Merge sort

In computer science, **merge sort** (also commonly spelled as **mergesort**) is an efficient, general-purpose, and comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output. Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.[2] A detailed description and analysis of bottom-up merge sort appeared in a report by Goldstine and von Neumann as early as 1948.[3]

Merge sort

An example of merge sort. First divide the list into the smallest unit (1 element), then compare each element with the adjacent list to sort and merge the two adjacent lists. Finally all the elements are sorted and merged.

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$
Best-case performance	$\Omega(n \log n)$ typical, $\Omega(n)$ natural variant
Average performance	$\Theta(n \log n)$
Worst-case space complexity	$O(n)$ total with $O(n)$ auxiliary, $O(1)$ auxiliary with linked lists[1]

Contents

- **Algorithm**
  - Top-down implementation
  - Bottom-up implementation
  - Top-down implementation using lists
  - Bottom-up implementation using lists
- **Natural merge sort**
- **Analysis**
- **Variants**
- **Use with tape drives**
- **Optimizing merge sort**
- **Parallel merge sort**
  - Merge sort with parallel recursion
  - Merge sort with parallel merging
  - Parallel multiway merge sort
    - Basic Idea
    - Multi-sequence selection
    - Pseudocode
    - Analysis
    - Practical adaption and application
  - Further Variants
- **Comparison with other sort algorithms**
- **Notes**
- **References**
- **External links**

Algorithm

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n* sublists, each containing one element (a list of one element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

Top-down implementation

Example C-like code using indices for top-down merge sort algorithm that recursively splits the list (called *runs* in this example) into sublists until sublist size is 1, then merges those sublists to produce a sorted list. The copy back step is avoided with alternating the direction of the merge with each level of recursion (except for an initial one-time copy). To help understand this, consider an array with 2 elements. The elements are copied to B[], then merged back to A[]. If there are 4 elements, when the bottom of the recursion level is reached, single element runs from A[] are merged to B[], and then at the next higher level of recursion, those 2 element runs are merged to A[]. This pattern continues with each level of recursion.

```
// Array A[] has the items to sort; array B[] is a work array.
void TopDownMergeSort(A[], B[], n)
{
    CopyArray(A, 0, n, B);           // one time copy of A[] to B[]
    TopDownSplitMerge(B, 0, n, A);   // sort data from B[] into A[]
}

// Split A[] into 2 runs, sort both runs into B[], merge both runs from B[] to A[]
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set).
void TopDownSplitMerge(B[], iBegin, iEnd, A[])
{
    if (iEnd - iBegin <= 1)                     // if run size == 1
        return;                                 //   consider it sorted
    // split the run longer than 1 item into halves
    iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
    // recursively sort both runs from array A[] into B[]
    TopDownSplitMerge(A, iBegin,  iMiddle, B);  // sort the left  run
    TopDownSplitMerge(A, iMiddle,    iEnd, B);  // sort the right run
    // merge the resulting runs from array B[] into A[]
    TopDownMerge(B, iBegin, iMiddle, iEnd, A);
}

//  Left source half is A[ iBegin:iMiddle-1].
// Right source half is A[iMiddle:iEnd-1   ].
// Result is            B[ iBegin:iEnd-1   ].
void TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
    i = iBegin, j = iMiddle;

    // While there are elements in the left or right runs...
    for (k = iBegin; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iMiddle && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            i = i + 1;
        } else {
            B[k] = A[j];
            j = j + 1;
        }
    }
}

void CopyArray(A[], iBegin, iEnd, B[])
{
    for (k = iBegin; k < iEnd; k++)
        B[k] = A[k];
}
```

Sorting the entire array is accomplished by `TopDownMergeSort(A, B, length(A))`.

Bottom-up implementation

Example C-like code using indices for bottom-up merge sort algorithm which treats the list as an array of *n* sublists (called *runs* in this example) of size 1, and iteratively merges sub-lists back and forth between two buffers:

```
// array A[] has the items to sort; array B[] is a work array
void BottomUpMergeSort(A[], B[], n)
{
    // Each 1-element run in A is already "sorted".
    // Make successively longer sorted runs of length 2, 4, 8, 16... until the whole array is
sorted.
    for (width = 1; width < n; width = 2 * width)
    {
        // Array A is full of runs of length width.
        for (i = 0; i < n; i = i + 2 * width)
        {
            // Merge two runs: A[i:i+width-1] and A[i+width:i+2*width-1] to B[]
            // or copy A[i:n-1] to B[] ( if (i+width >= n) )
            BottomUpMerge(A, i, min(i+width, n), min(i+2*width, n), B);
        }
        // Now work array B is full of runs of length 2*width.
        // Copy array B to array A for the next iteration.
        // A more efficient implementation would swap the roles of A and B.
        CopyArray(B, A, n);
        // Now array A is full of runs of length 2*width.
    }
}

//  Left run is A[iLeft :iRight-1].
// Right run is A[iRight:iEnd-1  ].
void BottomUpMerge(A[], iLeft, iRight, iEnd, B[])
{
    i = iLeft, j = iRight;
    // While there are elements in the left or right runs...
    for (k = iLeft; k < iEnd; k++) {
        // If left run head exists and is <= existing right run head.
        if (i < iRight && (j >= iEnd || A[i] <= A[j])) {
            B[k] = A[i];
            i = i + 1;
        } else {
            B[k] = A[j];
            j = j + 1;
        }
    }
}

void CopyArray(B[], A[], n)
{
    for (i = 0; i < n; i++)
        A[i] = B[i];
}
```

