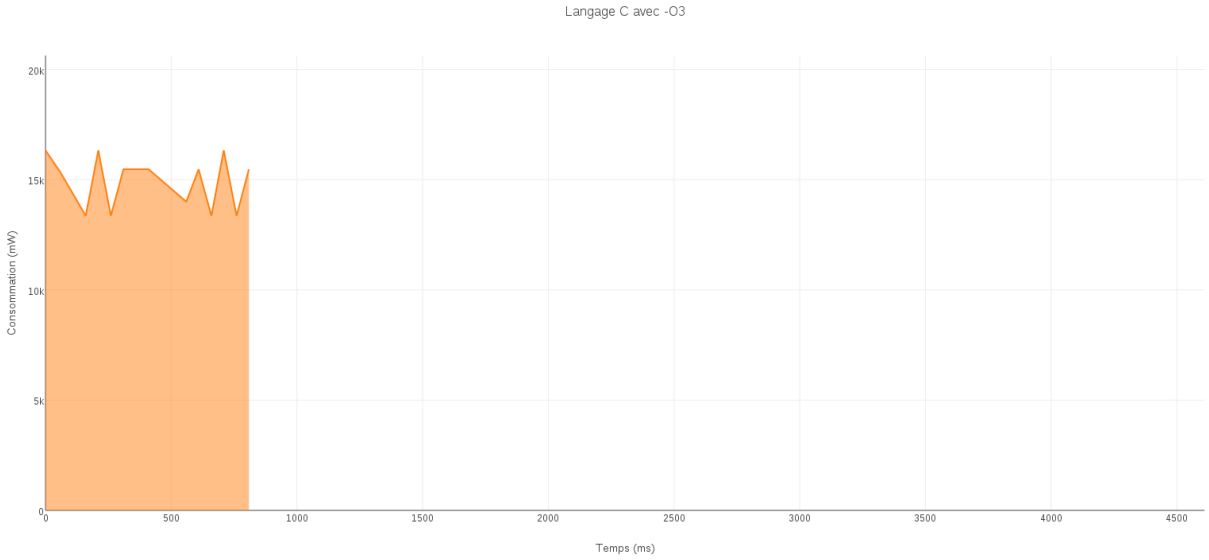

Figure 2.2: Consommation du QuickSort en C optimisé

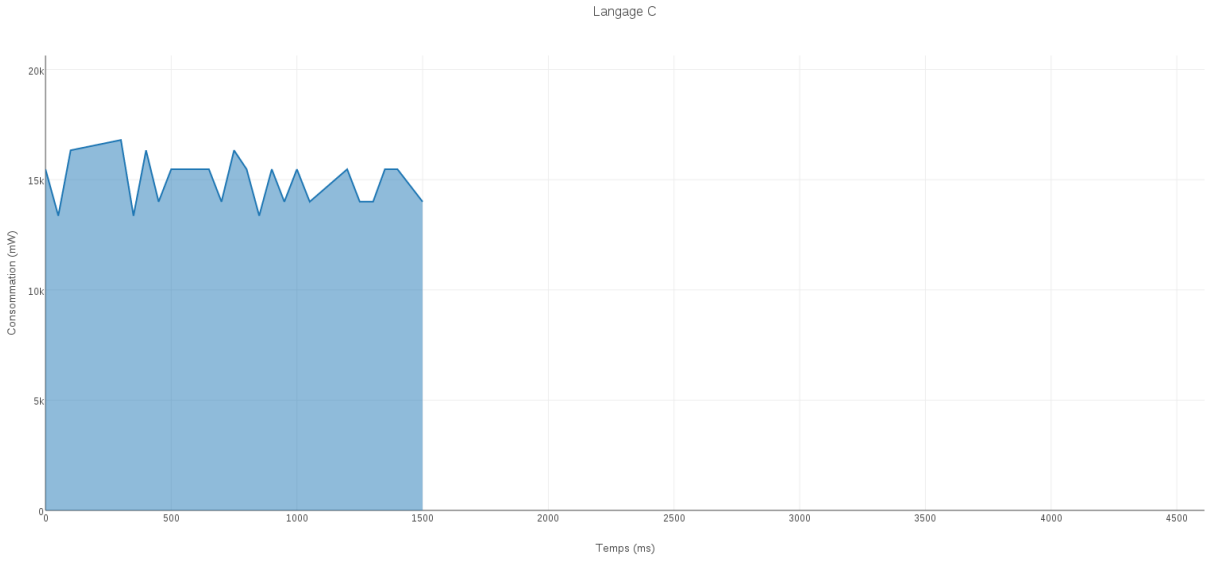

Figure 2.3: Consommation du QuickSort en C

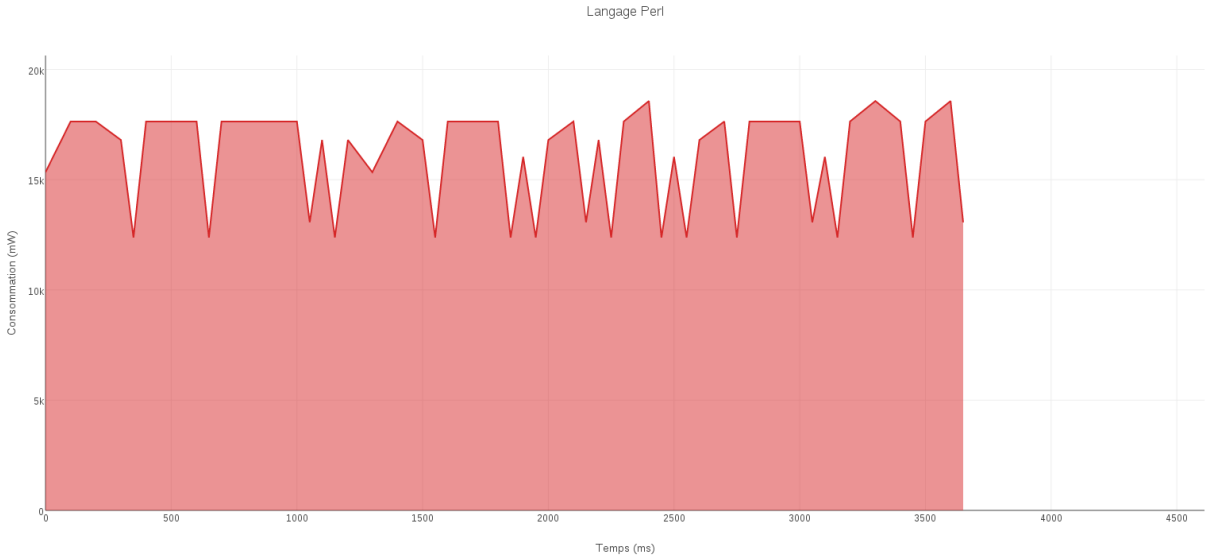

Figure 2.4: Consommation du QuickSort en Perl

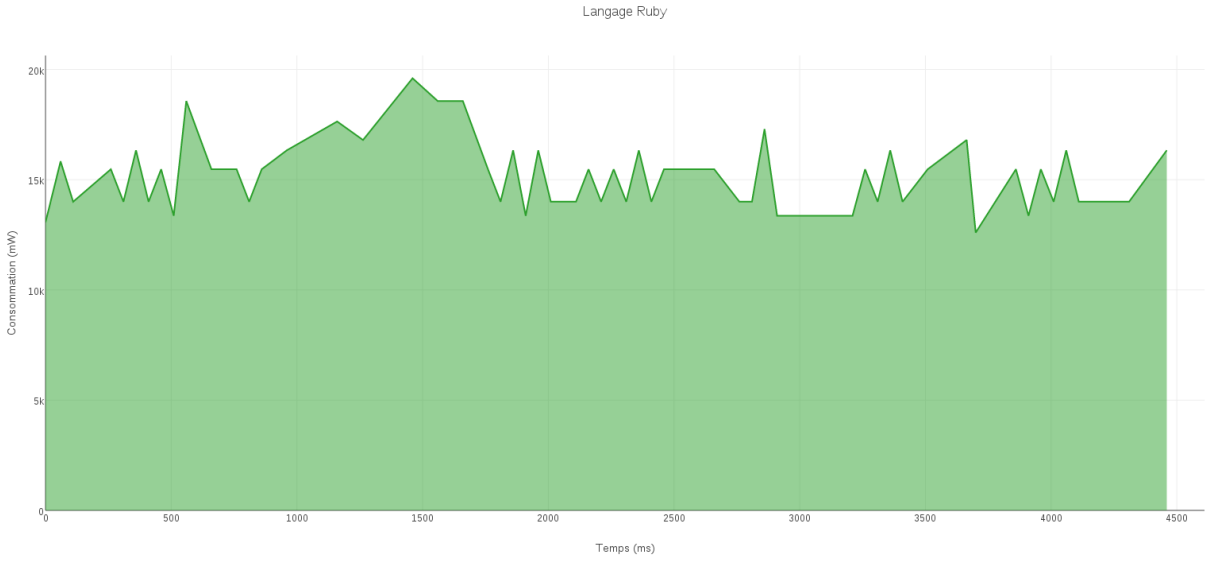

Figure 2.5: Consommation du QuickSort en C Ruby

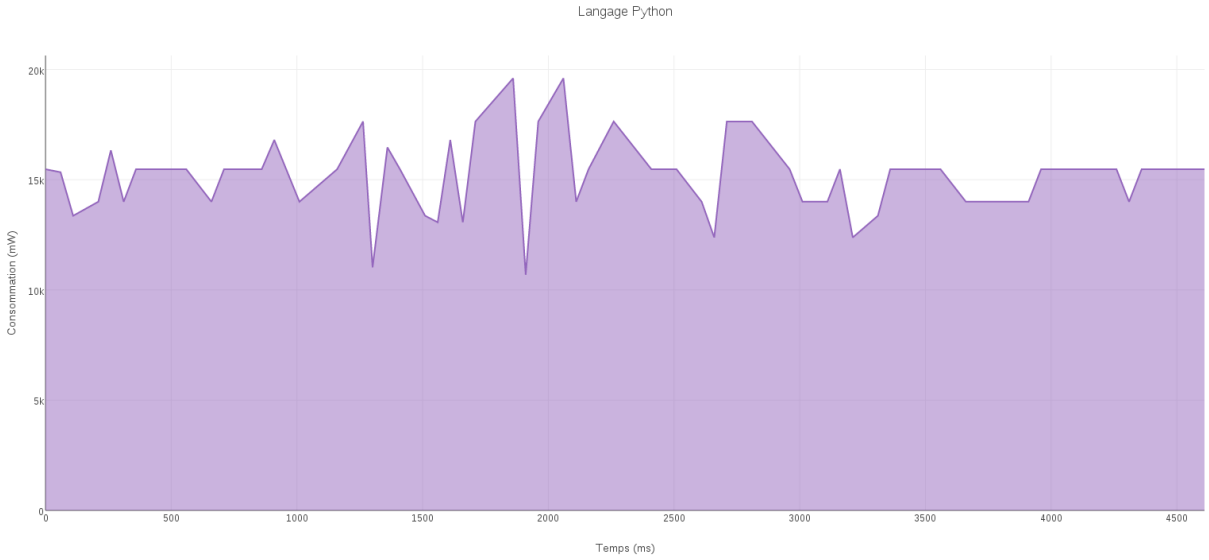

Figure 2.6: Consommation du QuickSort en C Pyhton

Cependant, ces graphiques ne représentent pas la valeur qui nous intéresse : le coût total en énergie pour une certaine exécution. En effet pour deux programmes ayant une consommation énergétique moyenne identique, le plus long à exécuter sera le plus couteux. Une telle