Karlsruhe Institute of Technology

Core-Count Independent Reproducible Reduce

Bachelor's Thesis of

Christoph Stelz

at the Department of Informatics
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. Alexandros Stamatakis
Second reviewer:	Prof. Dr. Peter Sanders
Advisor:	M.Sc. Lukas Hübner

01. December 2021 – 01. April 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 01.04.2022

. . . .

(Christoph Stelz)

Abstract

Because of rounding errors, parallel floating-point summation can produce different results on different core-counts. For some algorithms like hill climbing, RAxML-NG [7] or greedy algorithms, this implies that results may be irreproducible with different core-counts. We present the Binary Tree Reduction algorithm, which follows a distributed binary tree scheme that keeps the calculation order fixed and independent of the core-count p. A naive implementation requires up to $(p-1) * (\log_2\left(\frac{N-1}{p}\right) + 1)$ messages to sum N floating-point numbers. To reduce the message count, we introduce a message buffer and optimize data distribution across the cores, the latter results in a runtime decrease of $18\,\%$. We find that for $p = 256$, Binary Tree Reduction has a slowdown of less than 2 compared to a naive, irreproducible solution. It is able to compute the sum of $N \approx 21 * 10^6$ summands on $p = 256$ cores in about $248\,\mu\text{s}$.

Zusammenfassung

Die Addition von Gleitkommazahlen kann aufgrund von Rundungsfehlern bei unterschiedlicher Prozessorenanzahl zu unterschiedlichen Ergebnissen führen. Für manche Algorithmen wie RAxML-NG [7] oder Greedy-Algorithmen kann dies den Verlust der Reproduzierbarkeit bei unterschiedlicher Prozessorenanzahl bedeuten. Wir stellen einen Reduktionsalgorithmus vor, der nach dem Schema eines verteilten Binärbaums vorgeht, wodurch die Ausführungsreihenfolge unabhängig von der Prozessorenanzahl p bleibt. Eine naive Implementierung muss bis zu $(p-1) * (\log_2(\frac{N-1}{p}) + 1)$ Nachrichten senden, um N Gleitkommazahlen zu addieren. Um die Nachrichtenanzahl zu senken führen wir einen Nachrichtenpuffer ein und optimieren die Datenverteilung über die Prozessoren, wobei letzteres zur einer Verringerung der Laufzeit um 18 % führt. Wir stellen fest, dass für $p = 256$ Prozessoren die Laufzeit des Binärbaum-Reduktionsalgorithmus weniger als 200 % der eines naiven, unreproduzierbaren Algorithmus entspricht. Die Binärbaum-Reduktion ist imstande $N \approx 21 * 10^6$ Summanden auf $p = 256$ Prozessoren in 248 µs aufzusummieren.

I would like to thank Prof. Dr. Stamatakis and Lukas Hübner for their enlightening advice and guidance, my parents for their continued support throughout my studies, and Katrin Anne Mertes and Max Lennart Steiert for proofreading this thesis.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

A common problem in massively parallel computations is the reduction (for example summation) of results over the entire cluster. Widespread implementations do not account for cluster-size-independent reproducibility and will deliver different results even if the only variable element is the number of participating Processing Elements (PEs). Irreproducibility of reduction operations propagates upwards into the results of high-level scientific software packages, impeding researchers abilities to understand and exchange these results.

RAxML-NG [7] for example is a software package that searches for the most likely phylogenetic trees based on biological input sequences. Given the exponentially large number of possible trees (for 100 taxa there exist over 10^{182} distinct phylogenies [9]) and proven $\mathcal{NP}$-hardness of the problem [8], an exhaustive tree search is infeasible. Instead, RAxML-NG uses stochastic evolutionary models to determine the likelihood of a given tree and performs a tree search to find a maximum likelihood estimate. Because of their small magnitude, programs usually deal with likelihood values logarithmically. To increase execution speed, RAxML-NG assumes that different sites evolve independently and computes per-site likelihoods in parallel on different threads. The overall likelihood of a tree is the product of all per-site likelihoods, therefore the tree log-likelihood is the sum of all Per-Site Log-Likelihoods (PSLLHs):

$$L_{\text{tree}} = \prod_{s \in \text{sites}} L_s \tag{1.1}$$

$$\log L_{\text{tree}} = \log\left(\prod_{s \in \text{sites}} L_s\right) = \sum_{s \in \text{sites}} \log L_s \tag{1.2}$$

The search path is chosen based on the log-likelihood of the current candidate tree. Therefore, the correctness of sum (1.2) is critically important for the resulting trees. RAxML-NG uses IEEE 754 floating-point numbers [6] to represent PSLLHs values, but floating-point arithmetic is not necessarily associative due to rounding errors [5].

Darriba et al. [2] have shown that because of different summation orders, executing the same version of RAxML-NG with the same input data and same random seed can still produce different trees if the number of threads varies. Diethelm [3] reports on the irreproducibility of a software used to simulate sheet metal forming and identifies two common causes: sums whose order is determined by the time in which PEs finish intermediate results and different propagation of rounding errors for a varied number of PEs. Wiesenberger et al. [11] study the irreproducibility of the FELTOR software package used for fluid simulations due to floating-point non-associativity and counter the problem by deriving bitwise-reproducible subroutines.

1.2. Preliminaries

Let $\mathbb{F}$ be the set of floating-point numbers. Given

- a cluster of p PEs indexed with a rank $i \in \{0, \ldots, p-1\}$ and interconnected by a Message Passing Interface (MPI)
- n_i floating-point numbers (elements) on the PE with index i ($N := \sum_{i=0}^{p-1} n_i$ in total)
- a not necessarily associative binary operation $\circ : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$

we want to reduce all numbers by means of $\circ$ so that the end result is bitwise-reproducible. A reduction algorithm is bitwise-reproducible if multiple executions over the same set of numbers with a variable number of PEs produce bit-per-bit identical results.

In order to correctly distribute the N elements over p PEs, we need to deal with cases where N is not divisible by p. Let $a := \lfloor \frac{N}{p} \rfloor$ be the rounded number of elements per PE. We can assign the remaining $N \bmod p$ elements to the upper or lower processes:

$$n_i^{\text{lower}} = \begin{cases} a+1 & \text{if } i < N \bmod p \\ a & \text{otherwise} \end{cases} \tag{1.3}$$

$$n_i^{\text{upper}} = \begin{cases} a & \text{if } i < N - (N \bmod p) \\ a+1 & \text{otherwise} \end{cases} \tag{1.4}$$

1.3. Related Work

Because of the abundance of reduction operations in modern high-performance-computing applications, there already exist multiple reproducible reduction algorithms.

1.3.1. Sequential Left-to-right Reduction

A naive approach to solving above problem is to gather all elements on a single PE and then apply the reduction operation strictly from left to right:

$$x_0 \circ x_1 \circ x_2 \circ \ldots \circ x_{N-1} = ((x_0 \circ x_1) \circ x_2) \circ \ldots \tag{1.5}$$

While simple in implementation, this approach does not benefit from parallelization. It requires $O(T_{\text{Gather}} + N)$ time. Because of the communication overhead, performance decreases with an increasing number of PEs.

1.3.2. Reproducible Accumulators

For floating-point summation in particular, Ahrens et al. [1] have developed an algorithm that uses a 48 Byte reproducible accumulator to avoid unpredictable rounding errors. After reading all the input data, the summation can occur in parallel in no particular order and still produces bitwise identical results. This requires around $9N$ floating-point operations and $3N$ bitwise operations. The Reproducible Basic Linear Algebra Subprograms (ReproBLAS) software package implements this algorithm and exposes it via a user-friendly API.

This approach is not suitable for general reduction operations, since it depends on specific properties of floating-point numbers as specified in the IEEE 754 standard [6] and is specific to summation.

1.3.3. Reduction Tree

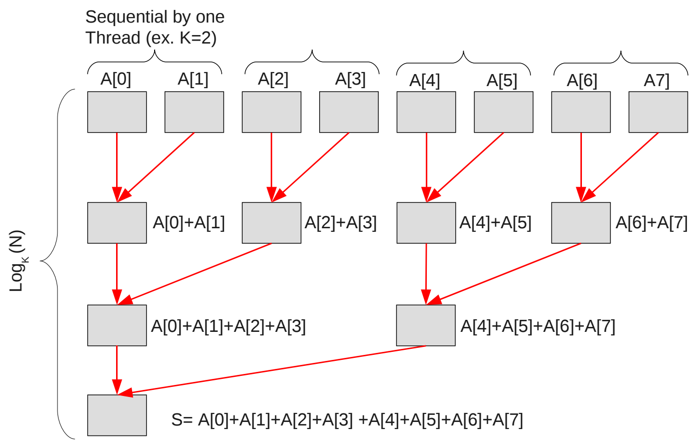
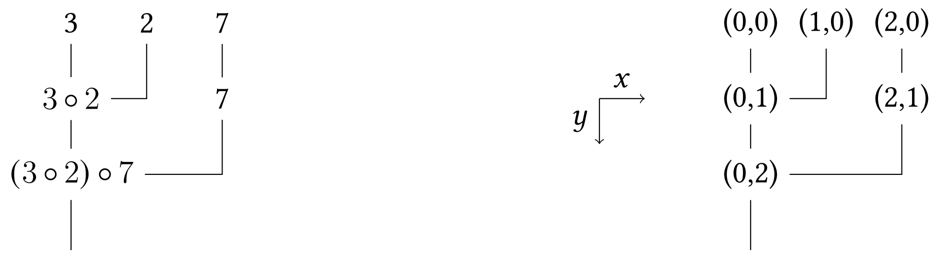


Figure 1.1.: General reduction tree (figure from Villa et al. [10]).

Villa et al. [10] utilize a K-ary tree structure on a Cray XMT system to sum floating-point numbers reproducibly (Figure 1.1). Using parallel-prefix accumulation, they compute the sum of N summands in $\log_K(N)$ steps, where K determines the amount of numbers the algorithm accumulates sequentially in each step. The reproducibility stems from the fact that the reduction tree depends only on the total number of summands N and the constant K, therefore the algorithm uses the same calculation order if the core-count differs.

The original source code is not available even after contacting the authors, therefore implementation details are unknown and runtime comparisons hardly possible.

2. Binary Tree Reduction



(a) Reduction of the elements 3, 2, and 7

(b) Corresponding coordinate system

Figure 2.1.: Example reduction tree for $N = 3$.

In this chapter we present a special case (where $K = 2$) of the reduction tree from Section 1.3.3. To reduce N elements, we construct a binary tree with N leaf nodes, each corresponding to a single element. By iteratively connecting adjacent nodes, we produce inner nodes that represent the intermediate result obtained by reducing their children. After $\lceil \log_2 N \rceil$ levels, the reduction of all elements into a single root node is complete. Figure 2.1a demonstrates this reduction scheme. In the first level, we construct an inner node that represents the reduction of the numbers 3 and 2. Since 7 has no adjacent element, the next inner node has only one child node and propagates the value along the tree unchanged. In the second and final layer, the root node represents the reduction of the two remaining elements $3 \circ 2$ and 7, producing the final result $(3 \circ 2) \circ 7$.

We can uniquely identify nodes by using two-dimensional **coordinates** (x, y). The leaf nodes have the coordinates $(0, 0)$ through $(N - 1, 0)$. To obtain the coordinates of inner nodes, we simply take over the x-coordinate of their left child node and increment the y-coordinate by 1. Figure 2.1b shows coordinates for all nodes in a binary tree with $N = 3$ leaf nodes. An element has index i if its corresponding leaf node has the coordinates $(i, 0)$.

