ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KHOA HỌC MÁY TÍNH

Phân tích và thiết kế thuật toán

CS112.N21.KHTN

Thiết kế thuật toán song song

Merge sort parallel

MSSV	Họ và tên
21521109	Trần Hoàng Bảo Ly
21520800	Lê thu Hà

TP. HỒ CHÍ MINH, 2023

Mục Lục

Phần 1. Giới thiệu về thuật toán song song ..... - 3 -
1.1. Xử lý song song là gì? ..... - 3 -
1.2. Thuật toán song song là gì? ..... - 3 -
Phần 2. Thuật toán Merge sort ..... - 3 -
Phần 3. Thiết kế thuật toán Merge sort song song ..... - 5 -
3.1. Giới thiệu ..... - 5 -
3.2. Cài đặt với 2 process chạy song song ..... - 5 -
3.2.1. Đánh giá về hiệu suất so với cài đặt thông thường. ..... - 6 -
3.3. Cài đặt với k process chạy song song. ..... - 7 -
3.4. Sử dụng thuật toán Quick sort để sắp xếp mảng con ..... - 10 -
3.4.1. Cài đặt với k luồng song song ..... - 10 -
3.4.2. Cài đặt với 2 luồng song song ..... - 11 -
Phần 4. Tài liệu tham khảo ..... - 11 -
Phần 5. Các liên kết ..... - 11 -

Phần 1. **Giới thiệu về thuật toán song song**

1.1. Xử lý song song là gì?

Xử lý song song là quá trình xử lý đồng thời một tập các lệnh, qua đó giảm thời gian tính toán để hoàn thành một công việc nào đó.

1.2. Thuật toán song song là gì?

Thuật toán (trong khoa học máy tính) là một tập hợp hữu hạn các hướng dẫn được xác định rõ ràng, có thể thực hiện được bằng máy tính, thường để giải quyết một lớp vấn đề hoặc để thực hiện một phép tính.

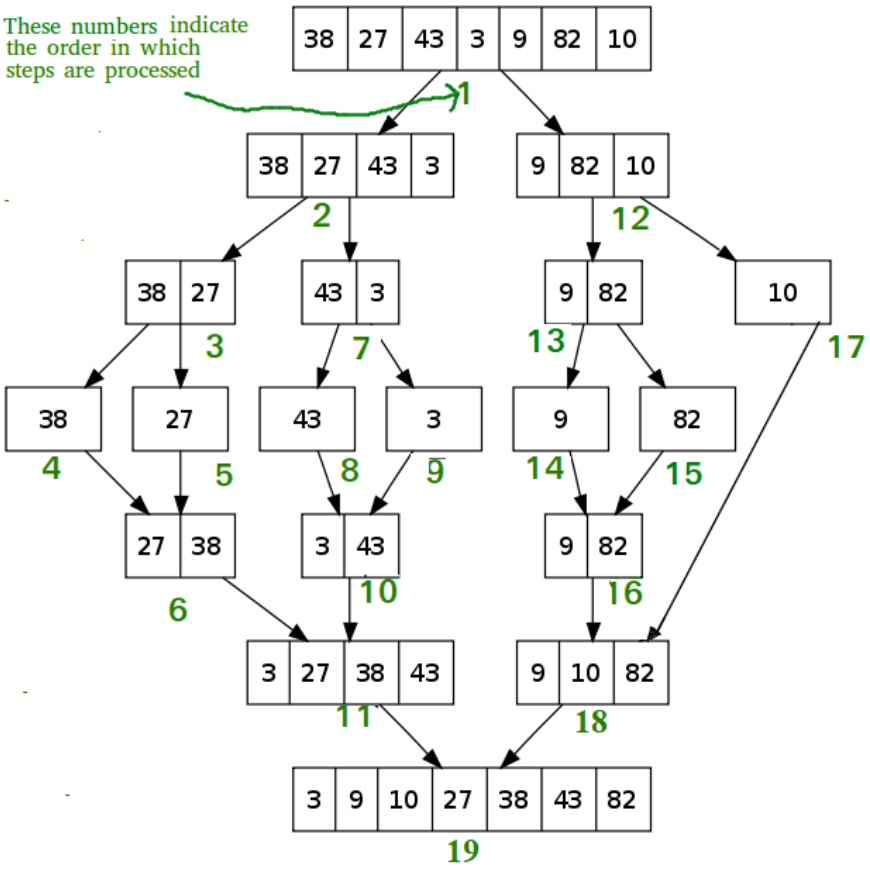
Dựa vào cách thực hiện chúng ta có thể chia thuật toán thành 2 loại:

