Trường Đại học Công nghệ Thông tin
Đại học Quốc gia Hồ Chí Minh

MERGE SORT PARALLEL

CS112.N21.KHTN

Phan Trường Trí - 21520117
Nguyễn Đức Nhân - 21520373

Ngày 10 tháng 4 năm 2023

TÓM TẮT NỘI DUNG

Đây là bài báo cáo về việc lập trình song song cho việc sắp xếp tăng dần bằng thuật toán Merge Sort. Là bài tập môn phân tích và thiết kế thuật toán (CS112.N21.KHTN).

Mục lục

1 MERGE SORT

1.1 Giới thiệu

Merge sort là một thuật toán sắp xếp theo cơ chế chia để trị. Ý tưởng cơ bản của nó là chia đoạn cần sắp xếp thành 2 đoạn con sau đó sắp xếp 2 đoạn con này và cuối cùng là hợp nhất hai đoạn con đã sắp xếp thành một đoạn duy nhất đã được sắp xếp

1.2 Thuật toán

Hàm $merge(arr, L, M, R)$ trong thuật toán dưới đây có chức năng gộp 2 dãy $arr[L \dots M]$ và $arr[M+1 \dots R]$ đều đã được sắp xếp thành một dãy $arr[L \dots R]$ cũng đã được sắp xếp. Hàm này có độ phức tạp là $O(n)$ với n là độ dài của dãy ($n = R - L + 1$)

Algorithm 1 $mergeSort(arr, L, R)$

Input: Mảng $arr[]$, các vị trí L, R
Output: Mảng $arr[]$ có các vị trí từ L đến R đã được sắp xếp

```
if L == R then
    return
else
    M ← (L + R)/2
    mergeSort(arr, L, M)
    mergeSort(arr, M + 1, R)
    merge(arr, L, M, R)
end if
return
```

1.3 Ví dụ

Ví dụ khi dùng merge sort sắp xếp lại dãy $arr[] = [16, 10, 8, 15, 14, 30, 29]$ được mô tả trong **Hình 1**

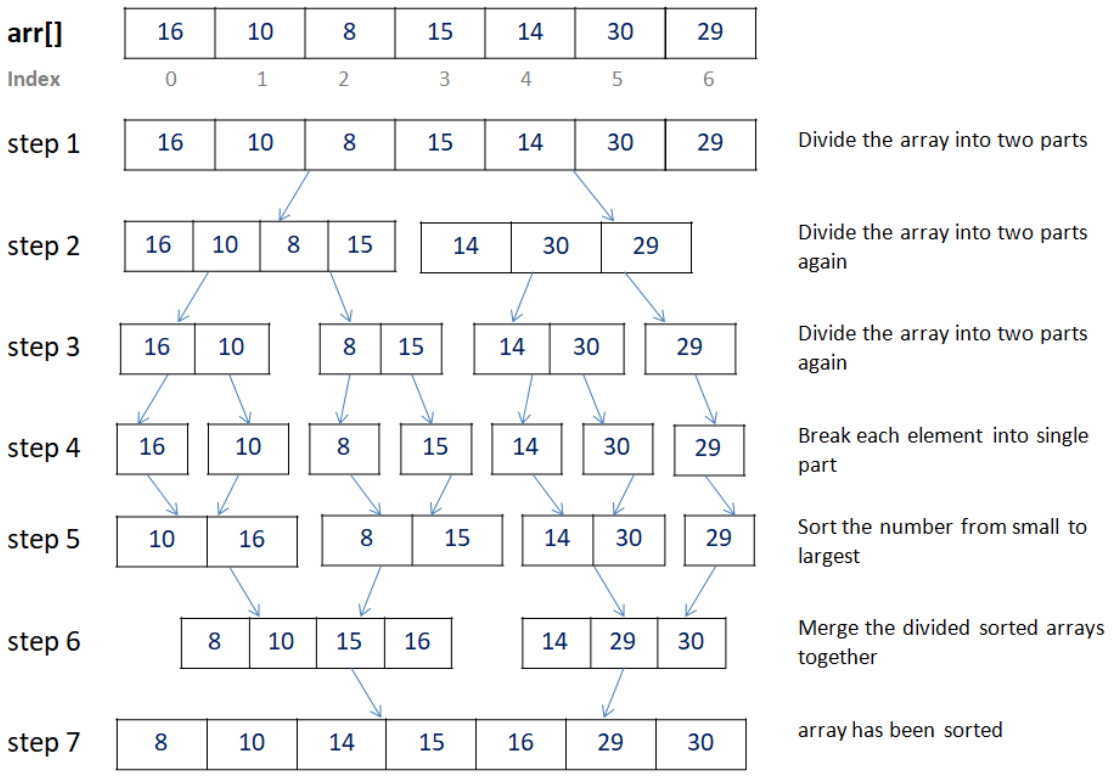
.