The absolute difference of x-coordinates of the child nodes of inner nodes doubles with each level, thus for an inner node with y-coordinate y the absolute difference is equal to 2^{y-1}. By differentiating three cases, we define a recursive reduction function that adheres to the above tree reduction scheme:

$$\text{reduce}(x, y) = \begin{cases} \text{element with index } x, & \text{for } y = 0 \qquad (2.1) \\ \text{reduce}(x, y - 1), & \text{for } x + 2^{y-1} \geq N \qquad (2.2) \\ \underbrace{\text{reduce}(x, y - 1)}_{\text{left child}} \circ \underbrace{\text{reduce}(x + 2^{y-1}, y - 1)}_{\text{right child}}, & \text{otherwise} \qquad (2.3) \end{cases}$$

Equation (2.1) defines the base case for leaf nodes, where no further reductions are necessary. If N is not a power of 2, there will not always be an adjacent element (as in Figure 2.1a). In this case, the inner node has only one child node whose value we can directly return (Equation (2.1)). Finally, Equation (2.1) defines the recursive reduction strategy for inner nodes with two child nodes.

We can express the entire reduction by applying the reduce-function to the root node $(0, \lceil \log_2 N \rceil)$. Consider the example in Figure 2.1a:

$$\begin{aligned}
\mathrm{reduce}(0, \lceil \log_2 N \rceil) &= \mathrm{reduce}(0,2) && \text{Apply (2.3)}\\
&= \mathrm{reduce}(0,1) \circ \mathrm{reduce}(2,1) && \text{Apply (2.3)}\\
&= (\mathrm{reduce}(0,0) \circ \mathrm{reduce}(1,0)) \circ \mathrm{reduce}(2,1) && \text{Apply (2.2)}\\
&= (\mathrm{reduce}(0,0) \circ \mathrm{reduce}(1,0)) \circ \mathrm{reduce}(2,0) && \text{Apply (2.1)}\\
&= (3 \circ 2) \circ 7
\end{aligned}$$

Binary Tree Reduction and the sequential left-to-right reduction from Section 1.3.1 require an equal amount of reduction operations. Binary Tree Reduction requires more memory for out-of-place operations where the input data can not be overwritten, since a single accumulator does not suffice to store all intermediate results. We expand our model to account for PE-boundaries. We split up our binary tree across multiple PEs by distributing the elements. A node with coordinates (x, y) belongs to the PE which holds the element with index x. Figure 2.2 shows the distribution of nine elements across three PEs.

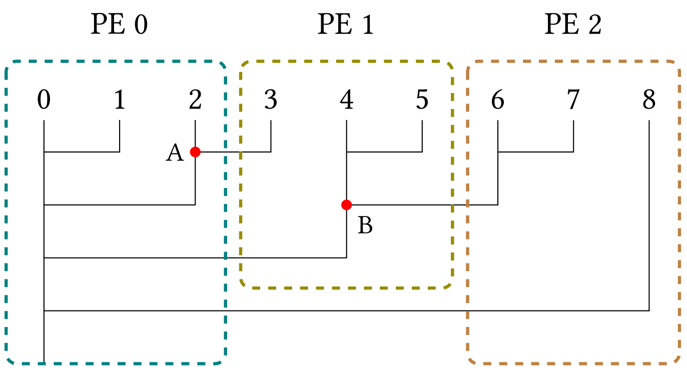


Figure 2.2.: Distributed binary tree with $N = 9$ leaf nodes and $p = 3$ PEs. The red dots indicate two PE-intersecting nodes.

Unlike sequential left-to-right reduction, Binary Tree Reduction can execute the reductions $(0) \circ (1)$, $(4) \circ (5)$ and $(6) \circ (7)$ in parallel, since there exist no data dependencies between them. Some calculations require communication between the PEs: a node is PE-intersecting if its child nodes belong to distinct PEs. We define an outbound subtree root as the right child node of a PE-intersecting node, since the algorithm must send its value over the MPI. In Figure 2.2, node A with coordinates $(2, 1)$ and node B with coordinates $(4, 2)$ are examples of PE-intersecting nodes. In this case, nodes $(3, 0)$ and $(6, 1)$ are outbound subtree roots.

2.1. Message Counts

The main barrier to efficient parallelization as outlined in the previous section is the need for synchronization between PEs because of data dependencies in the form of PE-intersecting nodes. The target implementation utilizes the MPI to communicate between different PEs and since the messages are small (one double precision floating-point value occupies 8 bytes, or 64 bits), the number of messages between PEs dominates the communication overhead. Under the assumption that messages are not bundled together by means of a message buffer, the message count is equal to the number of PE-intersecting nodes.

Figure 2.3 shows the number of messages depending on the dataset size for the two types of distributions introduced in Equation (1.3) and Equation (1.4). The functions display large differences in message counts for small differences in dataset sizes, but follow a trend akin to a logarithmic function. For $N > p$, the lower bound of the message count is $p - 1$, since all PEs with a rank larger than 0 must have at least one outbound subtree root in order for its elements to be included in the final result. For the n_i^{lower}-distribution (Equation (1.3)), the message count attains the global minima at $N = 2^i * p$, where $i \in \mathbb{N}$. In this case, each PE holds an independent subtree of size 2^i whose root node is part of a PE-intersecting node. If we increase our dataset size to $N + 1$, applying the n_i^{lower} distribution will shift all PE-boundaries one step in x-direction because of the additional element on the PE with rank 0. The subtree root nodes are now on different PEs than their children, producing $i+1$ PE-intersecting nodes on all PEs except the first one, as illustrated in Figure B.5. Therefore, the message count for $N = 2^i * p + 1$ is

$$(p - 1) * (i + 1) \tag{2.4}$$

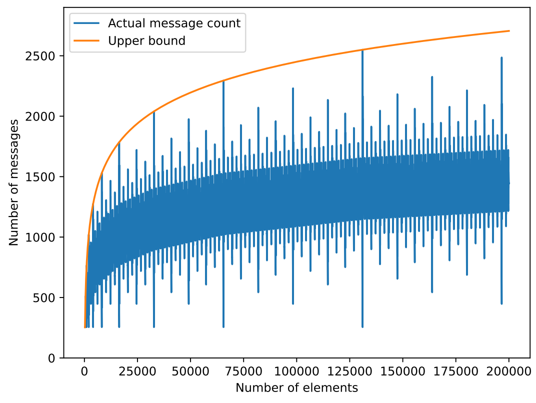
Furthermore,

$$\begin{aligned} &N = 2^i * p + 1 \\ \Leftrightarrow\ &N - 1 = 2^i * p \\ \Leftrightarrow\ &\frac{N-1}{p} = 2^i \\ \Leftrightarrow\ &i = \log_2\left(\frac{N-1}{p}\right) \end{aligned}$$

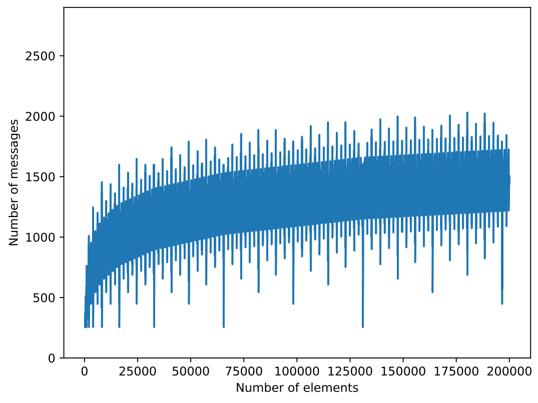
If we substitute i in Equation (2.4), we get the following closed-form upper bound on the message count for the n_i^{lower} distribution:

$$M(N) = (p - 1) * (\log_2\left(\frac{N-1}{p}\right) + 1) \tag{2.5}$$

Figure 2.3a displays a plot of Equation (2.5) in orange color. Under the assumption that the number of PEs is constant, the message count M is in $O(\log(N))$. Note that for the n_i^{upper} distribution in Figure 2.3b, the corresponding spikes are much less pronounced since the subtree shifting described above does not occur when adding only one element. Because of the lower upper bound, implementations should therefore prefer the n_i^{upper} distribution over the n_i^{lower} distribution.



(a) Remaining elements assigned to the lower ranks (n_i^{lower})



(b) Remaining elements assigned to the higher ranks (n_i^{upper})

Figure 2.3.: Simulated message counts for different dataset sizes on a cluster with $p = 256$ PEs.

3. Implementation

In this chapter, we deduce the necessary equations to implement the reduction algorithm and present multiple optimizations. We focus on summation, the reduction operator $\circ$ will be floating-point addition. While the recursive formula given in Equation (2.1) already defines a reduction algorithm, its implementation in imperative languages would heavily rely on the call stack to store intermediate results. In practice, reducing iteratively from leaf nodes to the root yields faster runtimes.[1]

The binary representation of element indices has $\lceil \log_2 N \rceil$ bits. If we order them from most- to least-significant and interpret them as a series of decisions, where 0 means "go up" and 1 means "go right and up", each index encodes a path from the tree root to the corresponding leaf node:

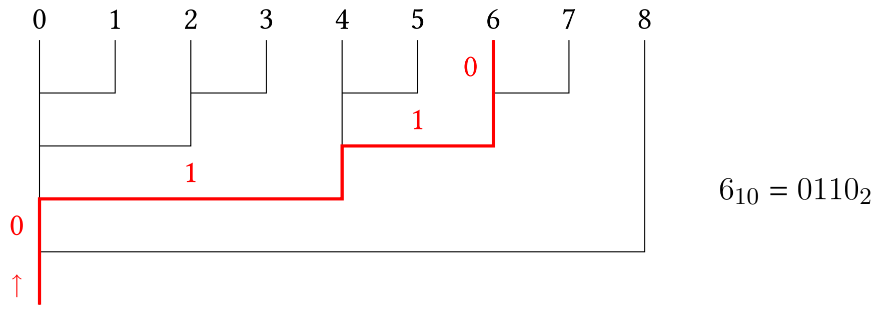


Figure 3.1.: Example path for index 6 with $N = 9$.

For any given x-coordinate $x > 0$, the **maximum y-coordinate** $\max_y(x)$ is equal to the number of trailing zeros of x, i.e. the zero-indexed position of the least-significant bit set in x. We denote this expression as $\text{ffs}(x) - 1$, where ffs is short for "find first bit set". The path representation offers intuition on why the equality holds: the least-significant bit set is the last time the x-coordinate changes along the path from the root, since all following bits are zero and encode the decision "go up".

To find the x-coordinate of the **parent node** of an inner node $(x, \max_y(x))$, we replace the last (least-significant) "go right and up" decision with "go up". Numerically, this is equivalent to cancelling the least-significant bit of x, which can be efficiently calculated using the bitwise AND-operation $'\&'$:

$$\mathit{parent}(x) := x \,\&\, (x - 1) \tag{3.1}$$

[1]For $N = 2^{27}$ elements on $p = 1$ PE, we observe a 4.9 speedup over the recursive reduction (Microbenchmark ID 5).