- **Thuật toán tuần tự:** Là thuật toán mà các bước được thực hiện tuần tự theo thời gian để giải quyết một bài toán.
- **Thuật toán song song:** Là thuật toán mà bài toán được chia nhỏ thành các bài toán con, các bài toán con được giải quyết một cách song song, sau đó được tập hợp lại thành lời giải cho bài toán cuối cùng. Đây cũng chính là thuật toán được sử dụng cho ngày hôm nay.

Việc thực hiện thuật toán song song không hề đơn giản, cần yêu cầu chặt chẽ giữa phần cứng và phần mềm. Một vài khó khăn khi thực hiện thuật toán song song, thứ tự công việc, chia nhỏ bài toán, vùng tranh chấp dữ liệu,... Tuy nhiên để tối ưu hóa được hiệu suất thuật toán chúng ta buộc phải thực hiện nó.

Phần 2. **Thuật toán Merge sort**

Thuật toán Merge sort (sắp xếp trộn) là thuật toán sắp xếp bằng cách chia mảng thành các mảng con nhỏ hơn, sắp xếp các mảng con đó sau đó ghép lại thành mảng đã được sắp xếp. Đây là thuật toán sắp xếp cho độ ổn định cao với độ phức tạp về thời gian ở cả 3 trường hợp tốt nhất, trung bình và tệ nhất đều là O($nlogn$), và độ phức tạp về bộ nhớ là O(n) (đối với cài đặt tối ưu).



Hình 1: Mô tả thuật toán Merge sort hoạt động (nguồn: geeksforgeeks.org)

Chúng ta đã có bài phân tích và so sánh thuật toán này với các thuật toán sắp xếp khác ở đây.

Dưới đây là phần cài đặt khác cho thuật toán Merge sort với ngôn ngữ python.

```
import numpy as np

def Merge(*args):
    res = []
    i = 0
    j = 0
    L,R = args[0] if len(args) == 1 else args
    while(i<len(L) and j<len(R)):
        if (L[i] < R[j]):
            res.append(L[i])
            i+=1
        else:
            res.append(R[j])
            j+=1
    if (i<len(L)):
        res = np.append(res, L[i:])
    if (j<len(R)):
        res = np.append(res, R[j:])
    return res
```

```
def MergeSort(A):
    if (len(A) <=1):
        return A
    mid = int(len(A)/2)
    L = MergeSort(A[0:mid])
    R = MergeSort(A[mid:])
    return Merge(L,R)
```

Phần 3. **Thiết kế thuật toán Merge sort song song**

3.1. Giới thiệu

Như chúng ta đã biết thuật toán Merge sort là thuật toán sắp xếp từng phần. Vậy nên ý tưởng của chúng ta là sẽ sử dụng mỗi luồng dữ liệu chạy song song để thực hiện sắp xếp. Lúc này thuật toán được cài đặt có thể trông như thế này.

```
import multiprocessing
def MergeSortParallel(A):
    if (len(A) <=1):
        return A
    mid = int(len(A)/2)
    with multiprocessing.Pool(2) as pool:
        L = pool.apply_async(MergeSortParallel, [A[0:mid]])
        R = pool.apply_async(MergeSortParallel, [A[mid:]])
        L = L.get()
        R = R.get()
    return Merge(L,R)
```