Top-down implementation using lists

Pseudocode for top-down merge sort algorithm which recursively divides the input list into smaller sublists until the sublists are trivially sorted, and then merges the sublists while returning up the call chain.

```
function merge_sort(list m) is
    // Base case. A list of zero or one elements is sorted, by definition.
    if length of m ≤ 1 then
        return m

    // Recursive case. First, divide the list into equal-sized sublists
    // consisting of the first half and second half of the list.
    // This assumes lists start at index 0.
    var left := empty list
```

```
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right

    // Recursively sort both sublists.
    left := merge_sort(left)
    right := merge_sort(right)

    // Then merge the now-sorted sublists.
    return merge(left, right)
```

In this example, the `merge` function merges the left and right sublists.

```
function merge(left, right) is
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    // Either left or right may have elements left; consume them.
    // (Only one of the following loops will actually be entered.)
    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

Bottom-up implementation using lists

Pseudocode for bottom-up merge sort algorithm which uses a small fixed size array of references to nodes, where array[i] is either a reference to a list of size 2^i or *nil*. *node* is a reference or pointer to a node. The merge() function would be similar to the one shown in the top-down merge lists example, it merges two already sorted lists, and handles empty lists. In this case, merge() would use *node* for its input parameters and return value.

```
function merge_sort(node head) is
    // return if empty list
    if head = nil then
        return nil
    var node array[32]; initially all nil
    var node result
    var node next
    var int  i
    result := head
    // merge nodes into array
    while result ≠ nil do
        next := result.next;
        result.next := nil
        for (i = 0; (i < 32) && (array[i] ≠ nil); i += 1) do
            result := merge(array[i], result)
            array[i] := nil
        // do not go past end of array
        if i = 32 then
            i -= 1
        array[i] := result
        result := next
    // merge array into single list
```

```
    result := nil
    for (i = 0; i < 32; i += 1) do
        result := merge(array[i], result)
    return result
```

Natural merge sort

A natural merge sort is similar to a bottom-up merge sort except that any naturally occurring runs (sorted sequences) in the input are exploited. Both monotonic and bitonic (alternating up/down) runs may be exploited, with lists (or equivalently tapes or files) being convenient data structures (used as FIFO queues or LIFO stacks).[4] In the bottom-up merge sort, the starting point assumes each run is one item long. In practice, random input data will have many short runs that just happen to be sorted. In the typical case, the natural merge sort may not need as many passes because there are fewer runs to merge. In the best case, the input is already sorted (i.e., is one run), so the natural merge sort need only make one pass through the data. In many practical cases, long natural runs are present, and for that reason natural merge sort is exploited as the key component of Timsort. Example:

```
Start       :  3  4  2  1  7  5  8  9  0  6
Select runs : (3  4)(2)(1  7)(5  8  9)(0  6)
Merge       : (2  3  4)(1  5  7  8  9)(0  6)
Merge       : (1  2  3  4  5  7  8  9)(0  6)
Merge       : (0  1  2  3  4  5  6  7  8  9)
```

Formally, the natural merge sort is said to be Runs-optimal, where $\mathbf{Runs}(L)$ is the number of runs in L, minus one.

Tournament replacement selection sorts are used to gather the initial runs for external sorting algorithms.

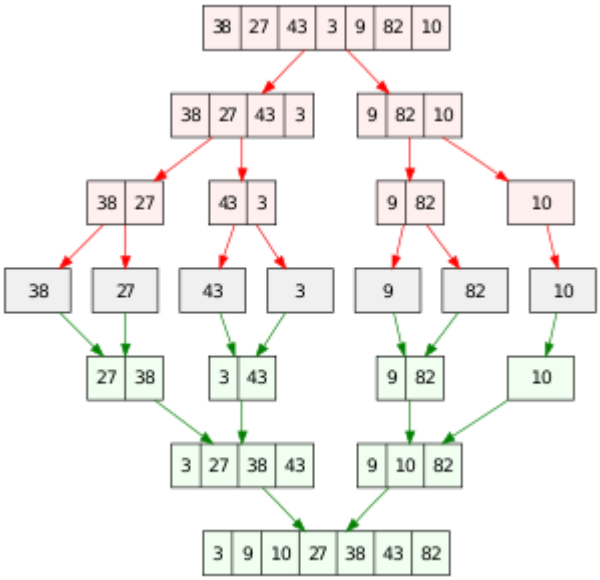
Analysis

In sorting n objects, merge sort has an average and worst-case performance of O(n log n). If the running time of merge sort for a list of length n is $T(n)$, then the recurrence relation $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists).[5] The closed form follows from the master theorem for divide-and-conquer recurrences.

The number of comparisons made by merge sort in the worst case is given by the sorting numbers. These numbers are equal to or slightly smaller than ($n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$), which is between ($n \lg n - n + 1$) and ($n \lg n + n + \mathrm{O}(\lg n)$).[6] Merge sort's best case takes about half as many iterations as its worst case.[7]

For large n and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches $\alpha \cdot n$ fewer than the worst case, where

$$\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k + 1} \approx 0.2645.$$



A recursive merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort (top-down).

In the *worst* case, merge sort uses approximately 39% fewer comparisons than quicksort does in its *average* case, and in terms of moves, merge sort's worst case complexity is O(n log n) - the same complexity as quicksort's best case.[7]

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort.

Merge sort's most common implementation does not sort in place;[8] therefore, the memory size of the input must be allocated for the sorted output to be stored in (see below for variations that need only n/2 extra spaces).

Variants

Variants of merge sort are primarily concerned with reducing the space complexity and the cost of copying.

A simple alternative for reducing the space overhead to n/2 is to maintain *left* and *right* as a combined structure, copy only the *left* part of m into temporary space, and to direct the *merge* routine to place the merged output into m. With this version it is better to allocate the temporary space outside the *merge* routine, so that only one allocation is needed. The excessive copying mentioned previously is also mitigated, since the last pair of lines before the *return result* statement (function *merge* in the pseudo code above) become superfluous.

One drawback of merge sort, when implemented on arrays, is its $O(n)$ working memory requirement. Several in-place variants have been suggested:

- Katajainen *et al.* present an algorithm that requires a constant amount of working memory: enough storage space to hold one element of the input array, and additional space to hold $O(1)$ pointers into the input array. They achieve an $O(n \log n)$ time bound with small constants, but their algorithm is not stable.[9]
- Several attempts have been made at producing an *in-place merge* algorithm that can be combined with a standard (top-down or bottom-up) merge sort to produce an in-place merge sort. In this case, the notion of "in-place" can be relaxed to mean "taking logarithmic stack space", because standard merge sort requires that amount of space for its own stack usage. It was shown by Geffert *et al.* that in-place, stable merging is possible in $O(n \log n)$ time using a constant amount of scratch space, but their algorithm is complicated and has high constant factors: merging arrays of length n and m can take $5n + 12m + o(m)$ moves.[10] This high constant factor and complicated in-place algorithm was made simpler and easier to understand. Bing-Chao Huang and Michael A. Langston[11] presented a straightforward linear time algorithm *practical in-place merge* to merge a sorted list using fixed amount of additional space. They both have used the work of Kronrod and others. It merges in linear time and constant extra space. The algorithm takes little more average time than standard merge sort algorithms, free to exploit O(n) temporary extra memory cells, by less than a factor of two. Though the algorithm is much faster in a practical way but it is unstable also for some lists. But using similar concepts, they have been able to solve this problem. Other in-place algorithms include SymMerge, which takes $O((n + m) \log (n + m))$ time in total and is stable.[12] Plugging such an algorithm into merge sort increases its complexity to the non-linearithmic, but still quasilinear, $O(n (\log n)^2)$.
- A modern stable linear and in-place merging is block merge sort.

An alternative to reduce the copying into multiple lists is to associate a new field of information with each key (the elements in m are called keys). This field will be used to link the keys and any associated information together in a sorted list (a key and its related information is called a record). Then the merging of the sorted lists

proceeds by changing the link values; no records need to be moved at all. A field which contains only a link will generally be smaller than an entire record so less space will also be used. This is a standard sorting technique, not restricted to merge sort.

Use with tape drives

An external merge sort is practical to run using disk or tape drives when the data to be sorted is too large to fit into memory. External sorting explains how merge sort is implemented with disk drives. A typical tape drive sort uses four tape drives. All I/O is sequential (except for rewinds at the end of each pass). A minimal implementation can get by with just two record buffers and a few program variables.

Merge sort type algorithms allowed large data sets to be sorted on early computers that had small random access memories by modern standards. Records were stored on magnetic tape and processed on banks of magnetic tape drives, such as these IBM 729s.

Naming the four tape drives as A, B, C, D, with the original data on A, and using only two record buffers, the algorithm is similar to the bottom-up implementation, using pairs of tape drives instead of arrays in memory. The basic algorithm can be described as follows:

1. Merge pairs of records from A; writing two-record sublists alternately to C and D.
2. Merge two-record sublists from C and D into four-record sublists; writing these alternately to A and B.
3. Merge four-record sublists from A and B into eight-record sublists; writing these alternately to C and D
4. Repeat until you have one list containing all the data, sorted—in $\log_2(n)$ passes.

Instead of starting with very short runs, usually a hybrid algorithm is used, where the initial pass will read many records into memory, do an internal sort to create a long run, and then distribute those long runs onto the output set. The step avoids many early passes. For example, an internal sort of 1024 records will save nine passes. The internal sort is often large because it has such a benefit. In fact, there are techniques that can make the initial runs longer than the available internal memory. One of them, the Knuth's 'snowplow' (based on a binary min-heap), generates runs twice as long (on average) as a size of memory used.[13]

With some overhead, the above algorithm can be modified to use three tapes. $O(n \log n)$ running time can also be achieved using two queues, or a stack and a queue, or three stacks. In the other direction, using $k >$ two tapes (and $O(k)$ items in memory), we can reduce the number of tape operations in $O(\log k)$ times by using a k/2-way merge.

A more sophisticated merge sort that optimizes tape (and disk) drive usage is the polyphase merge sort.

Optimizing merge sort

On modern computers, locality of reference can be of paramount importance in software optimization, because multilevel memory hierarchies are used. Cache-aware versions of the merge sort algorithm, whose operations have been specifically chosen to minimize the movement of pages in and out of a machine's memory cache, have been proposed. For example, the **tiled merge sort** algorithm stops partitioning subarrays when subarrays of size S are reached, where S is the number of data items fitting into a CPU's cache. Each of these subarrays is sorted with an in-place sorting algorithm such as insertion sort, to discourage memory swaps, and normal merge sort is then completed in the standard recursive fashion. This algorithm has demonstrated better performance on machines that benefit from cache optimization. (LaMarca & Ladner 1997)

Kronrod (1969) suggested an alternative version of merge sort that uses constant additional space. This algorithm was later refined. (Katajainen, Pasanen & Teuhola 1996)

Also, many applications of external sorting use a form of merge sorting where the input get split up to a higher number of sublists, ideally to a number for which merging them still makes the currently processed set of pages fit into main memory.

Tiled merge sort applied to an array of random integers. The horizontal axis is the array index and the vertical axis is the integer.

Parallel merge sort

Merge sort parallelizes well due to the use of the divide-and-conquer method. Several different parallel variants of the algorithm have been developed over the years. Some parallel merge sort algorithms are strongly related to the sequential top-down merge algorithm while others have a different general structure and use the K-way merge method.

Merge sort with parallel recursion

The sequential merge sort procedure can be described in two phases, the divide phase and the merge phase. The first consists of many recursive calls that repeatedly perform the same division process until the subsequences are trivially sorted (containing one or no element). An intuitive approach is the parallelization of those recursive calls.[14] Following pseudocode describes the merge sort with parallel recursion using the fork and join keywords:

```
// Sort elements lo through hi (exclusive) of array A.
algorithm mergesort(A, lo, hi) is
    if lo+1 < hi then  // Two or more elements.
        mid := ⌊(lo + hi) / 2⌋
        fork mergesort(A, lo, mid)
        mergesort(A, mid, hi)
        join
        merge(A, lo, mid, hi)
```

This algorithm is the trivial modification of the sequential version and does not parallelize well. Therefore, its speedup is not very impressive. It has a span of $\Theta(n)$, which is only an improvement of $\Theta(\log n)$ compared to the sequential version (see Introduction to Algorithms). This is mainly due to the sequential merge method, as it is the bottleneck of the parallel executions.

Merge sort with parallel merging

Better parallelism can be achieved by using a parallel merge algorithm. Cormen et al. present a binary variant that merges two sorted sub-sequences into one sorted output sequence.[14]

In one of the sequences (the longer one if unequal length), the element of the middle index is selected. Its position in the other sequence is determined in such a way that this sequence would remain sorted if this element were inserted at this position. Thus, one knows how many other elements from both sequences are smaller and the position of the selected element in the output sequence can be calculated. For the partial sequences of the smaller and larger elements created in this way, the merge algorithm is again executed in parallel until the base case of the recursion is reached.

The following pseudocode shows the modified parallel merge sort method using the parallel merge algorithm (adopted from Cormen et al.).

