Nhóm 3:

Mai Nhất Tùng - 21520523

Hà Trọng Tài - 21520436

Nguyễn Ngọc Nhung – 21521248

PARALLEL MERGE SORT

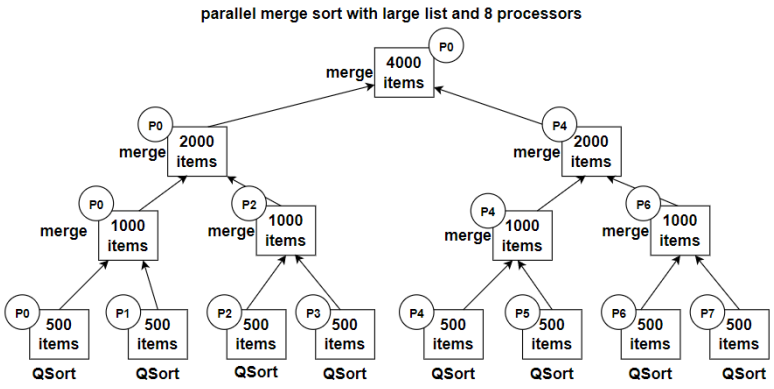
I. Abstraction:

Nhóm chúng em đã chọn cấu trúc dữ liệu là thực hiện sắp xếp cho mảng số nguyên tăng dần.

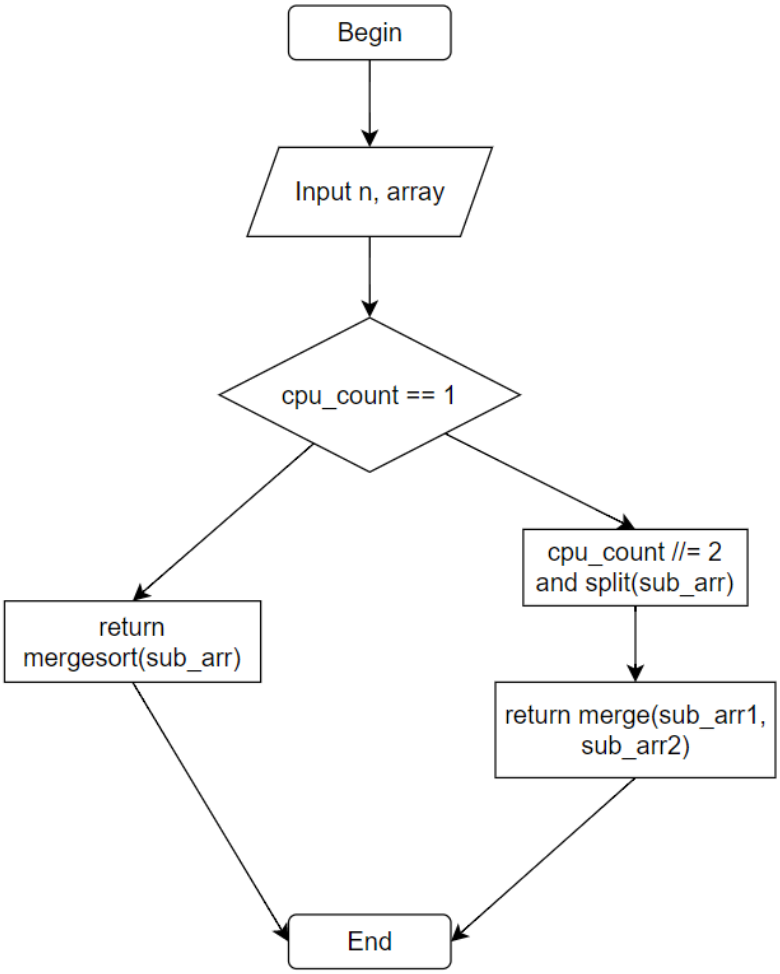
1. Merge sort:
   - Merge sort hoạt động trên nguyên tắc chia để trị.
   - Chia nhỏ một mảng thành 2 mảng con bằng nhau cho đến khi mỗi mảng con chỉ chứa một phần tử duy nhất. Cuối cùng, tất cả các mảng con đó được hợp nhất (merge) để sắp xếp.
2. Parallel Algorithm:
   - Thuật toán song song là thuật toán có thể thực hiện từng mẩu nhỏ công việc tại một thời điểm trên nhiều thiết bị xử lý khác nhau sau đó tổng hợp lại để lấy kết quả cuối cùng

II. Pattern Recognition:

- Mấu chốt của việc thiết kế thuật toán song song là tìm ra các bước trong thuật toán có thể thực hiện đồng thời. (Số lượng processors sẽ ảnh hưởng đến performance của thuật toán song song)
- Đối với nhóm chúng em thì chúng em sẽ thiết kế thuật toán song song để thực hiện sắp xếp dãy tăng dần theo thuật toán merge sort bằng cách:
  - Giả sử chúng ta có p processors, n phần tử thuộc mảng
  - Với trường hợp p >= n thì mỗi processors sẽ được phân chia xử lý với 1 phần tử. Sau khi xử lý xong sẽ gộp 2 tiến trình lân cận lại với nhau.
  - Với trường hợp p < n thì, thì mỗi tiến trình p sẽ được phân chia xử lý với n/p phần tử. Mỗi processor sẽ thực hiện merge sort trên n/p phần tử ấy, sau đó chúng ta sẽ thực hiện merge 2 tiến trình lân cận lại với nhau cho đến hết.
- VD: Trường hợp mảng có 4000 phân tử, tuy nhiên chỉ có 8 processors, chúng ta sẽ thực hiện chia processors như sau:



III. Algorithm Designed:



IV. Time and space complexity:

- Merge sort:
  - Time complexity: O(nlogn)
  - Space complexity: O(n)
- Parallel merge sort:
  - Time complexity: $O(nlog(n)^2/cpu\_count)$ (Nằm trong paper có ở phần reference dưới)
  - Space complexity: O(n)

V. Programming:

```python
def merge(L,R):
    i = j = k = 0
    res = []

    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            res.append(L[i])
            i += 1
        else:
            res.append(R[i])
            j += 1

    res.extend(L[i:])
    res.extend(R[j:])

    return res



def mergeSort(arr):

    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    L, R = mergeSort(arr[:mid]), mergeSort(arr[mid:])
    return merge(L,R)



def parallelMergeSort(arr,cpu_count=multiprocessing.cpu_count()):
        mid = len(arr) // 2
        full = [arr[mid:], arr[:mid]]
        if cpu_count == 1:
            return mergeSort(arr)

        else:
            result = []
            with futures.ProcessPoolExecutor(cpu_count) as p:
                cpu_count = cpu_count // 2
                if cpu_count > 0:

                    cpu_countlist = [cpu_count,cpu_count]
                    future = p.map(parallelMergeSort,full,cpu_countlist)
                    for value in future:
                        result.append(value)

            return merge(result[0],result[1])
```

- Kết quả thực nghiệm khi thực hiện sort một mảng 1000000 phần tử:

```
Merge Sort: 2.78853440284729 seconds
-------------------------------------------
Parallel Merge sort 1.5295667648315 43 seconds
```

REFERENCE:

(1) Paper: “Approaches to Parallelization of Merge Sort in Python”: 2211.16479.pdf (arxiv.org)
(2) Parallel Merge Sort: Parallel Merge Sort (opengenus.org)