Each PE with rank i (where $i \in [0, p-1]$) stores n_i consecutive elements. The global index of the first element assigned to a PE is the so-called start index, and it is equal to the prefix sum of the assigned number of elements:

$$\mathit{startIndex}(0) = 0 \tag{3.2}$$

$$\mathit{startIndex}(i) = \sum_{j=0}^{i-1} n_j \tag{3.3}$$

This allows us to define the function *rankFromIndex*, which computes the rank of the PE that stores the given element:

$$\mathit{rankFromIndex}(x) = \max \{i \in [0, p-1] \mid \mathit{startIndex}(i) \leq x\} \tag{3.4}$$

For any given set X of x-coordinates, we can use the *rankFromIndex*-function to determine the x-coordinates of outbound subtree roots, i.e. nodes whose parent node lies on a different PE:

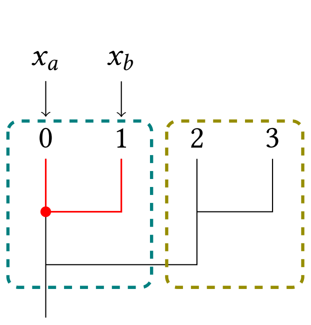
$$I_{\text{PE-intersecting}}(X) = \{x \in X \mid \mathit{rankFromIndex}(x) \neq \mathit{rankFromIndex}(\mathit{parent}(x))\} \tag{3.5}$$

largestSubtreeChildIndex returns the largest x-coordinate of all child nodes of the given subtree root, which is equal to setting all trailing zeros of the given index to 1. Our implementation calculates this using the bitwise OR-operation "|":

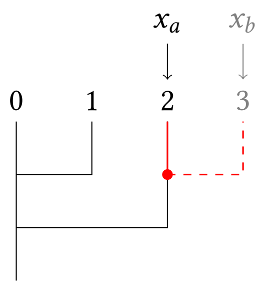
$$\mathit{largestSubtreeChildIndex}(\mathit{index}) = \mathit{index} \mid (\mathit{index} - 1) \tag{3.6}$$

Algorithm 1: Summation procedure

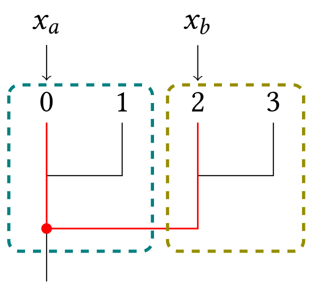
```
Data: PE-rank $rank$, $n_{rank}$ summands with coordinates $(startIndex, 0)$ through
      $(startIndex + n_{rank} - 1, 0)$
Result: Reduction result on the PE with rank $0$
$startIndex \leftarrow startIndices(rank)$
$endIndex \leftarrow startIndex + n_{\text{rank}}$
if $rank = 0$ then
    $outboundSubtreeRoots \leftarrow \{0\}$
else
    $outboundSubtreeRoots \leftarrow I_{\text{PE-intersecting}}([startIndex, endIndex))$
end
for $i \leftarrow outboundSubtreeRoots$ do
    // Reduce subtree level-by-level
    for $y \leftarrow [1, \max_y(x)]$ do
        $x \leftarrow i$
        while $x \leq largestSubtreeChildIndex(x)$ do
            $x_a \leftarrow x$                      // $x$-coordinate of left child node
            $x_b \leftarrow x + 2^{y-1}$            // $x$-coordinate of right child node
            $a \leftarrow (x_a, y - 1)$             // value of left child node
            if $x_b \geq N$ then
                // No adjacent node, passthrough
                $(x, y) \leftarrow a$
            else if $rankFromIndex(x_b) \neq rank$ then
                // PE-intersecting node, fetch over MPI
                $b \leftarrow$ receive $(x_b, \max_y(x_b))$ from $rankFromIndex(x_b)$
                $(x, y) \leftarrow a + b$
            else
                $b \leftarrow (x_b, y - 1)$         // value of right child node
                $(x, y) \leftarrow a + b$
            end
            $x \leftarrow x + 2^{y-1}$
        end
    end
    if $rank \neq 0$ then
        send $(i, \max_y(i))$ to $rankFromIndex(parent(i))$
    end
end
```



(a) Line 21: simple summation of inner nodes



(b) Line 16: x_b exceeds the number of elements $N = 3$



(c) Line 18: x_a and x_b point to elements located on different PEs

Figure 3.2.: The three distinctions in the inner loop of Algorithm 1.

Algorithm 1 shows the procedure that each PE follows. It consists of three nested loops. The most outer loop iterates over all PE-intersecting indices in ascending order; its body is responsible for the reduction of the subtree rooted at the corresponding PE-intersecting node. Because our reduction scheme is a left-leaning binary tree, nodes with lower indices occur earlier in the reduction equation, therefore processing PE-intersecting indices in ascending order minimizes wait-times for other PEs.

The inner loops in Line 9 and Line 11 implement the leaf-to-root scheme that reduces adjacent nodes level-by-level. The distinctions made in Equations (2.1)–(2.3) give rise to a series of conditional expressions: Line 16 deals with inner nodes that have no adjacent node (Figure 3.2b), Line 18 distinguishes between PE-intersecting nodes and local inner nodes (Figure 3.2c) and Line 21 performs the local reduction of adjacent elements (Figure 3.2a).

3.1. Applied Optimizations

The implementation used in Chapter 4 follows Algorithm 1 with the following optimizations. Message Buffering (Section 3.1.1) allows to reduce the number of messages sent across the network and therefore reduces the communication overhead. The number of messages further decreases with a data distribution optimized for the reduction algorithm (Section 3.1.2). Vectorizing the additions (Section 3.1.4) yields the largest performance gain of all optimizations presented in this section.

3.1.1. Message Buffering

Typically, Algorithm 1 sends multiple consecutive messages to the same target PE. Figure 3.3 is a minimal example for this observation: nodes $(3, 0)$ and $(4, 1)$ are outbound subtree roots that PE 1 sends to PE 0. PE 1 can avoid the additional message latency if it does not send $(3, 0)$ directly, but stores it in a buffer instead. Then, after finishing the computation of $(4, 1)$, PE 1 can send both results to PE 0 in a single message. As long as the communication overhead for sending a message sufficiently exceeds the time needed to sum two floating-point numbers, the delayed transmission of $(3, 0)$ does not impact the runtime negatively.

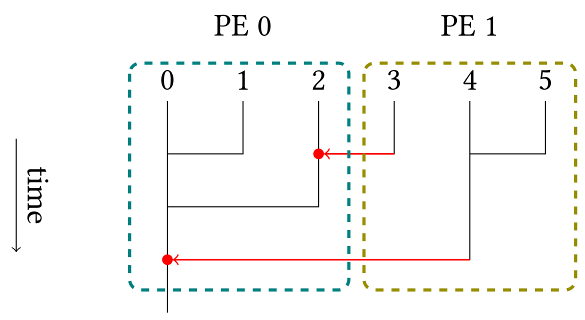


Figure 3.3.: Distributed tree with $N = 6$ nodes and $p = 2$ PEs.

The current implementation utilizes a buffer with a maximum of 4 elements per message. After the summation routine has computed an outbound subtree root, it places the result in the outbound message buffer. Flushing of the buffer occurs in either one of the following cases:

- The summation procedure inserts another outbound subtree root into the buffer which has a different target PE.
- The summation procedure begins work on an outbound subtree root whose subtree size is greater than 64 summands. This guarantees an upper limit on the time a finished result spends inside the buffer.

The buffer utilization averages about 1.2 summands per message. Figure B.1 shows that relaxing above flushing criteria does not yield a runtime benefit, presumably because of the induced latency.

3.1.2. Data Distribution

If the user can arbitrarily assign elements to PEs, multiple optimization techniques arise. To quantify them, we propose the following model:

$$\text{Score} = t_{\text{MPI\_Send}} * n_{\text{PE-intersecting nodes}} + \max\{i \in [0, p-1] \mid n_i * t_{\text{add}}\} \tag{3.7}$$

$t_{\text{MPI\_Send}}$ is an estimate of the time needed to send a single element between two PEs, t_{add} estimates the time needed to add two floating-point values with double precision (64 bit). On a shared-memory machine,[2] these estimates were empirically measured to be $t_{\text{MPI\_Send}} \approx 281ns$ and $t_{\text{add}} \approx 4.15ns$. While this model does not take any data dependencies between PEs into account, it balances the performance gain achieved by parallelization (represented by a low maximum on the right side of Equation (3.7)) against the communication overhead caused by binary tree fragmentation.

In this section we evaluate multiple approaches to the optimization of the data distribution with an example dataset with $N = 504\,850$ summands and $p = 256$ PEs.

[2]Refer to Chapter 4 for hardware specifications.

3.1.2.1. Efficient message count determination

To calculate the score in Equation (3.7), an algorithm must efficiently determine the number of PE-intersecting nodes. A naive approach would be to check whether $\mathit{rankFromIndex}(\mathit{parent}(i)) \neq \mathit{rankFromIndex}(i)$ for each element $i \in \{0, \ldots, N-1\}$, requiring $O(N)$ time.

Algorithm 2 represents a faster alternative requiring $O(p + \log_2 N)$ time.

Algorithm 2: Message count solver

```
Data: Distribution $n_i$, where $i \in \{0, \ldots, p-1\}$
Result: Number of PE-intersecting nodes
$startIndices \leftarrow \{\sum_{i=0}^{j} n_i \mid j \in [1, p-1]\}$
$messageCount \leftarrow 0$
$i \leftarrow 0$
while $i + 1 < ranks$ do
    $startIndex \leftarrow startIndices[i]$
    $endIndex \leftarrow startIndices[i+1]$
    $index \leftarrow startIndex$
    while $index < endIndex$ do
        $messageCount \leftarrow messageCount + 1$
        $index \leftarrow largestSubtreeChildIndex(index) + 1$
    end
    $i \leftarrow i + 1$
end
return $messageCount$
```

3.1.2.2. Even distribution

Distributing the data evenly across the PEs ensures maximum parallelization of the computational effort, since the maximum difference in the workload of PEs is 1 element. As shown in Table 3.1, distributing the elements evenly yields a score of around 400 µs. The lower expected worst-case message count of the n_i^{upper}-distribution compared to the n_i^{lower} distribution described in Section 2.1 expresses itself in both a lower score and a lower message count.

3.1.2.3. Round down to power of 2

The even distribution does not take the binary tree structure into account and may place PE-boundaries at places which produce numerous PE-intersecting nodes. By rounding

Table 3.1.: Scores for the even data distribution ($N = 504\,850$, $p = 256$).

Distribution	Score	Message count
n_i^{lower}	469.0 µs	1640
n_i^{upper}	401.9 µs	1401

Table 3.2.: Score of the n_i^{power2} data distribution ($N = 504\,850$, $p = 256$).

Distribution	Score	Message count
n_i^{power2}	1083.4 µs	256

down the number of assigned elements to the nearest power of 2, we obtain start indices with a lot of trailing zeros, which is desirable since they produce larger PE-local subtrees and reduce the number of PE-intersecting nodes. Equation (3.8) describes this approach formally. The element count on the first $p-1$ PEs is a power of 2. By placing the remaining elements on the last PE, we ensure that an uneven number of remaining elements does not reduce the trailing zeros of the start indices of the other PEs.

$$n_i^{\text{power2}} = \begin{cases} 2^{\lfloor \log_2 \frac{N}{p} \rfloor} & i < p-1 \\ N - \sum_{i=0}^{p-2} n_i^{\text{power2}} & i = p-1 \end{cases} \tag{3.8}$$

The rounding decreases the amount of messages considerably (Table 3.2). With our test dataset, the message count is in the vicinity of the lower bound of $p-1$, an 80% reduction compared to the even distribution. This comes at the cost of an imbalanced element distribution, the PE with the highest rank, which stores the remaining elements, contains about half the elements. The score function penalizes the inefficient parallelization with the second term, causing the score to be 2.7× higher compared to the even distribution.

3.1.2.4. Optimized-Index Distribution

As seen in Section 3.1.2.3, unbounded distribution optimization yields small message counts at the expense of computational imbalance. In this section we will present a method to optimize the data distribution within certain bounds, to balance communication costs against computational costs.

Let *fairShare* $:= \frac{N}{p}$ be the approximate number of elements per PE in an equivalent even distribution. We want to optimize each *startIndex*(i) to produce the least amount of PE-intersecting nodes while keeping the differences in the computational workload between PEs small. We express this requirement as follows:

$$\forall i \in \{1, \ldots, p-1\} : \textit{startIndex}(i) - \textit{startIndex}'(i) \leq \alpha * \text{fairShare} \tag{3.9}$$

where *startIndex*$'(i)$ is the optimized start index and α is the maximum deviation relative to the even distribution. In Algorithm 3, we iteratively improve our start index by applying the *parent*-function (Equation (3.1)) until the proposed index violates Equation (3.9). We obtain the complete data distribution $n_i^{\text{optimized}}$ by executing Algorithm 3 on each start index of the n_i^{upper}-distribution.