```
/**
 * A: Input array
 * B: Output array
 * lo: lower bound
 * hi: upper bound
 * off: offset
 */
algorithm parallelMergesort(A, lo, hi, B, off) is
    len := hi - lo + 1
    if len == 1 then
        B[off] := A[lo]
    else let T[1..len] be a new array
        mid := ⌊(lo + hi) / 2⌋
        mid' := mid - lo + 1
        fork parallelMergesort(A, lo, mid, T, 1)
        parallelMergesort(A, mid + 1, hi, T, mid' + 1)
        join
        parallelMerge(T, 1, mid', mid' + 1, len, B, off)
```

In order to analyze a recurrence relation for the worst case span, the recursive calls of parallelMergesort have to be incorporated only once due to their parallel execution, obtaining

$$T_\infty^{\text{sort}}(n) = T_\infty^{\text{sort}}\left(\frac{n}{2}\right) + T_\infty^{\text{merge}}(n) = T_\infty^{\text{sort}}\left(\frac{n}{2}\right) + \Theta\left(\log(n)^2\right).$$

For detailed information about the complexity of the parallel merge procedure, see Merge algorithm.

The solution of this recurrence is given by

$$T_\infty^{\text{sort}} = \Theta\left(\log(n)^3\right).$$

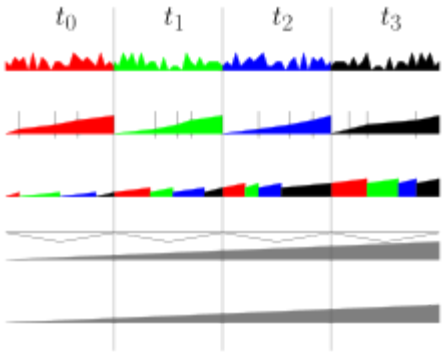
This parallel merge algorithm reaches a parallelism of $\Theta\left(\frac{n}{(\log n)^2}\right)$, which is much higher than the parallelism of the previous algorithm. Such a sort can perform well in practice when combined with a fast stable sequential sort, such as insertion sort, and a fast sequential merge as a base case for merging small arrays.[15]

Parallel multiway merge sort

It seems arbitrary to restrict the merge sort algorithms to a binary merge method, since there are usually p > 2 processors available. A better approach may be to use a K-way merge method, a generalization of binary merge, in which k sorted sequences are merged. This merge variant is well suited to describe a sorting algorithm on a PRAM.[16][17]

Basic Idea

Given an unsorted sequence of n elements, the goal is to sort the sequence with p available processors. These elements are distributed equally among all processors and sorted locally using a sequential Sorting algorithm. Hence, the sequence consists of sorted sequences $S_1, \ldots, S_p$ of length $\lceil \frac{n}{p} \rceil$. For simplification let n be a multiple of p, so that $|S_i| = \frac{n}{p}$ for $i = 1, \ldots, p$.



The parallel multiway mergesort process on four processors t_0 to t_3.

These sequences will be used to perform a multisequence selection/splitter selection. For $j = 1, \ldots, p$, the algorithm determines splitter elements v_j with global rank $k = j\frac{n}{p}$. Then the corresponding positions of $v_1, \ldots, v_p$ in each sequence S_i are determined with binary search and thus the S_i are further partitioned into p subsequences $S_{i,1}, \ldots, S_{i,p}$ with $S_{i,j} := \{x \in S_i | rank(v_{j-1}) < rank(x) \leq rank(v_j)\}$.

Furthermore, the elements of $S_{1,i}, \ldots, S_{p,i}$ are assigned to processor i, means all elements between rank $(i-1)\frac{n}{p}$ and rank $i\frac{n}{p}$, which are distributed over all S_i. Thus, each processor receives a sequence of sorted sequences. The fact that the rank k of the splitter elements v_i was chosen globally, provides two important properties: On the one hand, k was chosen so that each processor can still operate on n/p elements after assignment. The algorithm is perfectly load-balanced. On the other hand, all elements on processor i are less than or equal to all elements on processor $i+1$. Hence, each processor performs the *p*-way merge locally and thus obtains a sorted sequence from its sub-sequences. Because of the second property, no further *p*-way-merge has to be performed, the results only have to be put together in the order of the processor number.

Multi-sequence selection

In its simplest form, given p sorted sequences $S_1, \ldots, S_p$ distributed evenly on p processors and a rank k, the task is to find an element x with a global rank k in the union of the sequences. Hence, this can be used to divide each S_i in two parts at a splitter index l_i, where the lower part contains only elements which are smaller than x, while the elements bigger than x are located in the upper part.

The presented sequential algorithm returns the indices of the splits in each sequence, e.g. the indices l_i in sequences S_i such that $S_i[l_i]$ has a global rank less than k and $\mathbf{rank}\,(S_i[l_i + 1]) \geq k$.[18]

```
algorithm msSelect(S : Array of sorted Sequences [S_1,..,S_p], k : int) is
    for i = 1 to p do
        (l_i, r_i) = (0, |S_i|-1)

    while there exists i: l_i < r_i do
        // pick Pivot Element in S_j[l_j], .., S_j[r_j], chose random j uniformly
        v := pickPivot(S, l, r)
        for i = 1 to p do
            m_i = binarySearch(v, S_i[l_i, r_i]) // sequentially
        if m_1 + ... + m_p >= k then // m_1+ ... + m_p is the global rank of v
            r := m  // vector assignment
        else
            l := m

