

A Proof of Linear Work for Parallel Divide-and-Conquer Merge

Sam Westrick
New York University
s.westrick@nyu.edu

1 Introduction

A simple parallel algorithm for merging two sorted sequences in parallel uses a binary search to split one sequence at the median element of the other, leading to two recursive calls that can proceed in parallel. The final merged (output) sequence is just the concatenation of the recursive calls. This algorithm is quite efficient in practice, especially if the concatenation step is fused with the recursive unfolding, resulting in an algorithm that writes each output element exactly once, spread out across the leaves of the recursion. Pseudocode for this $\text{MERGE}(A, B)$ algorithm is shown in Figure 1.

Although not immediately obvious, $\text{MERGE}(A, B)$ performs only $O(n + m)$ work on inputs of size $n = |A|$ and $m = |B|$. At first glance, we might assume that it needs to do as much as $O(n \log m)$ work in the worst case; one possible analysis along these lines would be to observe that every element on the left participates as a “splitter” at most once, and the cost of splitting is $O(\log m)$. However, this analysis does not account for the fact that the right-hand-side sequences are getting smaller as we get deeper into the recursion.

In this short writeup, we give a fairly simple proof that the overall work cost is $O(n + m)$. The proof accounts for the data-dependent cost of splitting by summing level-by-level in the recursion tree. One insight is that the worst case is when B is split evenly across a level.

We note that we have formally mechanized the full proof in the Lean 4 theorem proving language, with precise (non-asymptotic) accounting of constant factors.

The full proof is given in Sections 2 and 3, with proofs of technical lemmas in Sections 4 and 5. The last two sections can be skipped if the reader prefers.

Acknowledgments. This proof is derived from a Rocq formalization, developed in collaboration with Alexandre Moine and Joe Tassarotti as part of our work on verification of time complexity for parallel programs (Moine et al., 2026).

Notation and Conventions. Throughout this document, we note that we are using the following conventions: $\lg$ is the rounded-up base-two logarithm with $\lg 0 = 0$, division is floor division, and subtraction on naturals is truncated.

```
MERGE( $A, B$ ):  
  if  $|A| + |B|$  is small  
    merge  $A$  and  $B$  sequentially  
  else if  $|A| = 0$   
    copy  $B$  to the output  
  else  
     $i \leftarrow \lfloor |A|/2 \rfloor$   
     $j \leftarrow$  position of  $A[i]$  in  $B$ ,  
      by binary search  
    write  $A[i]$  to output slot  $i + j$   
    in parallel:  
      MERGE( $A[0..i), B[0..j)$ )  
      MERGE( $A[i+1..), B[j..)$ )
```

Figure 1: Parallel merge by divide and conquer.

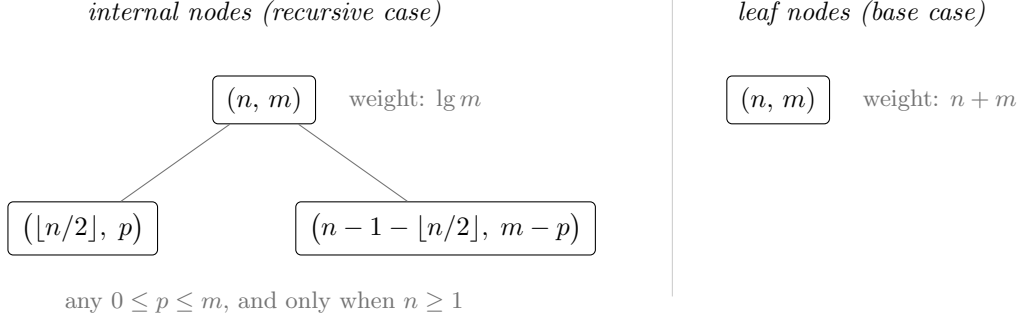


Figure 2: A node of a cost tree may split into two children (left), or may stop (right). Splitting halves the first size and consumes one unit of it for the pivot; the second size is cut at a point p that nothing constrains.

