Balancing Expression Dags for More Efficient Lazy Adaptive Evaluation

Martin Wilhelm[(✉)]

Institut für Simulation und Graphik, Otto-von-Guericke-Universität Magdeburg,
Universitätsplatz 2, 39106 Magdeburg, Germany
`martin.wilhelm@ovgu.de`

Abstract. Arithmetic expression dags are widely applied in robust geometric computing. In this paper we restructure expression dags by balancing consecutive additions or multiplications. We predict an asymptotic improvement in running time and experimentally confirm the theoretical results. Finally, we discuss some pitfalls of the approach resulting from changes in evaluation order.

1 Introduction

Most theoretical algorithms in the field of computational geometry are based on the real RAM model of computation and therefore assume exact real number arithmetic at unit cost. Actual processors cannot represent real numbers and instead use floating-point arithmetic. Computing exact numerical values is expensive and not always possible. Luckily, it is also seldom necessary to compute exact values in order to ensure robustness in geometric algorithms. The Exact Geometric Computation paradigm instead only demands that the decisions made in a geometrical algorithm are correct [9,10]. A common technique used for exact-decisions computation is to store the computation history in an arithmetic expression dag and then adaptively (re)compute the result with a higher precision until a verified decision can be made. Several expression-dag-based number types have been developed with different evaluation strategies. Strategies can be to gradually increase the precision bottom-up (`LEA` [1]) or fall back to exact computation (`CGAL::Lazy_exact_nt` [8]) if a decision cannot be verified, or to use a precision-driven[1] evaluation (`leda::real` [2], `Core::Expr` [5,11], `Real_algebraic` [6]). All of the mentioned number types suffer from high performance overhead compared to standard floating-point arithmetic.

In this work we make an attempt to improve the performance of dag-based number types by restructuring the underlying expression dag in certain situations. Restructuring the expression dag was originally proposed by Yap [10]. To our knowledge, there is no previous work that actually implements any restructuring strategy. We focus on reducing the depth of an expression dag, i.e. the size

---

[1] This should more correctly be called "accuracy-driven", but we use the term "precision-driven" throughout this paper for historical reasons.

of the longest path from any node to the root. The accuracy needed at a node in an expression dag to guarantee a certain error at its root generally increases with the size of the longest path between the node and the root. Therefore a decrease in expression depth can be expected to lead to better error bounds at lower precision and, consequently, to a better performance of dag-based number types. We restructure the expression dag by "balancing" consecutive additions and consecutive multiplications, such that the maximum depth of the involved operators is minimized. By this we also increase the independence of the nodes, which makes it more feasible to parallelize the evaluation. In this work, however, we will not elaborate on advantages regarding parallelization.

We provide a theoretical analysis and evaluate our strategy based on the number type `Real_algebraic` introduced by Mörig et al. [6].

2   Theoretical Foundation

An *expression dag* is a rooted ordered directed acyclic graph, which is either

1. A single node containing a number or
2. A node representing a unary operation $\{\sqrt{}, -\}$ with one, or a binary operation $\{+, -, *, /\}$ with two, not necessarily disjoint, expression dags as children.

We call an expression dag E' whose root is part of another expression dag E a *subexpression of E*.

Let E be an expression dag, let $\circ \in \{+, *\}$ and let E' be a subexpression of E with root r of type $\circ$. Let T be a connected subgraph of E', containing r, such that all nodes in $T - r$ have at most one predecessor in E and are of type $\circ$. Then T is a tree and we call T an *operator tree*. The children of the leaves of T in E are called *operands* of T. We restructure E by replacing all maximal operator trees in E by a balanced operator tree with the same number of operands. For a single tree, we call this replacement *balancing the operator tree*. If all maximal operator trees in E are replaced, we call the process *balancing the expression dag*.