Cứ mỗi mảng con được tách ra làm hai ta lại tiến hành khởi chạy một process mới, tuy nhiên cần phải biết rằng với cài đặt merge sort thông thường, việc phân chia sẽ được thực hiện đến khi mảng con chỉ còn một phần tử, điều đó đồng nghĩa với việc sẽ cần đến n process với n là số phần tử trong mảng. Có vài điều cần phải biết như sau:

- Số lượng process trong máy tính là giới hạn và thường là con số nhỏ.
- Chi phí cho việc tạo và quản lý một process là không nhỏ, chúng ta không thể bỏ qua nó.
- Với cài đặt Merge sort thông thường phần lớn thời gian dành cho giai đoạn merge chứ không phải là giai đoạn tách chiếm n trong O(nlogn)

Vậy nên với cài đặt trên là không hợp lý. Có vẻ một hướng tiếp cận hợp lý hơn là chia mảng thành k mảng con, thực hiện sắp xếp trên k mảng con đó sao đó ghép lại thành một mảng đã được sắp xếp.

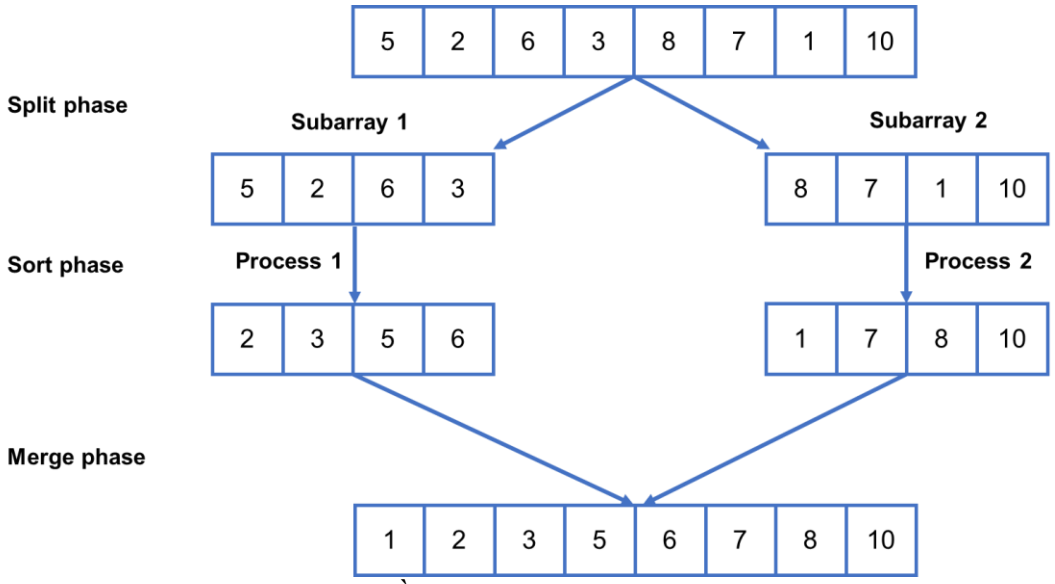
Câu hỏi đặt ra: k là bao nhiêu?

3.2. Cài đặt với 2 process chạy song song

Ta đều biết cận dưới của k là 1, đầu tiên thử cài đặt với k bằng 2.

Vậy nên hãy thử thiết kế thuật toán merge sort với 2 process chạy xong xong cho 2 mảng con.

Dưới đây là mô phỏng thuật toán merge sort với 2 process xử lý song song (về thuật toán merge sort ta đã biết ở các phần trước nên không cần phải nhắc lại cách nó hoạt động, trong bài viết này chỉ đề cập đến xử lý song song).



Với ý tưởng trên thuật toán gồm các bước sau đây:

Bước 1: Chia mảng thành 2 mảng con.

Bước 2: Thực hiện sắp xếp trên 2 mảng con song song (có thể sử dụng thuật toán merge sort đã đươc cài đặt ở trên).

Bước 3: Sử dụng hàm merge để merge 2 mảng con vừa được sắp xếp.