    return l
```

For the complexity analysis the PRAM model is chosen. If the data is evenly distributed over all p, the p-fold execution of the *binarySearch* method has a running time of $\mathcal{O}(p \log(n/p))$. The expected recursion depth is $\mathcal{O}(\log(\sum_i |S_i|)) = \mathcal{O}(\log(n))$ as in the ordinary Quickselect. Thus the overall expected running time is

$\mathcal{O}(p \log(n/p) \log(n))$.

Applied on the parallel multiway merge sort, this algorithm has to be invoked in parallel such that all splitter elements of rank $i\frac{n}{p}$ for $i = 1, .., p$ are found simultaneously. These splitter elements can then be used to partition each sequence in p parts, with the same total running time of $\mathcal{O}(p \log(n/p) \log(n))$.

Pseudocode

Below, the complete pseudocode of the parallel multiway merge sort algorithm is given. We assume that there is a barrier synchronization before and after the multisequence selection such that every processor can determine the splitting elements and the sequence partition properly.

```
/**
 * d: Unsorted Array of Elements
 * n: Number of Elements
 * p: Number of Processors
 * return Sorted Array
 */
algorithm parallelMultiwayMergesort(d : Array, n : int, p : int) is
    o := new Array[0, n]                          // the output array
    for i = 1 to p do in parallel                 // each processor in parallel
        S_i := d[(i-1) * n/p, i * n/p]            // Sequence of length n/p
        sort(S_i)                                 // sort locally
        synch
        v_i := msSelect([S_1,...,S_p], i * n/p)          // element with global rank i * n/p
        synch
        (S_i,1, ..., S_i,p) := sequence_partitioning(si, v_1, ..., v_p) // split s_i into
subsequences

        o[(i-1) * n/p, i * n/p] := kWayMerge(s_1,i, ..., s_p,i)  // merge and assign to output
array