Figure 3.5 shows the difference in assigned elements of the even distribution and the optimized distribution with a maximum deviation of $\alpha = 0.2$. The even distribution always assigns an equal number of elements to PEs, while the optimized distribution varies the number of elements according to the maximum deviation parameter α. Figure 3.4 shows a histogram of the summation duration for the two distributions. The runtime benefit of an

Algorithm 3: Index optimization procedure

```
   Data: PE-rank $i$, start index $j$, maximum deviation parameter $\alpha$
   Result: Optimized start index for the PE with rank $i$
1  $currentIndex = j$
2  $proposedIndex = currentIndex$
3  while $initialIndex - proposedIndex \leq \alpha * \frac{N}{p}$ do
4      $currentIndex = proposedIndex$
5      $proposedIndex = parent(initialIndex)$
6  end
7  return $currentIndex$
```

Table 3.3.: Score of the $n_i^{\text{optimized}}$ data distribution ($N = 504\,850$, $p = 256$).

Distribution	Score	Message count
$n_i^{\text{optimized}}$	184.5 µs	621

optimized data distribution is visible, with optimized distribution the algorithm calculates results about 4 µs faster than with an even distribution.

3.1.2.5. Drawbacks of the Scoring Function

The scoring function (3.7) has two major drawbacks: it does not consider the critical path of calculations and also assumes that the time $t_{\text{MPI\_Send}}$ is constant. This causes a discrepancy between predicted score times and measured benchmark times. Nonetheless, it provides a runtime model accurate enough so that distributions with a better score perform better under benchmark conditions.

3.1.3. Index-lookup Hashmap

Algorithm 1 uses the *rankFromIndex* function to lookup the PE-rank for a given element index. This function maps the binary tree structure to the underlying computing topology and its execution speed is performance-critical due to its position inside a frequently executed loop.

The initial implementation used a loop to find the first entry in the *startIndices* array which numerically exceeds the input index. The runtime of this algorithm is $O(p)$. Microbenchmarks revealed the linearly increasing runtime and the need for optimization (see Figure 3.6). The currently implemented version uses a red-black tree (std::map) to scan for the start index, which yields a runtime in $O(\log(p))$.

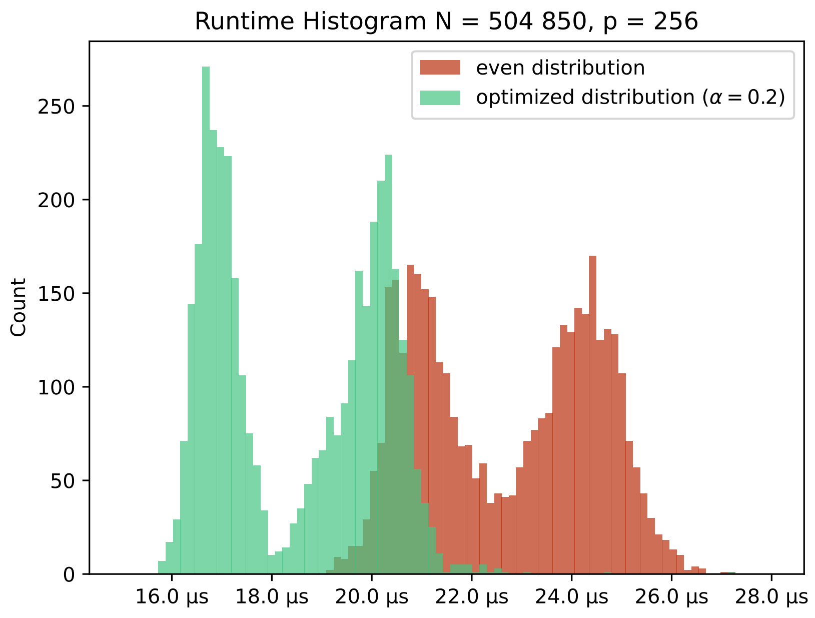


Figure 3.4.: Runtime comparison of even vs. optimized data distribution.

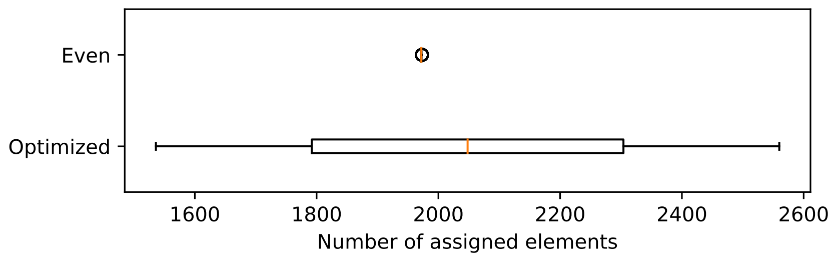


Figure 3.5.: Boxplot of the assigned number of elements per PE for the even and optimized data distribution.

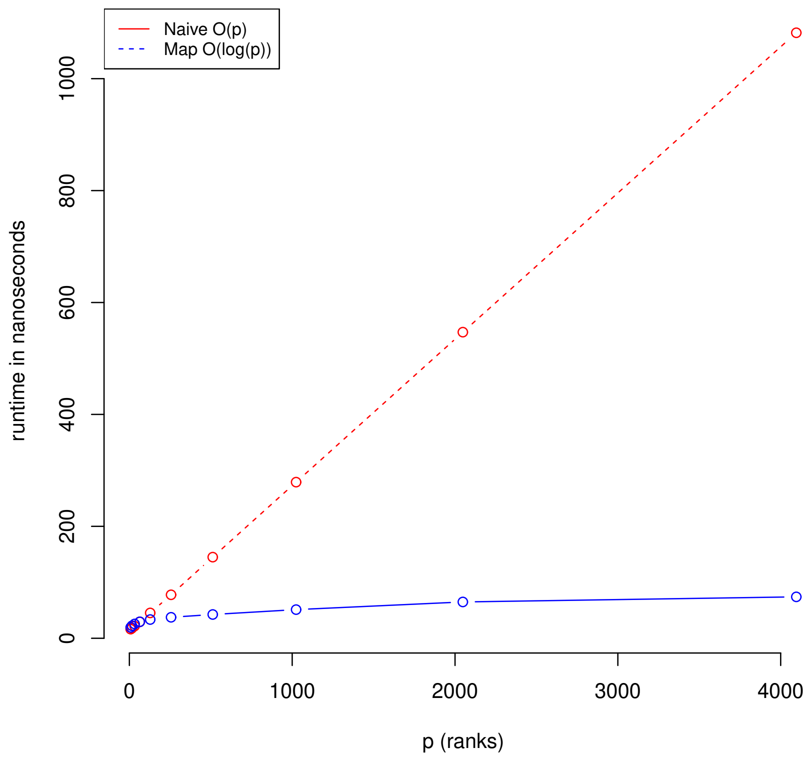


Figure 3.6.: Microbenchmark comparison of unoptimized and optimized *rankFromIndex* function.

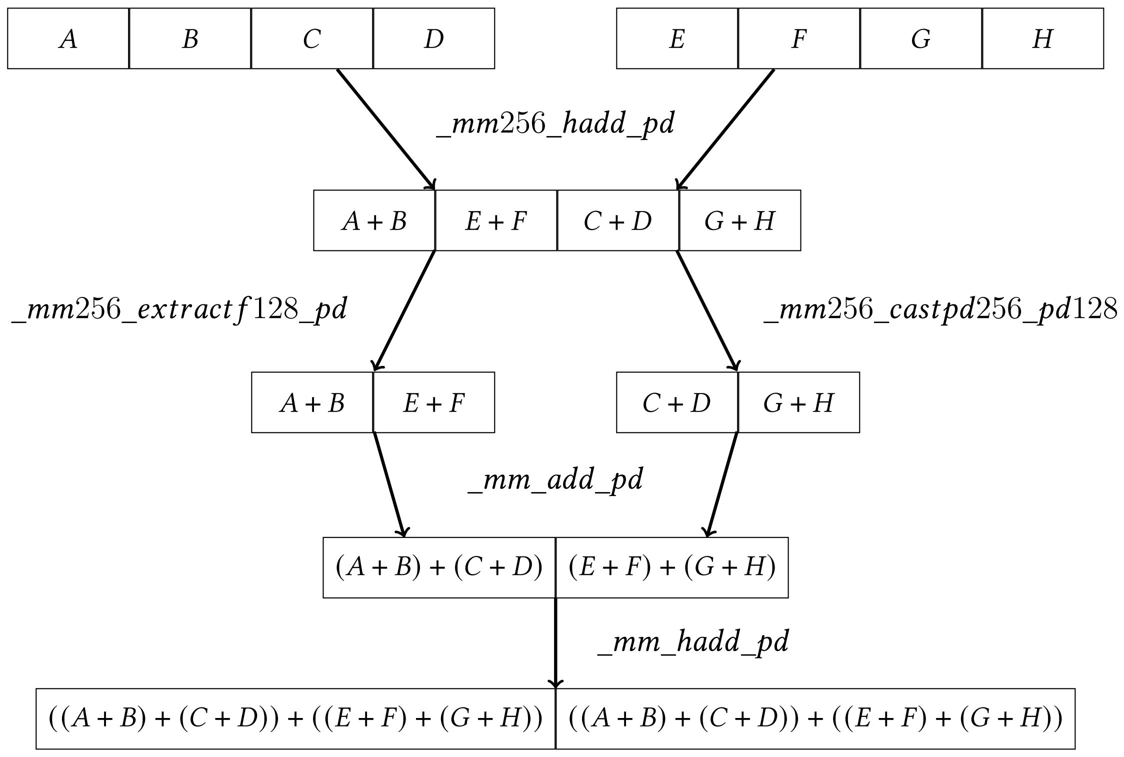


Figure 3.7.: Register content during AVX-2 subtree summation.

3.1.4. Vectorization

The theoretical per-node peak of Floating-Point Operations per Second (FLOPS) increases dramatically under the utilization of Single Instruction Multiple Data (SIMD) capabilities [4]. While modern compilers like the GNU C-Compiler try to automatically vectorize existing code,[3] optimization by hand can yield better results.

Compared to the simple left-to-right reduction presented in Section 1.3.1, a reduction tree lends itself better to parallelization, since an algorithm can reduce subtrees independently. Our implementation uses x86 Advanced Vector Extensions (AVX), specifically AVX-2. AVX-2 registers are 256 bits wide and can therefore store four double precision floating-point numbers. Algorithm 4 uses two registers to accumulate a subtree of eight elements at once. Because AVX-2 instructions manipulate data within 128-bit lanes, it is necessary to extract the upper 128-bit after the first horizontal add in order to follow the correct summation order. Figure 3.7 displays the register contents over time.

Our implementation uses Algorithm 4 as a subroutine inside Algorithm 1 to advance the iterative reduction three levels per iteration. If the remaining number of elements in the current iteration is not divisible by eight, the algorithm processes the remaining elements using non-vectorized instructions. Figure 3.8 compares the runtime of three accumulation algorithms: the initial recursive implementation of Binary Tree Reduction, the vectorized AVX-2 implementation and the std::accumulate routine from the C++ standard library.

[3]We confirmed manually that the relevant code sections compile to AVX machine instructions after passing the -mavx flag

Algorithm 4: 8-tree summation with AVX-2 instructions.

```
   Data: Buffer $buffer$ with at least 8 entries at offset $i$
   Result: Subtree sum of 8 elements
1  $a \leftarrow mm256\_load\_pd(buffer[i])$
2  $b \leftarrow mm256\_load\_pd(buffer[i+4])$
3  $level1Sum \leftarrow mm256\_hadd\_pd(a, b)$
4  $c \leftarrow mm256\_extractf128\_pd(level1Sum1)$
5  $d \leftarrow mm256\_castpd256\_pd128(level1Sum)$
6  $level2Sum \leftarrow mm\_add\_pd(c, d)$
7  $level3Sum \leftarrow mm\_hadd\_pd(level2Sum, level2Sum)$
8  return $mm\_cvtsd\_f64(level3Sum)$
```

Since only one PE executes this microbenchmark, no communication takes place and all runtime costs are purely computational. std::accumulate guarantees the summation order to be left-to-right and therefore can not be vectorized. For smaller workloads ($N < 64$), the overhead introduced by vectorization is larger than the performance gains, but for larger workloads the AVX-2 implementation outperforms std::accumulate by a factor of more than 2. For small inputs, it can be beneficial to switch to std::accumulate.