Dưới đây là cài đặt dựa trên ý tưởng trên bằng ngôn ngữ python.

```
import multiprocessing
import MSPL
def MergeSortParallelVer0(A):
    if (len(A) <=1):
        return A
    mid = int(len(A)/2)
    with multiprocessing.Pool(2) as pool:
        L = pool.apply_async(MSPL.MergeSort, [A[0:mid]])
        R = pool.apply_async(MSPL.MergeSort, [A[mid:]])
        L = L.get()
        R = R.get()
    return Merge(L,R
```

Chi tiết về cài đặt trên và các vấn đề xoay quanh được trình bày rõ ràng trong notebook.

3.2.1. Đánh giá về hiệu suất so với cài đặt thông thường.

Hàm đo thời gian thực hiện thuật toán:

```
import time
def Mesuare(arr , func):
    tic = time.time()
    arrS = func(arr)
    toc = time.time()
    MS_time = (toc-tic)*1000
    print(f'{func.__name__}, time used {MS_time} (ms) \n')
    return MS_time
```

Đánh giá trên mảng có 1,000,000 phần tử, thu được kết quả như sau:

```
MergeSortParallelVer0, time used 6433.24613571167 (ms)
MergeSort, time used 11197.022199630737 (ms)
Parrallel version faster 1.740493362670334 times
```

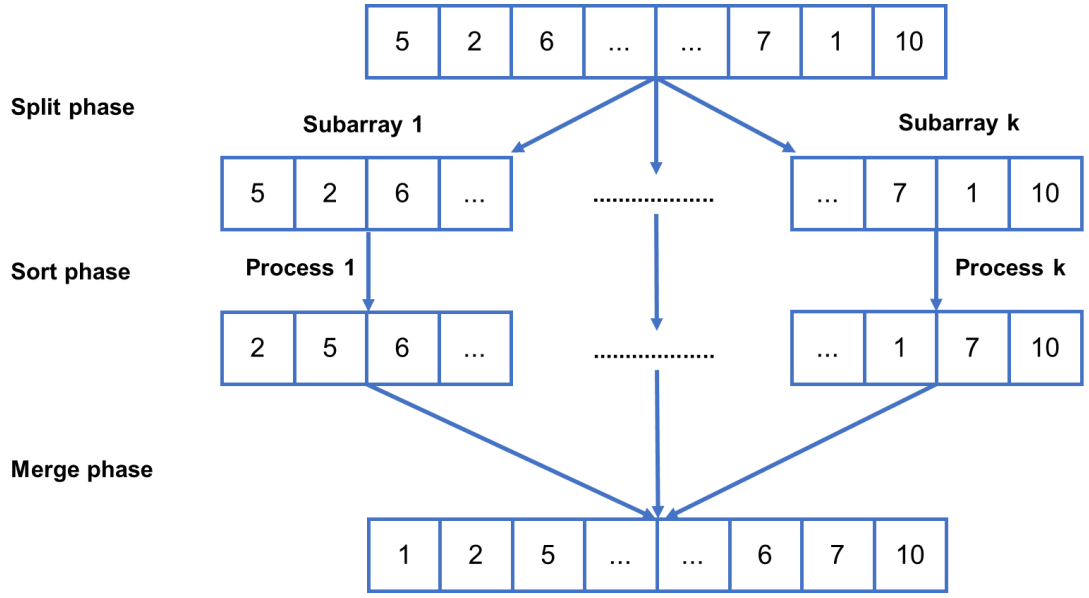
Cài đặt ở trên đã đạt được tốc độ gấp ≈ 1.74 lần so với merge sort thông thường. Tuy nhiên chúng ta có thể tận dụng tối đa hiệu suất của một máy tính hay thậm chí là một hệ thống máy tính với chỉ 2 luồng được chạy song song không?