1.4 Đánh giá độ phức tạp

Gọi T_n là số phép tính khi thực hiện merge sort với dãy gồm n phần tử, ta có:

$$T_n = \begin{cases} 2 \times T_{n/2} + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases} \approx nlogn$$

$T_n = 2 \times T_{n/2} + n$ vì dãy gồm n phần tử sẽ chia thành hai dãy có độ dài $n/2$ và thực hiện thuật toán trên hai mảng này ($2 \times T_{n/2}$), sau đó gọi hàm $merge(arr, L, M, R$ tốn thêm $O(n)$. Khai triển kết quả trên, ta sẽ thấy được thuật toán merge sort có độ phức tạp là $O(nlogn)$ với n là số phần tử của dãy.



Hình 1: Thuật toán merge sort

2 CẢI TIẾN - THỰC HIỆN SONG SONG

2.1 Parallel Merge Sort

Parallel merge sort thực hiện hai hàm $mergeSort(arr, L, M)$ và $mergeSort(arr, M+1, R)$ song song vì hai hàm này không ảnh hưởng tới nhau, điều này làm cho chương trình được chạy trên nhiều core hơn, giảm thời gian chạy của chương trình.

Algorithm 2 $parallelMergeSort(arr, L, R)$

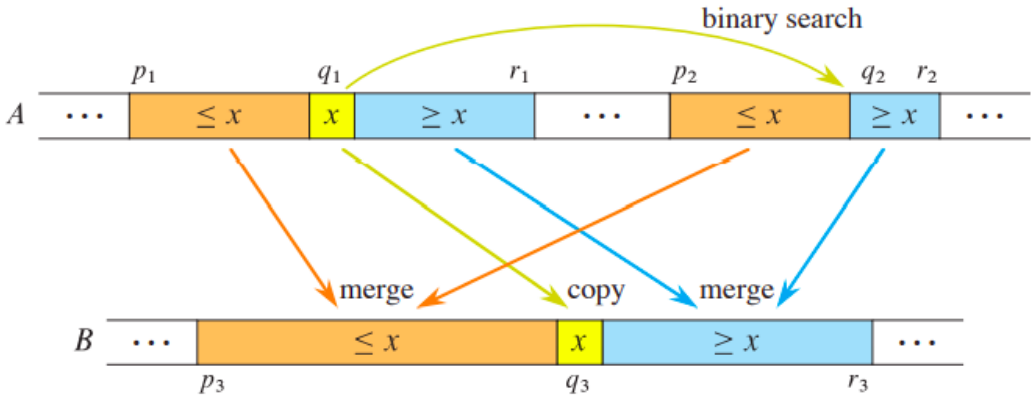
Input: Mảng $arr[]$, các vị trí L, R
Output: Mảng $arr[]$ có các vị trí từ L đến R đã được sắp xếp

```
if L == R then
    return
else
    M ← (L + R)/2
    spawn parallelMergeSort(arr, L, M)
    spawn parallelMergeSort(arr, M + 1, R)
    sync
    parallelMerge(arr, L, M, R)
end if
return
```

2.2 Parallel Merge

Ta sẽ cố gắng cải tiến thêm bằng cách thiết kế lại hàm $merge(arr, L, M, R)$ thành hàm $parallelMerge(arr, L, M, R)$ có sử dụng tính năng thực hiện song song. Ý tưởng chính của việc song song hóa hàm $merge$ là sử dụng một hàm $parallelMergeAux(A, p_1, r_1, p_2, r_2, B, p_3)$ để gộp hai đoạn đã sắp xếp $A[p_1 : r_1]$ và $A[p_2 : r_2]$ thành một đoạn đã sắp xếp trên mảng $B[p_3, r_3]$ với $r_3 = p_3 + (r_1 - p_1 + 1) + (r_2 - p_2 + 1) - 1$.
Hàm $parallelMergeAux$ sẽ được chạy song song bằng cách chia nhỏ bài toán thành hai phần theo các bước sau (xem hình minh họa để hiểu rõ hơn):