    return o
```

Analysis

Firstly, each processor sorts the assigned n/p elements locally using a sorting algorithm with complexity $\mathcal{O}(n/p \log(n/p))$. After that, the splitter elements have to be calculated in time $\mathcal{O}(p \log(n/p) \log(n))$. Finally, each group of p splits have to be merged in parallel by each processor with a running time of $\mathcal{O}(\log(p) n/p)$ using a sequential p-way merge algorithm. Thus, the overall running time is given by

$$\mathcal{O}\left(\frac{n}{p}\log\left(\frac{n}{p}\right) + p\log\left(\frac{n}{p}\right)\log(n) + \frac{n}{p}\log(p)\right).$$

Practical adaption and application

The multiway merge sort algorithm is very scalable through its high parallelization capability, which allows the use of many processors. This makes the algorithm a viable candidate for sorting large amounts of data, such as those processed in computer clusters. Also, since in such systems memory is usually not a limiting resource, the disadvantage of space complexity of merge sort is negligible. However, other factors become important in such systems, which are not taken into account when modelling on a PRAM. Here, the following aspects need to be considered: Memory hierarchy, when the data does not fit into the processors cache, or the communication overhead of exchanging data between processors, which could become a bottleneck when the data can no longer be accessed via the shared memory.

Sanders et al. have presented in their paper a bulk synchronous parallel algorithm for multilevel multiway mergesort, which divides p processors into r groups of size p'. All processors sort locally first. Unlike single level multiway mergesort, these sequences are then partitioned into r parts and assigned to the appropriate processor groups. These steps are repeated recursively in those groups. This reduces communication and especially avoids problems with many small messages. The hierarchical structure of the underlying real network can be used to define the processor groups (e.g. racks, clusters,...).[17]

Further Variants

Merge sort was one of the first sorting algorithms where optimal speed up was achieved, with Richard Cole using a clever subsampling algorithm to ensure $O(1)$ merge.[19] Other sophisticated parallel sorting algorithms can achieve the same or better time bounds with a lower constant. For example, in 1991 David Powers described a parallelized quicksort (and a related radix sort) that can operate in $O(\log n)$ time on a CRCW parallel random-access machine (PRAM) with n processors by performing partitioning implicitly.[20] Powers further shows that a pipelined version of Batcher's Bitonic Mergesort at $O((\log n)^2)$ time on a butterfly sorting network is in practice actually faster than his $O(\log n)$ sorts on a PRAM, and he provides detailed discussion of the hidden overheads in comparison, radix and parallel sorting.[21]

Comparison with other sort algorithms

Although heapsort has the same time bounds as merge sort, it requires only $\Theta(1)$ auxiliary space instead of merge sort's $\Theta(n)$. On typical modern architectures, efficient quicksort implementations generally outperform merge sort for sorting RAM-based arrays. On the other hand, merge sort is a stable sort and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\Theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

As of Perl 5.8, merge sort is its default sorting algorithm (it was quicksort in previous versions of Perl).[22] In Java, the Arrays.sort() (https://docs.oracle.com/javase/9/docs/api/java/util/Arrays.html#sort-java.lang.Object:A-) methods use merge sort or a tuned quicksort depending on the datatypes and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.[23] The Linux kernel uses merge sort for its linked lists.[24] Python uses Timsort, another tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7 (for arrays of non-primitive types),[25] on the Android platform,[26] and in GNU Octave.[27]

Notes

1. Skiena (2008, p. 122)
2. Knuth (1998, p. 158)
3. Katajainen, Jyrki; Träff, Jesper Larsson (March 1997). "A meticulous analysis of mergesort programs" (http://hjemmesider.diku.dk/~jyrki/Paper/CIAC97.pdf) (PDF). *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*. Italian Conference on Algorithms and Complexity. Rome. pp. 217–228. CiteSeerX 10.1.1.86.3154 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.3154). doi:10.1007/3-540-62592-5_74 (https://doi.org/10.1007%2F3-540-62592-5_74).
4. Powers, David M. W.; McMahon, Graham B. (1983). "A compendium of interesting prolog programs". DCS Technical Report 8313 (Report). Department of Computer Science, University of New South Wales.
5. Cormen et al. (2009, p. 36)

6. The worst case number given here does not agree with that given in Knuth's *Art of Computer Programming, Vol 3*. The discrepancy is due to Knuth analyzing a variant implementation of merge sort that is slightly sub-optimal
7. Jayalakshmi, N. (2007). *Data structure using C++* (https://www.worldcat.org/oclc/849900742). ISBN 978-81-318-0020-1. OCLC 849900742 (https://www.worldcat.org/oclc/849900742).
8. Cormen et al. (2009, p. 151)
9. Katajainen, Pasanen & Teuhola (1996)
10. Geffert, Viliam; Katajainen, Jyrki; Pasanen, Tomi (2000). "Asymptotically efficient in-place merging" (https://doi.org/10.1016%2FS0304-3975%2898%2900162-5). *Theoretical Computer Science*. **237** (1–2): 159–181. doi:10.1016/S0304-3975(98)00162-5 (https://doi.org/10.1016%2FS0304-3975%2898%2900162-5).
11. Huang, Bing-Chao; Langston, Michael A. (March 1988). "Practical In-Place Merging". *Communications of the ACM*. **31** (3): 348–352. doi:10.1145/42392.42403 (https://doi.org/10.1145%2F42392.42403). S2CID 4841909 (https://api.semanticscholar.org/CorpusID:4841909).
12. Kim, Pok-Son; Kutzner, Arne (2004). *Stable Minimum Storage Merging by Symmetric Comparisons*. European Symp. Algorithms. Lecture Notes in Computer Science. **3221**. pp. 714–723. CiteSeerX 10.1.1.102.4612 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.4612). doi:10.1007/978-3-540-30140-0_63 (https://doi.org/10.1007%2F978-3-540-30140-0_63). ISBN 978-3-540-23025-0.