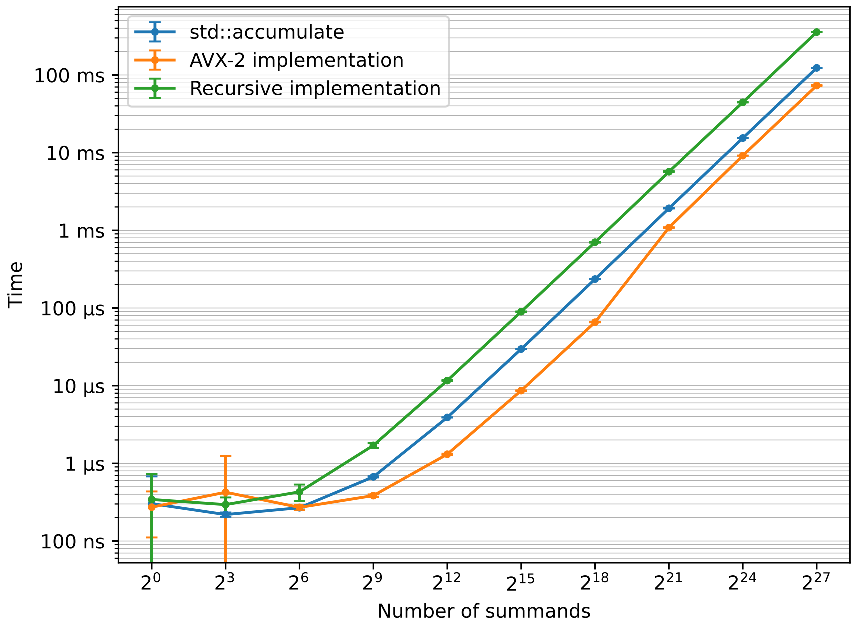


Figure 3.8.: Microbenchmark comparing sequential summation to AVX-2 binary tree reduction for $p = 1$.

4. Experiments

In this chapter, we compare the runtimes of three summation modes: The Binary Tree Summation algorithm as presented in Chapter 2 and Chapter 3, the ReproBLAS reduce operation and additionally, a bitwise-irreproducible implementation which uses std::accumulate to sum values locally and MPI_Allreduce for global reduction as baseline.

4.1. Experimental Setup

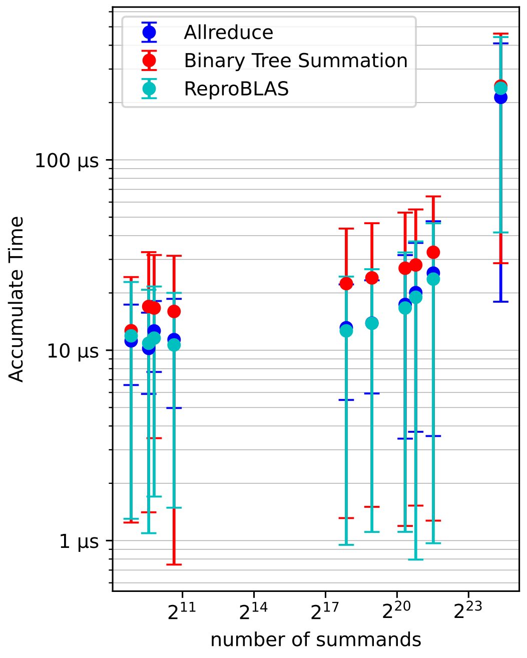
We run shared-memory benchmarks on a machine with two AMD EPYC 7713 CPUs with 64 cores each for a total of $p = 256$ PEs with hyper-threading. We execute distributed-memory benchmarks with more than 256 PEs on multiple thin nodes of the bwUniCluster 2.0 which have two Intel Xeon Gold 6230 with 40 cores (80 threads) per node.

Input data stems from RAxML-NG runs in the form of an array of double-precision floating point numbers representing PSLLH values. Dataset sizes range from 460 to 21 410 970 summands, as listed by Table 4.1.

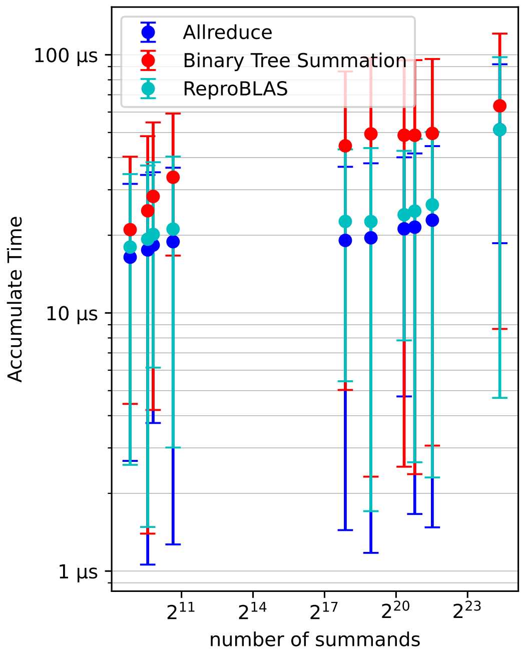
The benchmark execution performs the following steps for each summation mode (reproblas, allreduce, tree) and each dataset: First, it loads the input data from a file and distributes it among the PEs. Next, it performs the summation 100 times and measures the duration for each iteration. Finally, it discards the first and last eight measurements and outputs both the summation result and the remaining measurements.

Table 4.1.: Overview of benchmark datasets.

dataset name	**number of summands** N
354	460
multi100	767
prim	898
fusob	1 602
dna_rokasD4	239 763
aa_rokasA8	504 850
dna_rokasD1	1 327 505
aa_rokasA4	1 806 035
dna_PeteD8	3 011 099
dna_rokasD7	21 410 970



(a) Shared memory, $p = 256$



(b) Distributed memory, $p = 1280$

Figure 4.1.: Median accumulation time after 100 repetitions for different datasets. Error bars depict 1st and 99th percentile.

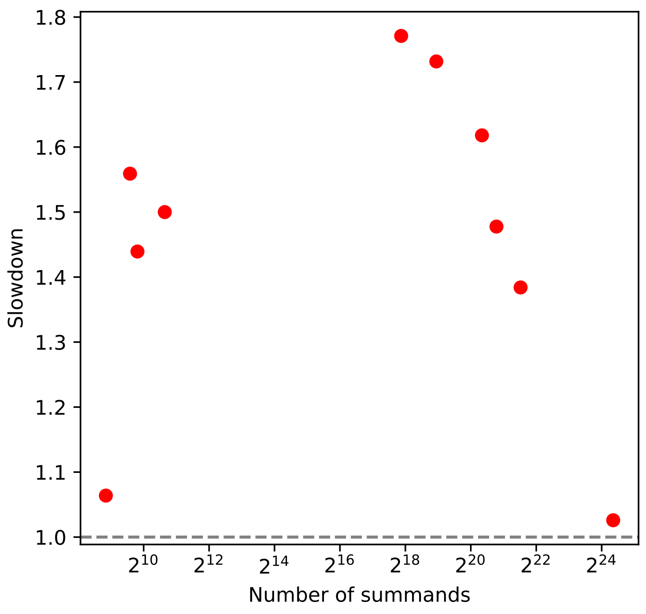


Figure 4.2.: Relative slowdown of Binary Tree Summation compared to ReproBLAS for $p = 256$ PEs.

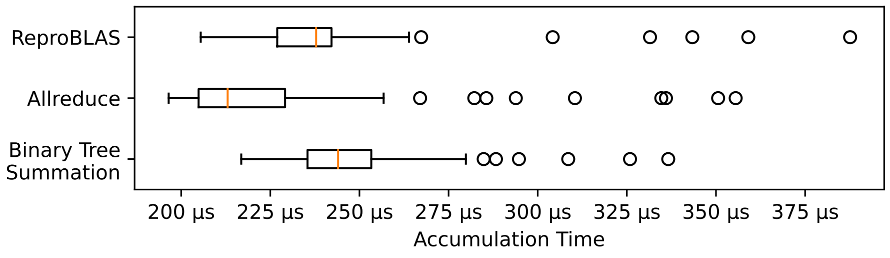


Figure 4.3.: Runtime distribution for all three summation modes on the dataset *rokasD7* ($N = 21\,410\,970, p = 256$). We removed *the* lowest and highest outlier for each accumulation mode.

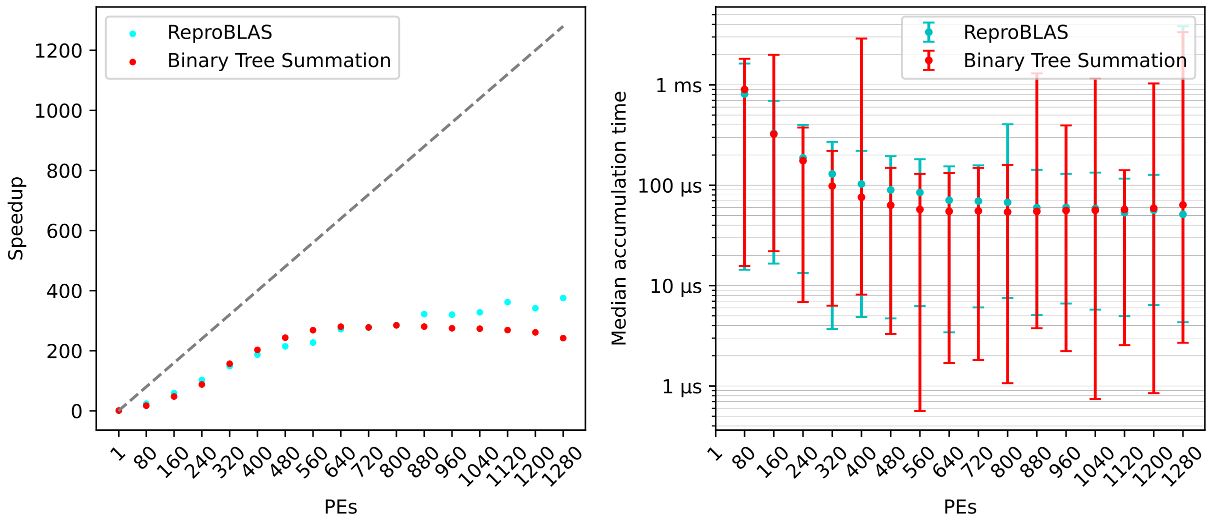


Figure 4.4.: Speedup and median accumulation time of $N = 21\,410\,970$ elements over p. Error bars depict 1st and 99th percentile.

4.2. Results

Figure 4.1 shows the runtime measurements across all datasets. We measure that all summation algorithms have a runtime that is linear in the number of summands (Figure 4.1). On the shared-memory machine, we measure a slowdown of less than 2 of Binary Tree Summation compared to ReproBLAS (Figure 4.2). On the largest dataset, Binary Tree Summation is only $2\,\%$ slower than ReproBLAS (Figure 4.3). The runtime of the irreproducible $\mathrm{std::accumulate}$ + Allreduce variant differs less than $15\,\%$ from the ReproBLAS runtime, possibly due to the missing vectorization of $\mathrm{std::accumulate}$ (Figure B.3). The performance of all three accumulation modes suffers from the increased number of PEs on the distributed-memory machine for small- and medium-sized datasets, as we observe no speedup for $N < 2^{22}$. Additionally, the logarithmically increasing message count of Binary Tree Summation further increases the gap to ReproBLAS for increasing PE-counts. Figure 4.4 shows the result of a strong-scaling benchmark on a distributed-memory machine. The speedup relative to the sequential computation levels off for $p \geq 640$ (about $33\,000$ elements per PE) for both ReproBLAS and Binary Tree Summation. Between 320 and 880 PEs the median accumulation time of Binary Tree Summation is smaller compared to ReproBLAS, but has a higher variance of runtimes.