We determine the asymptotic running time of a single precision-driven evaluation before and after balancing the expression dag for a series of additions or a series of multiplications. Assumptions on the unit costs for the arithmetic operations and the increase in accuracy are consistent with `leda::real`, `Real_algebraic` and partly with `Core::Expr`.

2.1   Addition

Assume we have a dag-based number type that determines the result of an addition $z = x + y$ with absolute accuracy q in time $\Theta(q + \log |z|)$ if x and y are accurate up to $q + c$ fractional digits, where c is some constant.

Let $x_1, ..., x_n$ be distinct floating point numbers with exponent $\leq e, e \geq 0$. We want to determine the running time to compute $z = \sum_{i=1}^{n} x_i$ with absolute accuracy q. Any expression dag for z contains an operator tree consisting of all addition nodes. Assume that the operator tree is a linear list, i.e. the computation

order is equivalent to $x_1 + (x_2 + (x_3 + ... + (x_{n-1} + x_n)))$. Then the i-th addition (counting from the root) must be accurate up to $q_i = q + ic$ fractional digits and the magnitude of its result is at most $e_i = e + \lceil \log(n - i) \rceil$. Therefore we get the time for computing z by adding the time needed on each level as

$$T_{\text{list}}(q) = O\left(\sum_{i=0}^{n-1}(q_i + e_i)\right) = O\left(\sum_{i=0}^{n-1}(q + ic + e + \log(n - i))\right)$$
$$= O\left(nq + n^2 + ne\right)$$

This bound is tight if all summands have maximum exponent. Now assume the operator tree is perfectly balanced, i.e. the computation order is equivalent to

$$((((x_1 + x_2) + (x_3 + x_4)) + ...) + (... + ((x_{n-3} + x_{n-2}) + (x_{n-1} + x_n))))$$

Then at level i there are 2^i additions, which must be accurate up to $q_i = q + ic$ fractional digits. The magnitude of their result is at most $e_i = e + \log n - i$. So the asymptotic bound for the computation time shrinks to

$$T_{\text{bal}}(q) = O\left(\sum_{i=0}^{\log n} 2^i(q + ic + e + \log n - i)\right) = O\left(nq + n\log n + ne\right)$$

2.2 Multiplication

For multiplication we assume the number type computes the result of $z = x * y$ with absolute accuracy q in time $\Theta((q + \log|z|)^{\log 3})$ if x is accurate up to $q + c + \lceil \log|y| \rceil$ and y up to $q + c + \lceil \log|x| \rceil$ fractional digits, where c is some constant. We determine the running time to compute $z = \prod_{i=1}^{n} x_i$ with absolute accuracy q.

We consider the operator tree consisting of all multiplication nodes in an expression dag for z. Let $e \geq 0$ be the maximum exponent of $x_1, ..., x_n$. In the unbalanced case the accuracy needed increases by at most $c + e$ with each level top-down, whereas the maximum exponent of the result increases by e bottom-up. Assuming that $x_1, ..., x_n$ are exact, we do not need to increase the accuracy of the leaves. Then we get

$$T_{\text{list}}(q) = O\left(\sum_{i=0}^{n-1}(q_i + e_i)^{\log 3}\right) = O\left(\sum_{i=0}^{n-1}(q + i(c + e) + (n - i)e)^{\log 3}\right)$$
$$= O\left(nq^{\log 3} + n^{\log 3 + 1} + n^{\log 3 + 1}e^{\log 3}\right)$$

This bound is tight if $x_1, ..., x_n$ all have exponent e. When the operator tree is balanced, the accuracy needed increases by $c + e_{i+1}$ at level i, where $e_i = 2^{\log n - i}e$, so the requested accuracy at level i is

$$q_i = q + ic + \sum_{j=0}^{i+1} 2^{\log n - j}e \leq q + ic + 2^{\log n + 1}e$$

Therefore

$$T_{\mathrm{bal}}(q) = O\left(\sum_{i=0}^{\log n} 2^i (q + ic + 2^{\log n+1} e + 2^{\log n-i} e)^{\log 3}\right)$$
$$= O\left(nq^{\log 3} + n(\log n)^{\log 3} + n^{\log 3+1} e^{\log 3}\right)$$