valeur correspond au Joule, qui est la consommation en W/s. Le classement par performance est répertorié dans le tableau 2.1.

A partir de ces résultats, nous pouvons en déduire que Java est le plus performant pour cette implémentation du QuickSort. En effet, bien que sa consommation soit légèrement supérieure à celle du C ainsi que celle du C optimisé, la rapidité de son exécution permet de compenser, de sorte que sa consommation en Joule soit plus faible.

A l'opposé, nous retrouvons Python, qui malgré une consommation moyenne proche de celle de Java, possède le temps d'exécution le plus long, ce qui se traduit au final par la consommation énergétique totale la plus élevée.

Algorithme	temps d'exécution (ms)	consommation moyenne (mW)	consommation (J)
Java	760	7552.99	5.74
C optimisé	850	7449.55	6.03
C	1500	7459.87	11.19
Perl	3650	7933.84	28.96
Ruby	4460	7615.42	33.96
Python	4610	7580.87	34.95

Table 2.1: Tableau comparatif des différentes implémentations

Conclusion

Pour conclure, les résultats obtenus montrent que la consommation énergétique en Watt ne suffit pas pour déterminer le langage ayant la meilleure performance d'un point de vue énergétique. En effet, il est plus pertinent de prendre en compte la consommation en Joule, car cette dernière prend en compte le temps d'exécution.

Contrairement aux idées reçues, il apparait que Java soit le langage le plus performant parmi ceux testés pour l'implémentation du QuickSort.