2 Cost Trees

A **cost tree** is a binary tree mirroring the recursive calls of MERGE across one run. We give a **label** (n, m) to each node, recording the input sizes at that recursive call, and assign each node a **weight** corresponding to the work cost of the algorithm at that node. For internal nodes, this weight is $\lg m$, to account for the binary search. For leaf nodes, the weight is $n + m$, to account for a sequential merge of the two inputs and writing the result to the output.

Cost trees and their weights (work costs) are depicted in Figure 2. Each internal node has two children, with labels determined by a split point, p . Every internal node can have a different split point, dependent upon the input data.

Since each cost tree corresponds to a run of MERGE, our goal (Theorem 2.1) is to prove that every cost tree with root labels (n, m) has a total weight of $O(n + m)$. Along these lines, we define $\|T\|$ as the sum of the weights of all nodes of T . We will write (n_v, m_v) for the labels of a node v .

Theorem 2.1. *Every cost tree T with root label (n, m) has $\|T\| \in O(n + m)$.*

3 Proof

For a cost tree T with root labels (n, m) , we can split it into two parts: the internal nodes, and the leaf nodes. The total weight of all leaf nodes is exactly $n + m - i$ where i is the number of internal nodes, because each internal node consumes one unit of the input size n and the input size m is partitioned onto the leaves. With a linear bound on the leaves, it suffices to show that the internal nodes also have linear weight.

The internal nodes of a cost tree can be grouped by their depth, d , and summarized by two quantities:

- k_d , the number of internal nodes at depth d , and
- M_d , the sum of the m_v labels across internal nodes v at depth d

The main idea in the proof is to bound the logarithmic sum of each level of the tree ($\sum_v \lg m_v$), showing that this quantity is maximized when the individual m_v labels are spread evenly across the layer (i.e., where each $m_v \approx M_d/k_d$).

$$\sum_{v \text{ at depth } d} \lg m_v \leq k_d \lg(M_d/k_d) + O(k_d) \quad (1)$$

We note that the additional $O(k_d)$ term is an artifact of discretization, accounting for small ± 1 differences in individual terms. This “even spreading” bound is essentially a discretized form of Jensen’s inequality, specialized for the rounded logarithms we are using; we formalize it in Lemma 4.1 in Section 4.

With the “even spreading” bound, the idea is to consider the overall sum, level-by-level, indexed by the depth d . The sum of the $O(k_d)$ term contributes $O(n)$ to the overall sum, because k_d grows exponentially (up to 2^d) and the tree has height $\lg n$. The $\sum_d k_d \lg(M_d/k_d)$ term is trickier to bound directly, but we can adjust the original application of Inequality 1 by “padding” each level to have exactly 2^d elements, with the additional elements set to $m_v = 0$. A direct application of the “even spreading” lemma to the padded layer yields the following bound:

$$\sum_{\substack{v \text{ at} \\ \text{depth } d}} \lg m_v = \sum_{\substack{v \text{ at} \\ \text{depth } d}} \lg m_v + \underbrace{(2^d - k_d) \lg 0}_{\text{padding for } 2^d \text{ elems total}} \leq 2^d \lg(M_d/2^d) + O(2^d)$$

The main proof then follows from the observation that $M_d \leq m$ (each layer essentially partitions the input size m), together with the following a technical lemma (Lemma 5.2; see Section 5), which bounds the summation across the levels: $\sum_{d < \lg n} 2^d \lg(m/2^d) \in O(n + m)$. This bounds the internal weights of the cost tree by $O(n + m)$. Together with the earlier bound on the leaf weights, this proves Theorem 2.1.

4 Technical Lemma: Even Spreading

The “even spreading” bound, as described above, is an instance of a general fact about lists of naturals, which we formalize here. The main point is the concavity of $\lg$: the logarithmic sum of a collection of values is most expensive when the total is spread evenly across the collection. Note that in the proof of Theorem 2.1 as described in Section 3, we instantiate this lemma with x_i taken to be the padded list of labels m_v at depth d .