13. Ferragina, Paolo (2009–2019), "5. Sorting Atomic Items" (http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformaticanetworking/ae/ae2018/ferragina_notes_algoeng_2019_vers_5_.pdf) (PDF), *The magic of Algorithms!*, p. 5-4, archived (https://web.archive.org/web/20210512230253/http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformaticanetworking/ae/ae2018/ferragina_notes_algoeng_2019_vers_5_.pdf) (PDF) from the original on 2021-05-12
14. Cormen et al. (2009, pp. 797–805)
15. Victor J. Duvanenko "Parallel Merge Sort" Dr. Dobb's Journal & blog [1] (https://duvanenko.tech.blog/2018/01/13/parallel-merge-sort/) and GitHub repo C++ implementation [2] (https://github.com/DragonSpit/ParallelAlgorithms)
16. Peter Sanders; Johannes Singler (2008). "Lecture *Parallel algorithms*" (http://algo2.iti.kit.edu/sanders/courses/paralg08/singler.pdf) (PDF). Retrieved 2020-05-02.
17. Axtmann, Michael; Bingmann, Timo; Sanders, Peter; Schulz, Christian (2015). "Practical Massively Parallel Sorting" (https://publikationen.bibliothek.kit.edu/1000050651/37296033). *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*: 13–23. doi:10.1145/2755573.2755595 (https://doi.org/10.1145%2F2755573.2755595). ISBN 9781450335881. S2CID 18249978 (https://api.semanticscholar.org/CorpusID:18249978).
18. Peter Sanders (2019). "Lecture *Parallel algorithms*" (http://algo2.iti.kit.edu/sanders/courses/paralg19/vorlesung.pdf) (PDF). Retrieved 2020-05-02.
19. Cole, Richard (August 1988). "Parallel merge sort". *SIAM J. Comput*. **17** (4): 770–785. CiteSeerX 10.1.1.464.7118 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.464.7118). doi:10.1137/0217049 (https://doi.org/10.1137%2F0217049). S2CID 2416667 (https://api.semanticscholar.org/CorpusID:2416667).
20. Powers, David M. W. (1991). "Parallelized Quicksort and Radixsort with Optimal Speedup" (https://web.archive.org/web/20070525234405/http://citeseer.ist.psu.edu/327487.html). *Proceedings of International Conference on Parallel Computing Technologies, Novosibirsk*. Archived from the original (http://citeseer.ist.psu.edu/327487.html) on 2007-05-25.
21. Powers, David M. W. (January 1995). *Parallel Unification: Practical Complexity* (http://david.wardpowers.info/Research/AI/papers/199501-ACAW-PUPC.pdf) (PDF). Australasian Computer Architecture Workshop Flinders University.
22. "Sort – Perl 5 version 8.8 documentation" (https://perldoc.perl.org/5.8.8/sort.html). Retrieved 2020-08-23.

23. coleenp (22 Feb 2019). "src/java.base/share/classes/java/util/Arrays.java @ 53904:9c3fe09f69bc" (https://hg.openjdk.java.net/jdk/jdk/file/9c3fe09f69bc/src/java.base/share/classes/java/util/Arrays.java#l1331). *OpenJDK*.
24. linux kernel /lib/list_sort.c (https://github.com/torvalds/linux/blob/master/lib/list_sort.c)
25. jjb (29 Jul 2009). "Commit 6804124: Replace "modified mergesort" in java.util.Arrays.sort with timsort" (http://hg.openjdk.java.net/jdk7/jdk7/jdk/rev/bfd7abda8f79). *Java Development Kit 7 Hg repo*. Archived (https://web.archive.org/web/20180126184957/http://hg.openjdk.java.net/jdk7/jdk7/jdk/rev/bfd7abda8f79) from the original on 2018-01-26. Retrieved 24 Feb 2011.
26. "Class: java.util.TimSort<T>" (https://web.archive.org/web/20150120063131/https://android.googlesource.com/platform/libcore/%2B/jb-mr2-release/luni/src/main/java/java/util/TimSort.java). *Android JDK Documentation*. Archived from the original (https://android.googlesource.com/platform/libcore/+/jb-mr2-release/luni/src/main/java/java/util/TimSort.java) on January 20, 2015. Retrieved 19 Jan 2015.
27. "liboctave/util/oct-sort.cc" (http://hg.savannah.gnu.org/hgweb/octave/file/0486a29d780f/liboctave/util/oct-sort.cc). *Mercurial repository of Octave source code*. Lines 23-25 of the initial comment block. Retrieved 18 Feb 2013. "Code stolen in large part from Python's, listobject.c, which itself had no license header. However, thanks to Tim Peters for the parts of the code I ripped-off."

References

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.
- Katajainen, Jyrki; Pasanen, Tomi; Teuhola, Jukka (1996). "Practical in-place mergesort" (https://web.archive.org/web/20110807033704/http://www.diku.dk/hjemmesider/ansatte/jyrki/Paper/mergesort_NJC.ps). *Nordic Journal of Computing*. **3** (1): 27–40. CiteSeerX 10.1.1.22.8523 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.8523). ISSN 1236-6064 (https://www.worldcat.org/issn/1236-6064). Archived from the original (http://www.diku.dk/hjemmesider/ansatte/jyrki/Paper/mergesort_NJC.ps) on 2011-08-07. Retrieved 2009-04-04.. Also Practical In-Place Mergesort (http://citeseer.ist.psu.edu/katajainen96practical.html). Also [3] (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.8523)
- Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". *Sorting and Searching*. The Art of Computer Programming. **3** (2nd ed.). Addison-Wesley. pp. 158–168. ISBN 0-201-89685-0.
- Kronrod, M. A. (1969). "Optimal ordering algorithm without operational field". *Soviet Mathematics - Doklady*. **10**: 744.
- LaMarca, A.; Ladner, R. E. (1997). "The influence of caches on the performance of sorting". *Proc. 8th Ann. ACM-SIAM Symp. On Discrete Algorithms (SODA97)*: 370–379. CiteSeerX 10.1.1.31.1153 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.1153).
- Skiena, Steven S. (2008). "4.5: Mergesort: Sorting by Divide-and-Conquer". *The Algorithm Design Manual* (2nd ed.). Springer. pp. 120–125. ISBN 978-1-84800-069-8.
- Sun Microsystems. "Arrays API (Java SE 6)" (http://java.sun.com/javase/6/docs/api/java/util/Arrays.html). Retrieved 2007-11-19.
- Oracle Corp. "Arrays (Java SE 10 & JDK 10)" (https://docs.oracle.com/javase/10/docs/api/java/util/Arrays.html). Retrieved 2018-07-23.

External links

- Animated Sorting Algorithms: Merge Sort (https://web.archive.org/web/20150306071601/http://www.sorting-algorithms.com/merge-sort) at the Wayback Machine (archived 6 March 2015) – graphical demonstration
- Open Data Structures - Section 11.1.1 - Merge Sort (http://opendatastructures.org/versions/edition-0.1e/ods-java/11_1_Comparison_Based_Sorti.html#SECTION00141100000000000000),

Pat Morin