4.3. Reproducibility of Results

To verify the reproducibility of the results, we ran all three summation algorithms with different core-counts in steps of 16 ($p \in \{1, 17, \ldots, 241\}$). With ReproBLAS and Binary Tree Summation, we detected no deviation between results for all test datasets, while values produced by Allreduce were already irreproducible between runs with $p = 1$ and $p = 17$ PEs. Table 4.2 shows the difference between the largest and smallest result collected over all values of p. For ReproBLAS and Binary Tree Summation, this value is zero,[1] indicating that the result is independent from the number of PEs. For Allreduce, the largest observed relative error was $3.9 * 10^{-13}$ (dataset *rokasD1*), which could potentially cause tools like RAxML-NG to dismiss certain trees during likelihood maximization leading to diverging tree searches. Results from ReproBLAS and Binary Tree Summation were also reproducible across different machines and compiler versions.[2]

[1]Or at least smaller than the machine epsilon for IEEE 754 double precision floating-point numbers.

[2]Linux 5.4.0-89-generic with GCC 9.4.0 on i10pc138, Linux 4.18.0-193.65.2.el8_2.x86_64 with GCC 11.2 on bwUniCluster 2.0

Table 4.2.: Difference between smallest and largest obtained sum from runs with varying PE-count.

Dataset	**ReproBLAS**	**Binary Tree Summation**	std::accumulate **+ AllReduce**
multi100	0.0	0.0	$3.6 * 10^{-12}$
rokasD1	0.0	0.0	$4.5 * 10^{-6}$
rokasA8	0.0	0.0	$4.5 * 10^{-8}$
fusob	0.0	0.0	$4.5 * 10^{-11}$
PeteD8	0.0	0.0	$6.1 * 10^{-6}$
rokasD4	0.0	0.0	$1.9 * 10^{-7}$
rokasA8	0.0	0.0	$2.8 * 10^{-6}$
354	0.0	0.0	$6.4 * 10^{-12}$
prim	0.0	0.0	$7.3 * 10^{-12}$

5. Conclusion

Binary Tree Reduction offers reproducible results independent of the core-count. It is not limited to floating-point operations and can be extended to the general set of reduction operations. Its message count per PE is bounded logarithmically by the number of elements per PE (Equation 2.5). If the Binary Tree Reduction takes up the majority of the runtime, optimizing the data distribution for the reduction can yield better results (Section 3.1.2).

For floating-point summation in particular, solutions like ReproBLAS [1] outperform Binary Tree Reduction and have a negligible runtime penalty compared to naive MPI_Allreduce implementations. The slowdown of Binary Tree Reduction compared to ReproBLAS is typically less than 2.

Future work could explore additional optimizations of the Binary Tree Reduction algorithm. Under the assumption of a specific data distribution, *rankFromIndex*-lookups could be replaced with a constant time algorithm. Furthermore, critical path analysis could provide better-performing calculation orders for outbound subtree roots. While Chapter 4 provides insight in the runtime of isolated reduction operations, additional examinations must be made to determine the runtimes of reductions which are a small part of larger workloads.

Bibliography

[1] Peter Ahrens, James Demmel, and Hong Diep Nguyen. "Algorithms for Efficient Reproducible Floating Point Summation". In: *ACM Transactions on Mathematical Software* 46.3 (Sept. 25, 2020), pp. 1–49. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3389360. URL: https://dl.acm.org/doi/10.1145/3389360.

[2] Diego Darriba, Tomáš Flouri, and Alexandros Stamatakis. "The State of Software for Evolutionary Biology". In: *Molecular Biology and Evolution* 35.5 (May 1, 2018), pp. 1037–1046. ISSN: 0737-4038. DOI: 10.1093/molbev/msy014. URL: https://doi.org/10.1093/molbev/msy014.

[3] Kai Diethelm. "The Limits of Reproducibility in Numerical Simulation". In: *Computing in Science Engineering* 14.1 (Jan. 2012). Conference Name: Computing in Science Engineering, pp. 64–72. ISSN: 1558-366X. DOI: 10.1109/MCSE.2011.21.

[4] Romain Dolbeau. "Theoretical peak FLOPS per instruction set: a tutorial". In: *The Journal of Supercomputing* 74.3 (Mar. 2018), pp. 1341–1377. ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-017-2177-5. URL: http://link.springer.com/10.1007/s11227-017-2177-5.

[5] David Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM Computing Surveys* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/103162.103163. URL: https://dl.acm.org/doi/10.1145/103162.103163.

[6] *IEEE Standard for Floating-Point Arithmetic.* ISBN: 9780738157528. IEEE. DOI: 10.1109/IEEESTD.2008.4610935. URL: http://ieeexplore.ieee.org/document/4610935/ (visited on 10/01/2021).

[7] Alexey M Kozlov et al. "RAxML-NG: a fast, scalable and user-friendly tool for maximum likelihood phylogenetic inference". In: *Bioinformatics* 35.21 (Nov. 1, 2019), pp. 4453–4455. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btz305. URL: https://doi.org/10.1093/bioinformatics/btz305.

[8] S. Roch. "A short proof that phylogenetic tree reconstruction by maximum likelihood is hard". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 3.1 (Jan. 2006). Conference Name: IEEE/ACM Transactions on Computational Biology and Bioinformatics, pp. 92–94. ISSN: 1557-9964. DOI: 10.1109/TCBB.2006.4.

[9] Alexandros Stamatakis and Alexey M Kozlov. "Efficient maximum likelihood tree building methods". In: *Phylogenetics in the Genomic Era.* 2020, 1.2:1–1.2:18.

[10] Oreste Villa et al. “Effects of floating-point non-associativity on numerical computations on massively multithreaded systems”. In: *Proceedings of Cray User Group Meeting (CUG)*. Vol. 3. 2009. URL: http://www.sci.utah.edu/~beiwang/teaching/cs6210-fall-2016/nonassociativity.pdf.

[11] Matthias Wiesenberger et al. “Reproducibility, accuracy and performance of the Feltor code and library on parallel computer architectures”. In: *Computer Physics Communications* 238 (May 2019), pp. 145–156. ISSN: 00104655. DOI: 10.1016/j.cpc.2018.12.006. arXiv: 1807.01971. URL: http://arxiv.org/abs/1807.01971.

A. Acknowledgements

This work was performed on the computational resource bwUniCluster funded by the Ministry of Science, Research and the Arts Baden-Württemberg and the Universities of the State of Baden-Württemberg, Germany, within the framework program bwHPC. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 882500).

Acronyms

AVX	Advanced Vector Extensions. 19
FLOPS	Floating-Point Operations per Second. 19
MPI	Message Passing Interface. 2, 6, 7, 11
PE	Processing Element. vii, ix, 1, 2, 6–17, 20–25, 37, 39
PSLLH	Per-Site Log-Likelihood. 1, 21
SIMD	Single Instruction Multiple Data. 19

B. Appendix

B.1. Buffer Subtree-Flushing Criterion Comparison

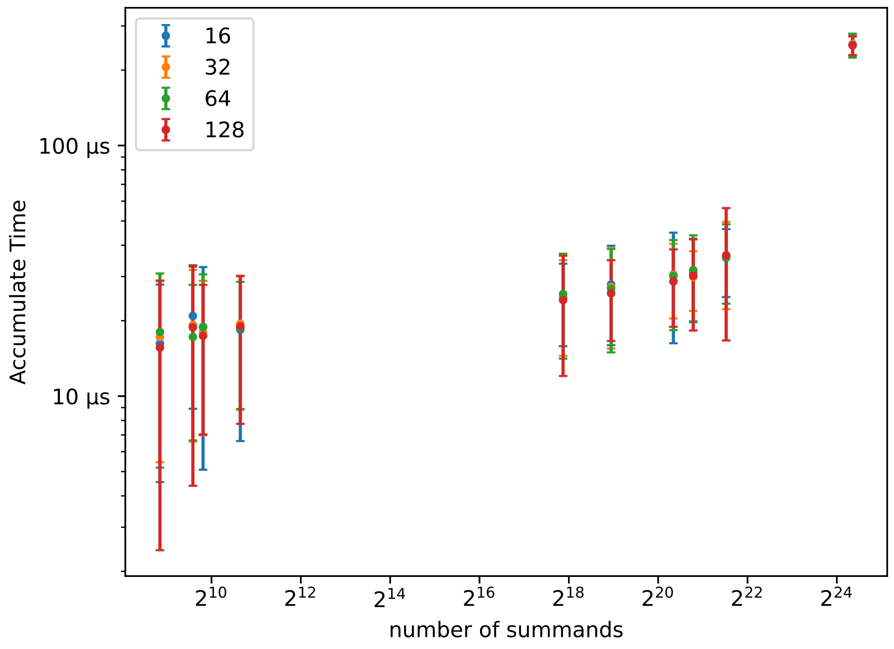
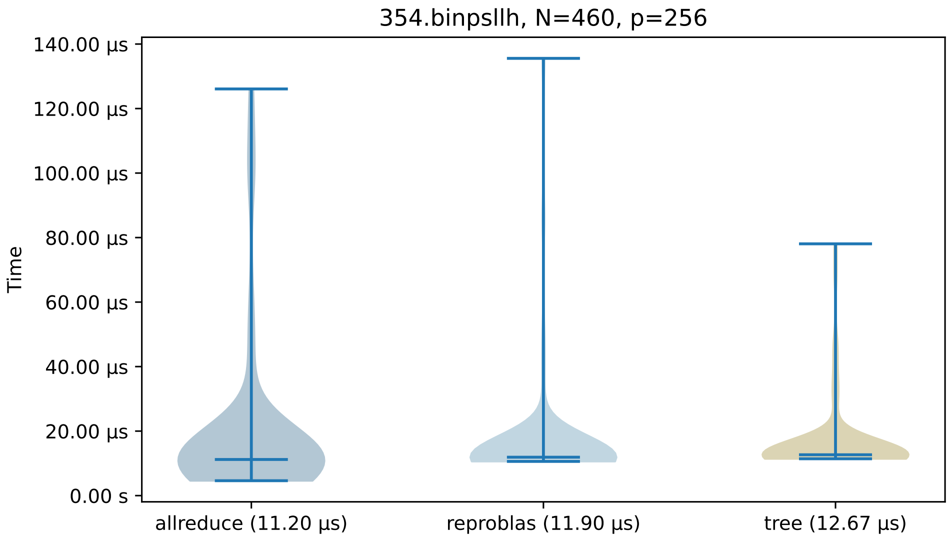
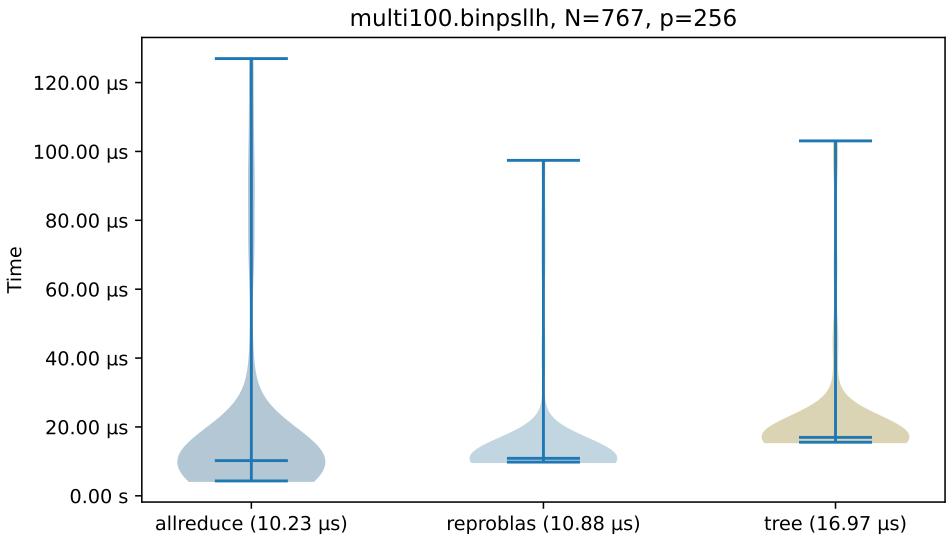
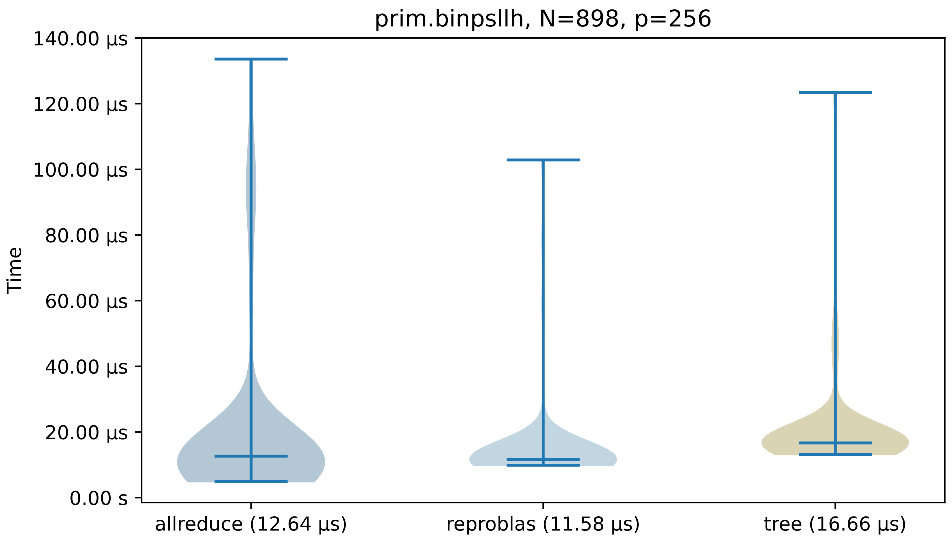