3.3. Cài đặt với k process chạy song song.

Trong thực tế chúng ta muốn tăng tốc độ xử lý bằng cách tận dụng tối đa số process mà một máy tính hay hệ thống máy tính có thể cung cấp, nên một cài đặt khác sẽ hợp lý hơn. Lúc này k tối đa phụ thuộc vào từng máy, là số process tối đa mà mỗi máy tính hoặc hệ thống máy tính có. Hàm `multiprocessing.cpu_count()` sẽ trả về số process tối đa mà máy tính có thể cung cấp.

Lúc này việc của chúng ta sẽ chia mảng thành k mảng con nhỏ với (k = cpu_count()). Sắp xếp mỗi mãng con song song và lần lượt merge các mảng con lại thành một mảng cuối cùng đã được sắp xếp.

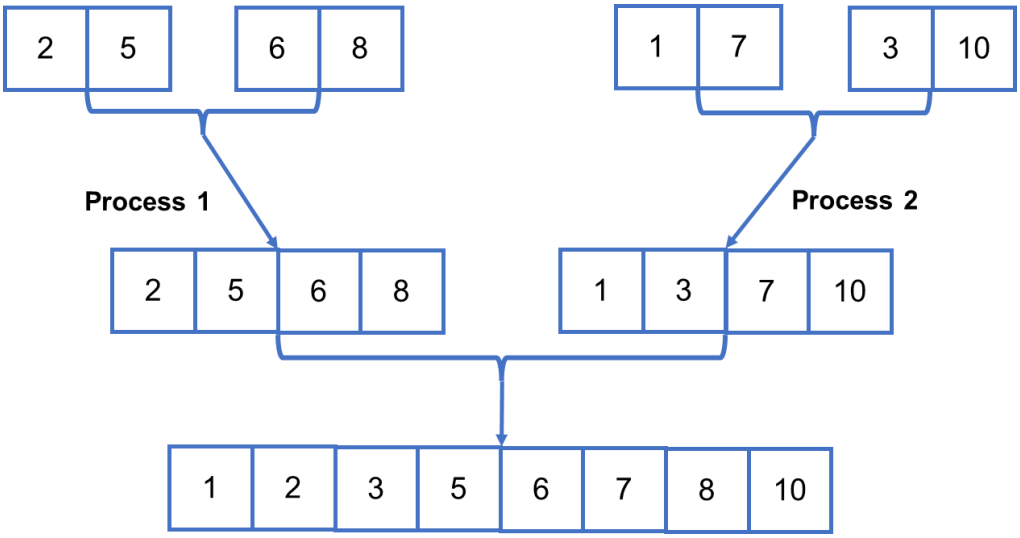
Mô phỏng thuật toán lúc này sẽ trông như thế này.



Tuy nhiên với k luồng được thực hiện cùng lúc có một vấn đề ở merge phase là phải xử lý quá trình merge như thế nào? Liệu tuần tự merge từng phần có phải giải pháp tối ưu?

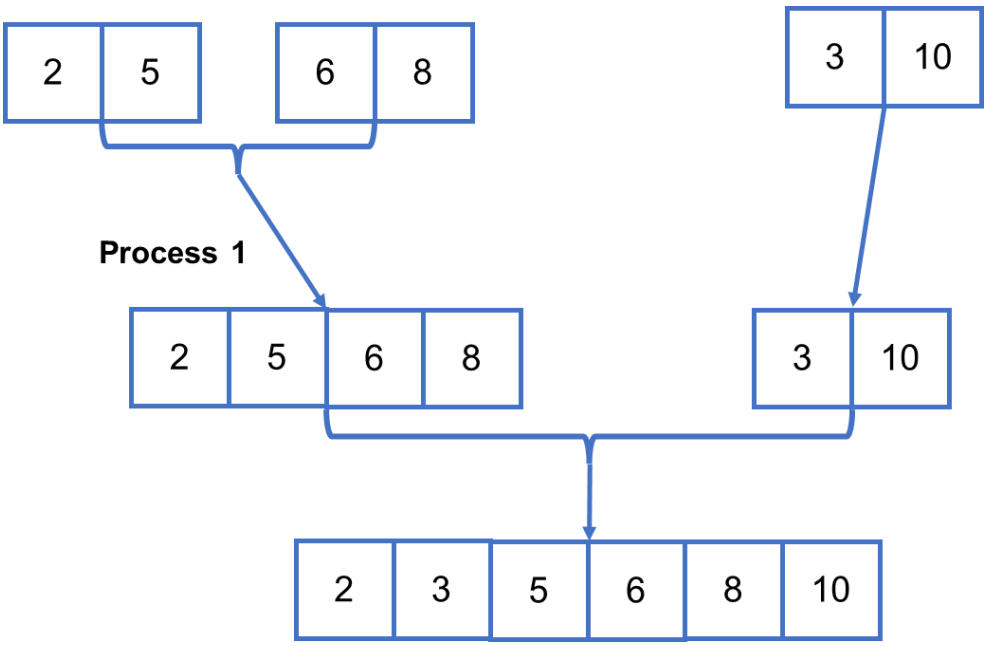
Câu trả lời là không? Ta có thể tận dụng xử lý đa luồng để thực hiện merge các cặp song song với nhau đó là ý tưởng của thuật toán này. Như vậy việc thực hiện song song không chỉ ở sort phase mà cả ở merge phase.

