Adaptive Sorting Algorithm

Md Mehedi Hasan

February 7, 2018

The Algorithm: This hybrid sorting algorithm has three routines: amergesort, hybridsort and, sort. The hybridsort routine performs sorting for collections with big enough runs [Rocca and Cantone,] where amergesort is for small collection and sort is the contract routine with output iterator for returning the result to the main application.

This algorithm is based on a principle that it tries to find big runs in the collection to sort to take advantage of the presortedness, and falls back to another sorting routine amergesort to handle the sections of the collection to sort without big enough runs. The amergesort is an implementation of divide and conquer based mergesort [Vignesh and Pradhan, 2016].

The hybridsort routine runs through the collection while it is sorted in ascending or descending order, and computes the size of the current run. If the run is big enough, then hybridsort remembers the bounds of the run, to merge it in another step (it reverses the run first if it is sorted in descending order). If the run is not big enough, hybridsort just remembers its beginning and moves on to the next run. When it reaches a big enough run, it calls the fallback sorting routine amergesort to sort every element between the beginning of the section without big enough runs and the beginning of the current big enough run. For your information, algorithm pseudo-code is added here.

Once hybridsort has finished crossing the entire collection, there is only big sorted runs left. Then, hybridsort merges all the runs and leave a fully sorted collection. Finally, sort routine return the sorted result to the main application.

A run is considered big enough when its size is bigger than n / log n, where n is the size of the entire collection to sort. For

Algorithm 1 AdaptiveSorting

```
$first$ : the iterator pointing to the first element in the range (first, beyond) of the collection
$beyond$ : the iterator pointing to the last element
$result$ : the iterator for returning sorted collection to the application
$less$ : is the less demanding interface to compare two objects

procedure AMergeSort(first, beyond, less)
    var $n \leftarrow distance(first, beyond)$
    if the collection is not already sorted then
        if the collection contains more than 1 element then
            var $middle \leftarrow$ iterator to the element n/2 positions away from first element
            AMergeSort(first, middle, less)                ▷ recursive routine call
            AMergeSort(middle, beyond, less)               ▷ recursive routine call
            merge two consecutive sorted ranges
        end if
    end if
end procedure
```

```
procedure HybridSort(first, beyond, less)
    var $n \leftarrow distance(first, beyond)$
    if n is less then 128 then
        AMergeSort(first, beyond, less)        ▷ fall-back into AMergeSort routine for small collections
        Return
    end if
    var $limit \leftarrow n/floor(log(n))$        ▷ limit under which std::sort in c++ is used to sort a sub-sequence
    var runs                        ▷ declare an empty collection accessible from any direction
    var $begin \leftarrow beyond$                ▷ iterator pointing beginning of a partition
    var $current \leftarrow next(first)$              ▷ get to the next point of $first$
    while true do
        var $begin\_range \leftarrow current$           ▷ Beginning of the current sequence
        if distance between $next$ and $beyond$ greater than or equal to $limit$ then
            if $begin$ is equal to $beyond$ then
                $begin \leftarrow begin\_range$
            end if
            break
        end if
        advance $current$ by the $limit$ amount
        advance $next$ by the $limit$ amount
        var $current2 \leftarrow current$;
        var $next2 \leftarrow next$;
        if value in $next$ position is less than value in $current$ position then
            do
                –current
                –next
                if value in $current$ position is less than value in $next$ position then
                    break
                end if
            while $current$ is not equal to $begin\_range$
            if value in $current$ position is less than value in $next$ position then
                $++current$
            end if
            ++current2
            ++next2
            while $next2! = beyond$ do
                if value in $current2$ position is less than value in $next2$ position then
                    ++current2
                    ++next2
                end if
            end while
            **if** distance between $current$ and $next2$ is less than or equal $limit$ **then**
                reverse the order of the element in the range $current$, $next2$
                **if** distance between $begin\_range$ and current and begin is $equal$ to $beyond$ **then**
                    $begin = begin\_range$
                **end if**
                **if** $begin$ is not equal to $beyond$ **then**
                    sort the range $begin$, $current$ with the state of the art    ▷ sort the range using std::sort
                    add $current$ at the end of the $runs$, after its current last element
                    $begin = beyond$
                **end if**
                add $next2$ to $runs$
            **else**
                **if** $begin == beyond$ **then**
                    $begin = begin\_range$
                **end if**
            **end if**
        **else**
            **do**
                –current
                –next
                **if** value in $next$ position is less than value in $current$ position **then**
                    break
                **end if**
            **while** $current$ is not equal to $begin\_range$
            **if** value in $next$ position is less than value in $current$ position **then**
                ++current
            **end if**
            ++current2
            ++next2
            **while** $next2! = beyond$ **do**
                **if** value in $next2$ position is less than value in $current2$ position **then**
                    ++current2
                    ++next2
                **end if**
            **end while**
            **if** distance between $current$ and $next2$ is greater than or equal to $limit$ **then**
                **if** distance between $begin\_range$ and current and $begin$ is equal to $beyond$ **then**
                    $begin = begin\_range$
                **end if**
                **if** $begin$ is not equal to $beyond$ **then**
                    sort the range $begin$, $current$ with the state of the art algorithm
                    add $current$ into collection $runs$
                    $begin = beyond$
                **end if**
                add $next2$ into $runs$
            **else**
                **if** $begin == beyond$ **then**
                    $begin = begin\_range$
                **end if**
            **end if**
        **end if**
        if $next2 == beyond$ then
            break
            $current = std :: next(current2)$
            $next = std :: next(next2)$
        end if
        if $begin! = beyond$ then
            add $beyond$ into runs
            sort the range $begin$, $beyond$ with the state of the art sorting algorithm
        end if
        if size of $runs$ is less than 2 then
            $return$
        end if
        do
            var $again\_begin \leftarrow first$
            for from $first$ to until $runs$ end do
                merge runs pairwise
                remove the middle iterator
                advance $again\_begin$
            end for
        while $runs$ size is greater than 1
    end while
end procedure
```

```
function SORT(first, beyond, less)
    construct a container $V$ with as many elements as the range $[first, beyond]$
    HYBRIDSORT($first, beyond, less$)
    copy all the sorted elements in the range $[first, beyond]$ into $result$
    return $result$
end function
```

References

[Rocca and Cantone,] Rocca, M. L. and Cantone, D. NeatSort -A practical adaptive algorithm.

[Vignesh and Pradhan, 2016] Vignesh, R. and Pradhan, T. (2016). Merge sort enhanced in place sorting algorithm. In *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, pages 698–704.