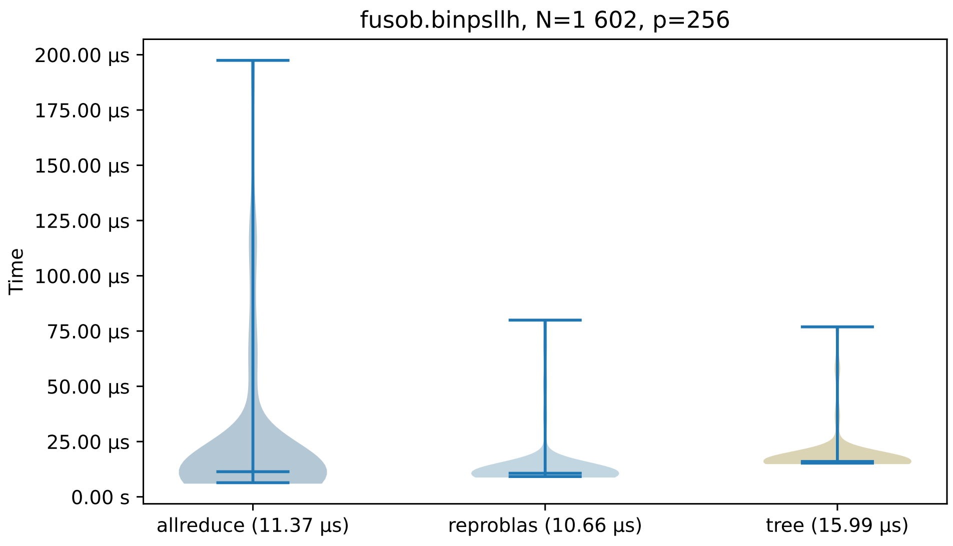
Figure B.1.: Accumulation runtime with different subtree sizes used as flushing criterion.

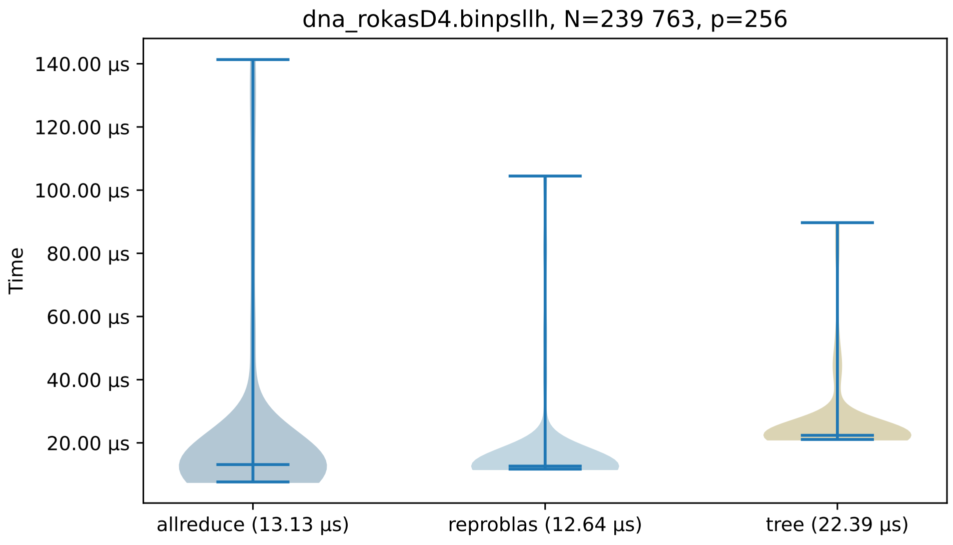
B.2. Detailed Benchmark Results

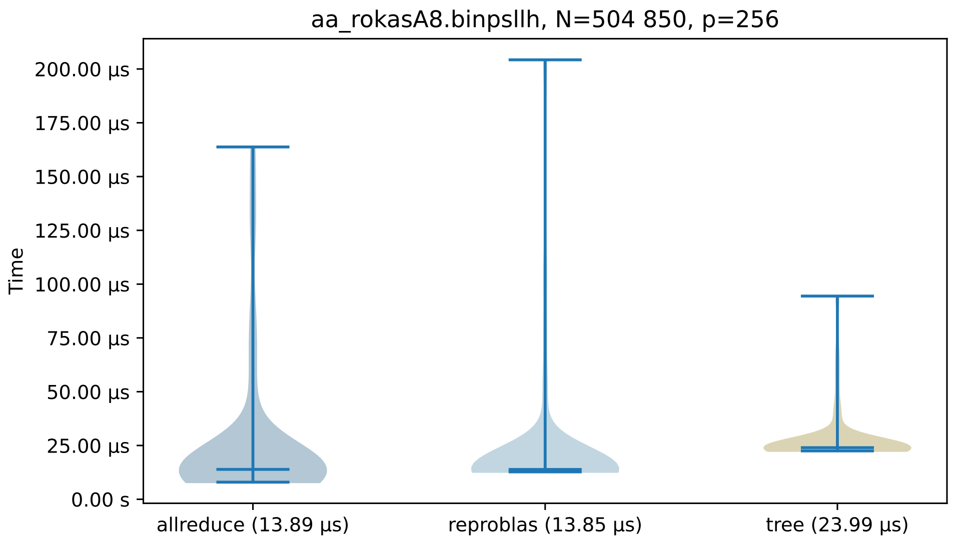

354.binpsllh, N=460, p=256
Time
140.00 µs
120.00 µs
100.00 µs
80.00 µs
60.00 µs
40.00 µs
20.00 µs
0.00 s
allreduce (11.20 µs)
reproblas (11.90 µs)
tree (12.67 µs)

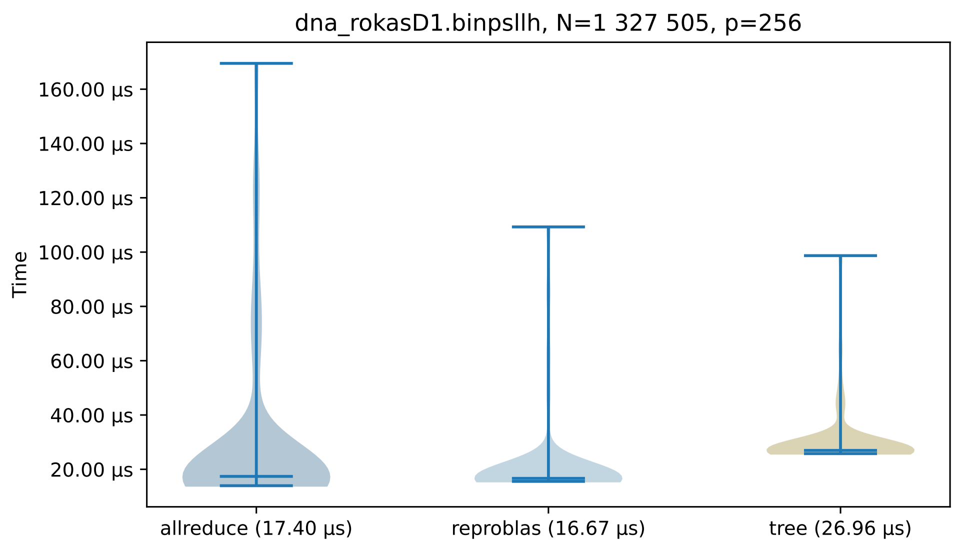

multi100.binpsllh, N=767, p=256
Time
120.00 µs
100.00 µs
80.00 µs
60.00 µs
40.00 µs
20.00 µs
0.00 s
allreduce (10.23 µs)
reproblas (10.88 µs)
tree (16.97 µs)

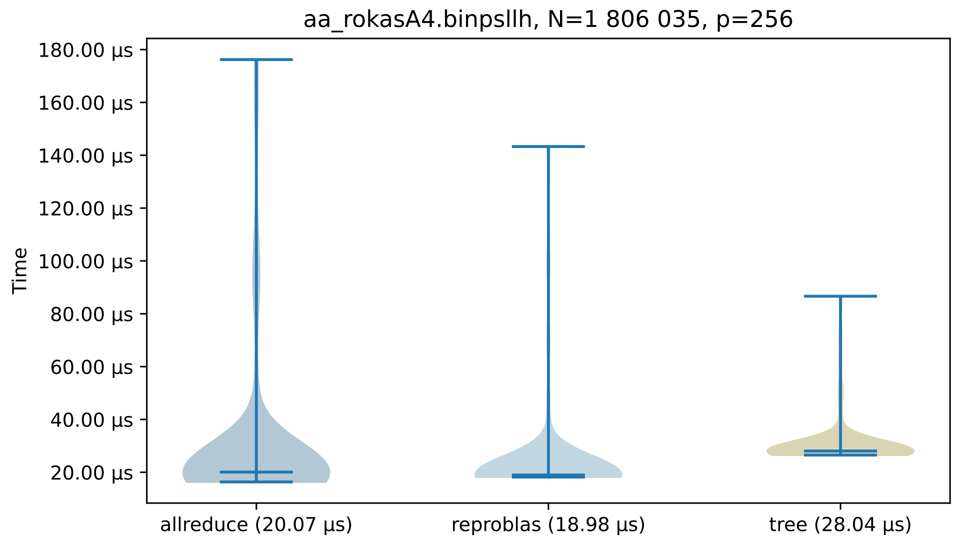

prim.binpsllh, N=898, p=256
Time
140.00 µs
120.00 µs
100.00 µs
80.00 µs
60.00 µs
40.00 µs
20.00 µs
0.00 s
allreduce (12.64 µs)
reproblas (11.58 µs)
tree (16.66 µs)