- Chọn vị trí $q_1 = (r_1 - p_1)/2$ là trung vị của dãy $A[p_1 : r_1]$
- Đặt $x = A[q_1]$
- Tìm vị trí $p_2 \leq q_2 \leq r_2$ nhỏ nhất thỏa mãn $A[q_2] \geq x$ (bằng binary search)
- Gộp hai đoạn $A[p_1 : q_1 - 1]$ và $A[p_2 : q_2 - 1]$ vào mảng các vị trí đầu tính từ p_3 của mảng B
- Copy $A[q_1]$ vào vị trí tiếp theo
- Gộp hai đoạn $A[q_1 + 1 : r_1]$ và $A[q_2 : r_2]$ vào các vị trí tiếp theo của mảng B



Hình 2: parallelMergeAux

Quá trình tìm q_2 được mô tả bằng thuật toán sau:

Algorithm 3 $findPosition(A, p, r, x)$

Input: Mảng A, số x, các vị trí p, r
Output: vị trí q nhỏ nhất thỏa mãn $p \leq q \leq r, A[q] > x$

```
low ← p
high ← r
ans ← high + 1
while low ≤ high do
  mid ← ⌈(low + high)/2⌉
  if A_mid ≥ x then
    ans ← mid
    high ← mid − 1
  else
    low ← mid + 1
  end if
end while
return ans
```

Algorithm 4 $parallelMergeAux(A, p_1, r_1, p_2, r_2, B, p_3)$

Input: Mảng A, B, các vị trí p_1, r_1, p_2, r_2, p_3, trong đó $A[p_1 : r_1], A[p_2 : r_2]$ đã được sắp xếp
Output: Mảng $B[p_3 :]$ chứa các phần tử của $A[p_1 : r_1], A[p_2 : r_2]$ đã được sắp xếp

```
if p1 > r1 and p2 > r2 then
  return
end if
if r1 − p1 < r2 − p2 then
  swap(p1, p2)
  swap(r1, r2)
end if
q1 ← ⌈(p1 + r1)/2⌉
x ← A[q1]
q2 ← findPosition(A, p2, r2, x)
q3 ← p3 + (q1 − p1) + (q2 − p2)
spawn parrallelMergeAux(A, p1, q1 − 1, p2, q2 − 1, B, p3)
B[q3] = x
spawn parrallelMergeAux(A, q1 + 1, r1, q2, r2, B, q3 + 1)
sync
```

Trong C++ có hỗ trợ **for parallel** nên ta có thể thực hiện song song quá trình copy từ mảng B về mảng A. Ta sẽ biểu diễn hàm $parralelMerge(A, L, M, R)$ như sau:

Algorithm 5 *parallelMerge*(A, L, M, R)

Input: Mảng A các vị trí L, M, R, trong đó $A[L : M], A[M+1 : R]$ đã được sắp xếp
Output: Mảng $A[L : R]$ chứa các phần tử của $A[L : M], A[M+1 : R]$ đã được sắp xếp

```
new array B size R − L + 1
parallelMergeAux(A, L, M, M + 1, R, B, 0)
for parallel i = 0 to R − L do
    A[L + i] = B[i]
end for
delete B
```

3 THỰC NGHIỆM

3.1 Chương trình không sử dụng mergeParallel

3.1.1 *Chương trình*

Chương trình chạy bằng thuật toán merge sort parallel như trên nhưng hàm *merge* là hàm *merge* bình thường (là hàm *mergeArray* trong chương trình). Chương trình đầy đủ có thể xem tại đây.
Chương trình sinh ra một dãy bằng phương pháp random, dùng dãy đó chạy thuật toán merge sort thông thường và merge sort song song. Thời gian chạy được đo bằng **std::chrono::high_resolution_clock::time_point**.
Sau mỗi lần sort có dùng hàm *checkSorted* để kiểm tra tính đúng đắn.
Biến *MIN_TO_THREAD* dùng để hạn chế hao phí thời gian khi tạo mới thread cho các chương trình nhỏ.