Lemma 4.1 (Even spreading is worst). *For any $x_1, \dots, x_k \in \mathbb{N}$ with $\sum_i x_i \leq M$, we have $\sum_i \lg x_i \leq k \lg(M/k) + O(k)$.*

Proof sketch. The elements x_i equal to zero don’t contribute to the sum, since $\lg 0 = \lg 1 = 0$, so we replace each of them by one. This leaves the logarithmic sum $\sum_i \lg x_i$ and the length k unchanged, and raises the total $\sum_i x_i$ to at most $M + k$. Now that every element is nonzero, we have $\sum_i \lg x_i \leq \lg \prod_i x_i + O(k)$, where the $O(k)$ term accounts for per-element rounding. Now consider a maximization process, modifying the values of x_i until $\prod_i x'_i$ is maximized (where x'_i are the modified values). Specifically, for any two elements that differ by two or more, move one unit from the larger element to the smaller element. Moving one unit increases the overall product $\prod_i x'_i$ while preserving both the length and the total sum. This process continues until every pair of elements differ by at most 1. At this point, every element is within one of the average, so $\prod_i x'_i \leq (M/k + O(1))^k$. Taking $\lg$ gives $k \lg(M/k) + O(k)$. $\square$

5 Technical Lemma: Level Summation

Here we prove the last technical lemma, that $\sum_{d < \lg n} 2^d \lg(m/2^d) \in O(n + m)$. This is a fairly common summation found in algorithm analysis. We reprove it here only for clarity.

The tricky thing about the sum is that two quantities are changing in different directions as we go deeper in the tree: the quantity 2^d is rapidly approaching $2^{\lg n} = n$, whereas $\lg(m/2^d) \approx \lg(m) - d$ is decreasing linearly towards 0. Which one arrives first depends on the relative values of n and m .

It's helpful to start by considering the case where $n = m$. In this case we can re-index the summation as $\sum_{i=1}^k i 2^{k-i}$ where $k = \lg n$. This summation is relatively easy to bound by $2^{k+1} = O(n)$, using simple induction over k . If we generalize this for any k , we get Lemma 5.1, which will be useful moving forward.

Lemma 5.1 (Largest term dominates). $\sum_{i=1}^k i 2^{k-i} \leq 2^{k+1}$.

Proof. A straightforward induction on k proves exactly $\sum_{i=1}^k i 2^{k-i} = 2^{k+1} - k - 2$. $\square$

Now we are ready to give the proof of Lemma 5.2. The main idea in the proof is just to consider whether $\lg n$ is bigger than $\lg m$ or vice versa; both cases ultimately appeal to Lemma 5.1 for different choices of k .

Lemma 5.2 (Level sum). $\sum_{d < \lg n} 2^d \lg(m/2^d) \in O(n + m)$.

Proof. Let $D \triangleq \lg n$ and $B \triangleq \lg m$. We have $\lg(m/2^d) \leq B - d$ (truncated at zero), so it suffices to bound $\sum_{d < D} 2^d (B - d)$. Here we case on which is bigger: D or B .

Case $D > B$. Every term with $d \geq B$ is zero, so reindexing by $i = B - d$ and applying Lemma 5.1, we have $\sum_{d < D} 2^d (B - d) = \sum_{i=1}^B i 2^{B-i} \leq 2^{B+1} \leq 4m + 2$.

Case $D \leq B$. Now no term is truncated. Split $B - d = (B - D) + (D - d)$ and reindex the second part by $i = D - d$:

$$\begin{aligned} \sum_{d < D} 2^d (B - d) &= \sum_{d < D} 2^d (B - D) + \sum_{d < D} 2^d (D - d) \\ &= (B - D)(2^D - 1) + \sum_{i=1}^D i 2^{D-i}. \end{aligned}$$

For the first term, $B - D \leq 2^{B-D}$ because $x \leq 2^x$, so it is at most $2^{B-D} 2^D = 2^B \leq 2m + 1$. The second is at most $2^{D+1} \leq 4n + 2$ by Lemma 5.1. Either way the sum is $O(n + m)$. $\square$

References

Alexandre Moine, Sam Westrick, and Joseph Tassarotti. 2026. A Separation Logic for Parallel Time Complexity with Work and Span Credits. *Proc. ACM Program. Lang.* 10, ICFP, Article 281 (Aug. 2026), 30 pages. doi:10.1145/3828679