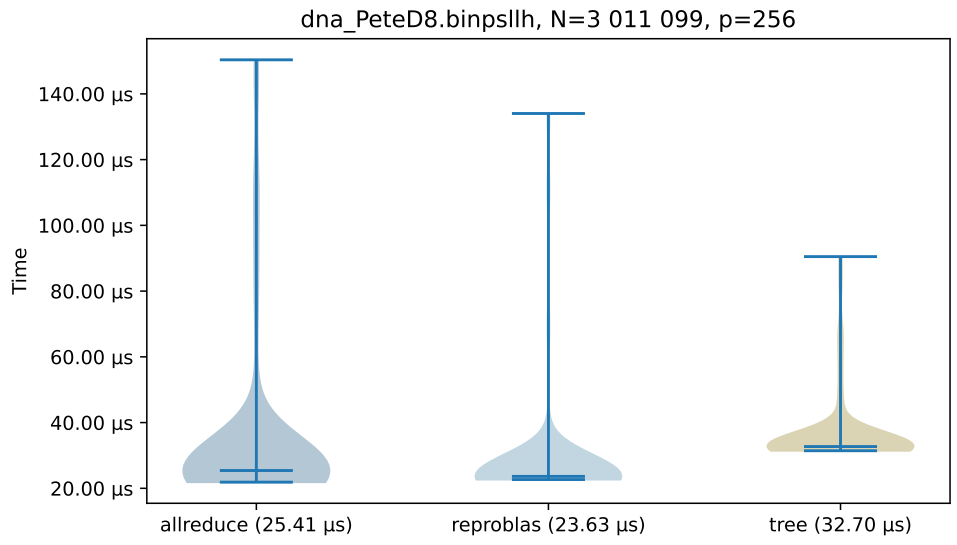

fusob.binpsllh, N=1 602, p=256
Time
200.00 µs
175.00 µs
150.00 µs
125.00 µs
100.00 µs
75.00 µs
50.00 µs
25.00 µs
0.00 s
allreduce (11.37 µs)
reproblas (10.66 µs)
tree (15.99 µs)


dna_rokasD4.binpsllh, N=239 763, p=256
Time
140.00 µs
120.00 µs
100.00 µs
80.00 µs
60.00 µs
40.00 µs
20.00 µs
allreduce (13.13 µs)
reproblas (12.64 µs)
tree (22.39 µs)


aa_rokasA8.binpsllh, N=504 850, p=256
Time
200.00 µs
175.00 µs
150.00 µs
125.00 µs
100.00 µs
75.00 µs
50.00 µs
25.00 µs
0.00 s
allreduce (13.89 µs)
reproblas (13.85 µs)
tree (23.99 µs)


dna_rokasD1.binpsllh, N=1 327 505, p=256
Time
160.00 µs
140.00 µs
120.00 µs
100.00 µs
80.00 µs
60.00 µs
40.00 µs
20.00 µs
allreduce (17.40 µs)
reproblas (16.67 µs)
tree (26.96 µs)


aa_rokasA4.binpsllh, N=1 806 035, p=256
Time
180.00 µs
160.00 µs
140.00 µs
120.00 µs
100.00 µs
80.00 µs
60.00 µs
40.00 µs
20.00 µs
allreduce (20.07 µs)
reproblas (18.98 µs)
tree (28.04 µs)


dna_PeteD8.binpsllh, N=3 011 099, p=256
Time
140.00 µs
120.00 µs
100.00 µs
80.00 µs
60.00 µs
40.00 µs
20.00 µs
allreduce (25.41 µs)
reproblas (23.63 µs)
tree (32.70 µs)

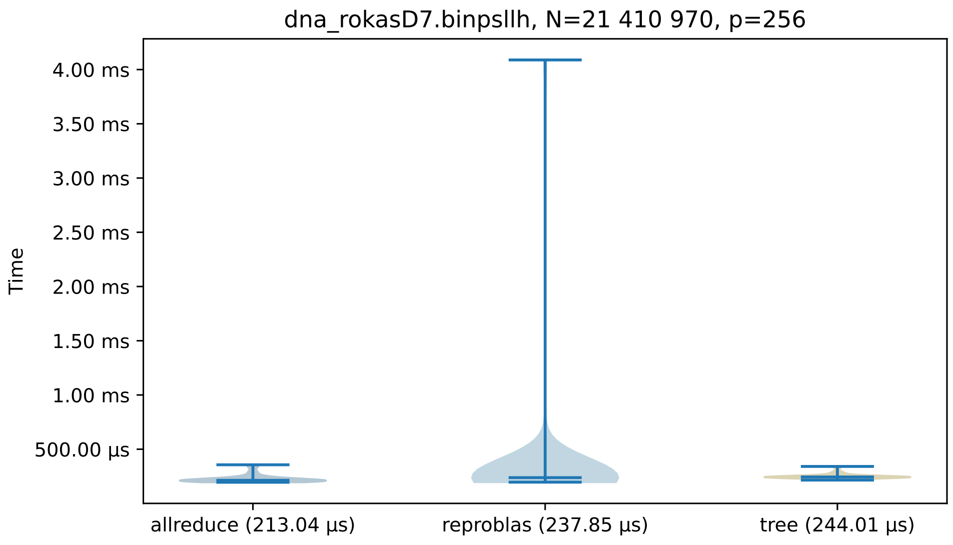


Figure B.2.: Runtime distribution for all datasets with $p = 256$ PEs.

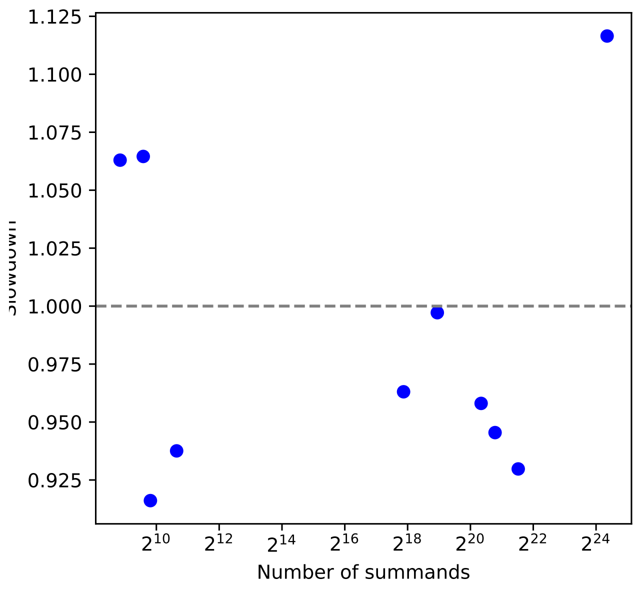


Figure B.3.: Slowdown of ReproBLAS compared to Allreduce ($p = 256$).

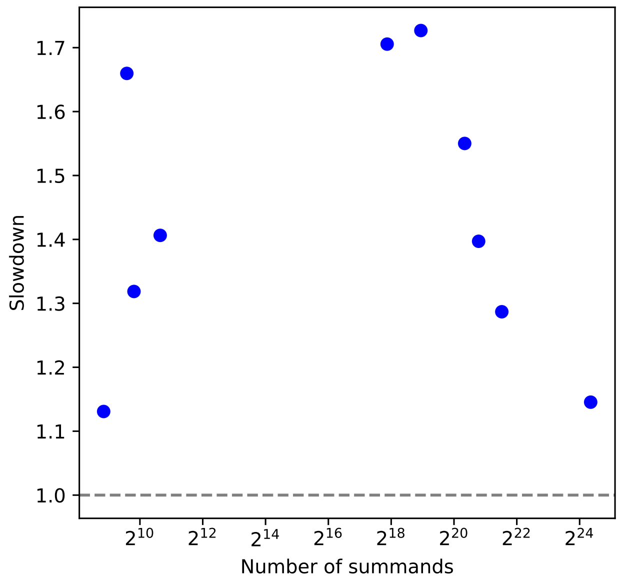
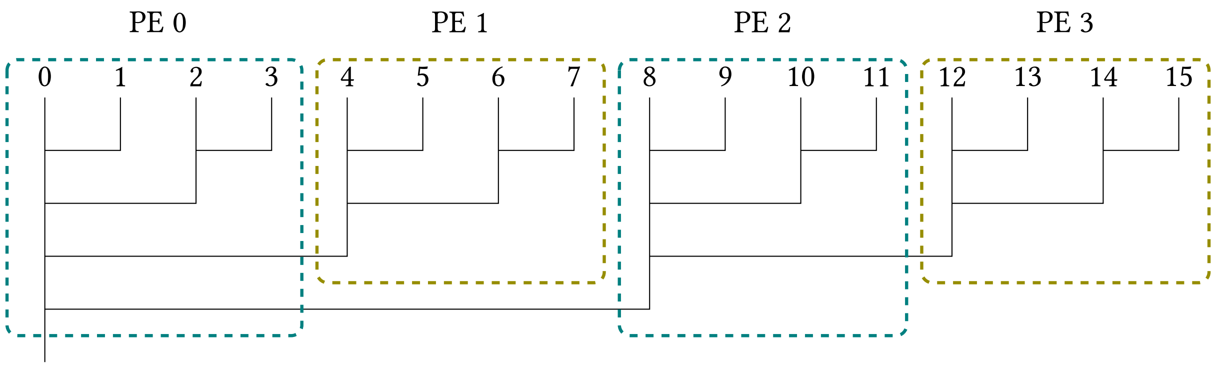
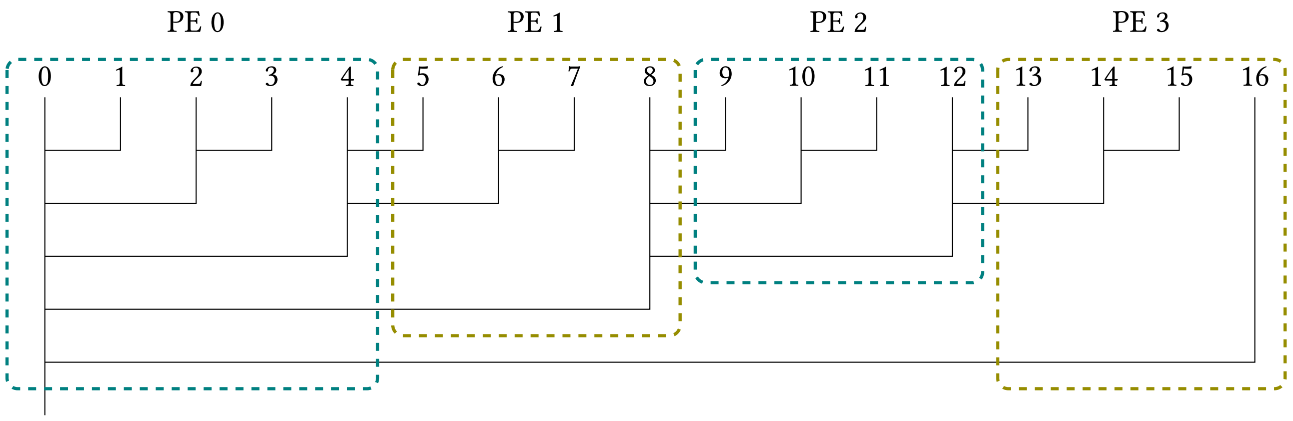


Figure B.4.: Slowdown of Binary Tree Summation compared to Allreduce ($p = 256$).

B.3. Message Count Maxima



(a) Optimal distribution of $N = 2^2 * p$ elements over $p = 4$ PEs.



(b) Worst-case distribution of $N = 2^2 * p + 1$ elements over $p = 4$ PEs.

Figure B.5.: Illustration of the shifting behaviour in the n_i^{lower}-distribution that produces maximal message counts.

B.4. Source Code

The complete source code, a database of benchmarks and exploratory Jupyter notebooks can be found inside the Git repository hosted at https://github.com/stelzch/allreduce. The LaTeX source code for this document with accompanying scripts to render the figures can be found at https://github.com/stelzch/bachelor-thesis.