Mô phỏng merge phase với 4 mảng con.



Nếu số mảng con là số lẻ th trước tiên ta bỏ qua mảng con lẻ cuối cùng tiến hành merge các cặp còn lại sau đó ghép mảng con vào tập mảng con sau khi merge, lặp lại các bước trên.

Mô phỏng merge phase với 3 mảng con:



Từ các nội dung nêu trên ta tóm tắt được các bước của thuật toán như sau:

Bước 1: Chia mảng ban đầu thành k mảng con với k là số process tối đa mà máy tính có thể cung cấp.

Bước 2: Tiến hành merge k mảng con trên k luồng chạy song song.

Bước 3 Tiến hành merge các mảng con thành từng cặp mỗi cặp chạy một process song song, thực hiện $log_2 n$ lần đến khi chỉ còn một mảng con đã được sắp xếp duy nhất.

Cài đặt bằng python

```
def MergeSortParallel(A):
    n = len(A)
    if n <=1:
        return A
    n_process = multiprocessing.cpu_count()

    A = np.array_split(A, n_process)

    pool = multiprocessing.Pool(n_process)
    A = pool.map(MSPL.MergeSort, A)
    while len(A)>1:
        odd_part = []
        if len(A) % 2 != 0:
            odd_part = A.pop()
        A = [(A[i],A[i+1]) for i in range(0,len(A),2)]
        if len(odd_part) == 0:
            A = pool.map(MSPL.Merge, A)
        else:
            A = pool.map(MSPL.Merge, A) + odd_part
    return A[0]
```

Cùng tiến hành đo thời gian thực hiện của các thuật toán với mảng có 1,000,000 phần tử.

Thu được kết quả dưới đây.

```
MergeSortParallel, time used 4734.275102615356 (ms)

MergeSort: Parallel version !!! FASTER !!! than normal version 2.36509749
79137067 times!!!
```

Chúng ta đã tăng hiệu suất của thuật toán thông thường lên ≈ 2.36 Lần, tốt hơn nhiều so với chỉ 2 process chạy song song, cùng thử với mảng có 2,000,000 phần tử.

```
arr = np.random.randint(0,2000000, size=2000000)
MSP_time = Mesuare(arr, MergeSortParallel)
MS_time = Mesuare(arr, MergeSort)
print(f"Parrallel version faster {float(MS_time)/MSP_time} times")
```

Thu được kết quả:

```
MergeSortParallel, time used 9430.230140686035 (ms)
MergeSort, time used 23615.66138267517 (ms)
Parrallel version faster 2.504250800920238 times
```

Lần này tốc độ của cài đặt parallel tăng 2.5 lần so với cài đặt thông thường. Và hiệu suất sẽ tiếp tục tăng nếu như số phần tử trong mảng tiếp tục tăng, tối đa là số process mà máy tính sở hữu nếu như số lượng phần tử tiến đến vô cùng (sự thật thì chẳng ai muốn điều đó xảy ra cả).