Listing 1: Parallel Merge Sort

```
#include <bits/stdc++.h>
using namespace std;
const int MIN_TO_THREAD = 1000;
template<typename T>
void mergeArray(T arr[], int leftPos, int midPos, int rightPos) {
   int arrSize = rightPos - leftPos + 1;
   T* tempArr = new T[arrSize];
   for (int i = 0, leftId = leftPos, rightId = midPos + 1; i < arrSize;
      i++) {
      if (leftId > midPos)
         tempArr[i] = arr[rightId++];
      else if (rightId > rightPos)
         tempArr[i] = arr[leftId++];
      else if (arr[leftId] < arr[rightId])
         tempArr[i] = arr[leftId++];
      else
         tempArr[i] = arr[rightId++];
   }
   for (int i = leftPos; i <= rightPos; i++)
      arr[i] = tempArr[i - leftPos];
   delete[] tempArr;
```

```
}

template<typename T>
void mergeSort(T arr[], int leftPos, int rightPos) {
    if (leftPos == rightPos)
        return;
    int midPos = (rightPos + leftPos) / 2;
    mergeSort<T>(arr, leftPos, midPos);
    mergeSort<T>(arr, midPos + 1, rightPos);
    mergeArray<T>(arr, leftPos, midPos, rightPos);
}

template<typename T>
void mergeSortParallel(T arr[], int leftPos, int rightPos) {
    if (leftPos == rightPos)
        return;

    int midPos = (rightPos + leftPos) / 2;
    if (rightPos - leftPos + 1 >= MIN_TO_THREAD) {
        std::thread left(mergeSortParallel<T>,arr, leftPos, midPos);
        std::thread right(mergeSortParallel<T>,arr, midPos + 1, rightPos);
        left.join();
        right.join();
    } else {
        mergeSort<T>(arr, leftPos, midPos);
        mergeSort<T>(arr, midPos + 1, rightPos);
    }

    mergeArray<T>(arr, leftPos, midPos, rightPos);
}

const int N = 10241024;
string checkSorted(int a[]) {
    bool checkSorted = true;
    for (int i = 0; i + 1 < N; i++)
        if (a[i] > a[i + 1])
            checkSorted = false;
    return "CHECK SORTED: " + (string)(checkSorted ? "TRUE":"FALSE") +
        "\n";
}
int main() {
    freopen("./Merge Sort parallel/OutFullParallel.txt", "w", stdout);
    srand(time(0));
    int* temp = new int[N];
    int* a = new int[N];
    for (int i = 0; i < N; i++) {
        temp[i] = rand();
        a[i] = temp[i]; // array a used twice
    }

    cout << "WITH N = " << N << endl;
```

```
    chrono::high_resolution_clock::time_point t1 =
        chrono::high_resolution_clock::now();
    mergeSort<int>(a, 0, N - 1);
    chrono::high_resolution_clock::time_point t2 =
        chrono::high_resolution_clock::now();
    cout << checkSorted(a);

    for (int i = 0; i < N; i++)
        a[i] = temp[i];

    chrono::high_resolution_clock::time_point t3 =
        chrono::high_resolution_clock::now();
    mergeSortParallel<int>(a, 0, N - 1);
    chrono::high_resolution_clock::time_point t4 =
        chrono::high_resolution_clock::now();
    cout << checkSorted(a);

    int duration_1 =
        std::chrono::duration_cast<std::chrono::microseconds>(t2 -
        t1).count();
    int duration_2 =
        std::chrono::duration_cast<std::chrono::microseconds>(t4 -
        t3).count();
    printf("NORMAL TOOK %d microseconds\n", duration_1);
    printf("PARALLEL TOOK %d microseconds\n", duration_2);

    delete[] a;
    delete[] temp;
}
```

3.1.2 *Kết quả*

Tính đúng đắn và thời gian chạy:

```
WITH N = 10241024
CHECK SORTED: TRUE
CHECK SORTED: TRUE
NORMAL TOOK 2550450 microseconds
PARALLEL TOOK 1639847 microseconds
```

Thời gian chạy khi song song hóa thuật toán merge sort thấp hơn so với merge sort thông thường. Hàm *mergeSortParallel* thể hiện được tính song song, report về core được sử dụng xem tại đây. Report về core đã sử dụng của thuật toán merge sort thông thường tại đây

3.2 Chương trình có sử dụng mergeParallel

Chương trình xem tại đây Theo lí thuyết từ sách (Introduction to Algorithms), nếu cải tiến hàm merge theo Algorithm 3, 4, 5 thì sẽ có thời gian chạy tốt hơn, nhưng code của sinh viên không cho ra được kết quả như vậy (chậm hơn rất nhiều so với thuật toán ở **3.1**.

3.2.1 *Kết quả*

```
WITH N = 1024102
CHECK SORTED: TRUE
CHECK SORTED: TRUE
NORMAL PARALLEL TOOK 202503 microseconds
FULL PARALLEL TOOK 897772 microseconds
```

Có thể là do sinh viên code sai !

4 THAM KHẢO

1. Introduction to Algorithms - Fourth Edition
2. Stackexchange