Điều gì sẽ xảy ra nếu mỗi mảng con ta sử dụng thuật toán quick sort để sắp xếp?

3.4. Sử dụng thuật toán Quick sort để sắp xếp mảng con

3.4.1. Cài đặt với k luồng song song

```
def MergeQuickSortParallel(A):
    n = len(A)
    if n <=1:
        return A
    n_process = multiprocessing.cpu_count()

    A = np.array_split(A, n_process)

    pool = multiprocessing.Pool(n_process)
    A = pool.map(MSPL.QuickSort, A)
    while len(A)>1:
        odd_part = []
        if len(A) % 2 != 0:
            odd_part = A.pop()
        A = [(A[i],A[i+1]) for i in range(0,len(A),2)]
        if len(odd_part) == 0:
            A = pool.map(MSPL.Merge, A)
        else:
            A = pool.map(MSPL.Merge, A) + odd_part
    return A[0]
```

Cùng thử đo hiệu suất của cài đặt này, tương tự với mảng có 2,000,000 phần tử. Kết quả thu được

```
MergeQuickSortParallel, time used 9232.069969177246 (ms)

Merge Quick Sort faster 1.0214643273036685 times than old parallel version
Quick Sort faster 1.3511858348160557 times than merge quick parallel version
```

Hiệu suất không tăng đáng kể, tuy nhiên nó vẫn còn chậm hơn so với thuật toán quick sort thông thường. Tuy nhiên ta có một vài nhận xét như sau:

Chi phí cho việc tạo và quản lý các process mới là không nhỏ, hiệu quả thực sự rõ ràng khi và chỉ khi số lượng phần tử trong mảng là đủ lớn, có thể vào khoảng 5,000,000 sẽ có sự khác biệt, tuy nhiên vì thời gian chờ đợi khá lâu (python khá chậm

nếu không sử dụng các hàm built-in) nên việc test có thể thực hiện sau. Trong trường hợp này có lẽ việc cài đặt chỉ với 2 luồng song song sẽ tối ưu hơn.

3.4.2. Cài đặt với 2 luồng song song

Trong trường hợp này (mảng có 2,000,000 phần tử), chúng ta thử lại cài đặt với 2 luồng song song, mỗi luồng sử dụng thuật toán quick sort.

```
def MergeQuickSortParallelVer2(A):
    if (len(A) <=1):
        return A
    mid = int(len(A)/2)
    with multiprocessing.Pool(2) as pool:
        L = pool.apply_async(MSPL.QuickSort, [A[0:mid]])
        R = pool.apply_async(MSPL.QuickSort, [A[mid:]])
        L = L.get()
        R = R.get()
    return Merge(L,R)
```

Tiếp tục đo hiệu suất. Kết quả thu được:

```
MergeQuickSortParallelVer2, time used 6529.356956481934 (ms)

Merge Quick Sort ver 2 faster 1.4442816043813163 times than old parallel version
Quick Sort faster 0.9556225916301525 times than merge quick parallel version 2
```

Điều đáng ngạc nhiên là chúng ta đã đạt được hiệu suất x1.44 lần so với cài đặt song song ở trên và nó thậm chí còn nhanh hơn cả thuật toán quick sort thông thường.

Phần 4. **Tài liệu tham khảo**

- Blelloch, G. E., & Maggs, B. M. (1996). Parallel algorithms. *ACM Computing Surveys*, *28*(1), 51–54. https://doi.org/10.1145/234313.234339
- Lumunge, E. (2021, November 18). *Parallel Merge Sort*. OpenGenus IQ: Computing Expertise & Legacy; OpenGenus IQ: Computing Expertise & Legacy. https://iq.opengenus.org/parallel-merge-sort/
- *Merge Sort Algorithm*. (2013, March 15). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/merge-sort/

Phần 5. **Các liên kết**

Liên kết đến Github