If $e > 0$ the improvement we get from balancing the tree is dominated by the cost for managing the increasing number of integer digits. If one can expect the exponent to be bounded from above, the improvement gets asymptotically significant. Let $e_{\max}$ be the largest exponent occurring during the whole computation. Then

$$T_{\mathrm{list}}(q) = O\left(\sum_{i=0}^{n-1} (q + i(c + e_{\max}) + e_{\max})^{\log 3}\right)$$
$$= O\left(nq^{\log 3} + n^{\log 3+1} + n^{\log 3+1} e_{\max}^{\log 3}\right)$$

whereas

$$T_{\mathrm{bal}}(q) = O\left(\sum_{i=0}^{\log n} 2^i (q + i(c + e_{\max}) + e_{\max})^{\log 3}\right)$$
$$= O\left(nq^{\log 3} + n(\log n)^{\log 3} + n(\log n)^{\log 3} e_{\max}^{\log 3}\right)$$

The asymptotic bound for $T_{\mathrm{list}}(q)$ is tight if the values of $\Theta(n)$ inner nodes are of order $e_{\max}$.

3   Implementation

The balancing strategy has been implemented for the dag-based exact-decision numbertype `Real_algebraic` designed by Mörig et al. [6]. This number type consists of a single- or multi-layer floating-point-filter [4], which falls back to adaptive evaluation with bigfloats stored in an expression dag [3]. Balancing is done at most once at each node, right before the first bigfloat evaluation. Otherwise, existing results would have to be recomputed after changing the structure of the dag, which could potentially lead to a massive overhead if subexpressions need to be evaluated during dag construction. Once evaluated nodes are therefore treated as operands in any subsequent balancing process.

We call evaluations of subexpressions of an expression dag E *partial evaluations of E*. By preventing evaluated nodes to be part of another operator tree, frequent partial evaluations during construction can fully negate the benefits of the balancing strategy. If partial evaluations occur only sporadically, then their impact on the *expression depth of E*, i.e. the maximum distance of any node to its root, is small, since each of the involved subexpressions have been balanced when they were evaluated for the first time.

Observation 1. *Let E be an expression dag consisting of n additions or n multiplications and n + 1 bigfloats. Let d be the expression depth of E after its evaluation. If at most k partial evaluations of E occur before the evaluation of E, then $d \leq k \lceil \log \frac{n}{k} \rceil$.*

On the first evaluation of a node that cannot be handled by the floating-point-filter, the balancing process starts at this node recursively. If the current node contains an addition or a multiplication and has not be balanced before, all operands of the maximal operator tree containing this node as root will be retrieved. If the depth of the operator tree can be reduced, it gets balanced (cf. Algorithm 1). For almost-balanced trees a slight decrease in depth may not justify restructuring a large tree. Therefore it might be useful to experimentally decide on a factor tightening this condition in later implementations.

Algorithm 1. The relevant operator trees are retrieved in the form of an operand list with an associated depth. By comparing the depth with the number of operands it gets decided whether the trees should be balanced.

Data: current node *node*
if *node is not balanced* **then**
    if *node is addition or multiplication* **then**
        $(operands, depth) = retrieve\_operands(node)$
        if $depth > \lceil \log |operands| \rceil$ **then**
          | balance current operation
        end
        foreach $op \in operands$ **do**
          | recurse on *op*
        end
    else
      | recurse on children
    end
    mark *node* as balanced
end

The operands get retrieved through a depth-first search. Nodes can be retrieved more than once and therefore the same node can represent multiple operands. A node is treated as an operand if one of the following conditions holds:

1. The node is not of the same type as the current operation (i.e. + or ∗).
2. The node has already been balanced (and therefore initialized).
3. The node has more than one parent.

The third condition is necessary, since the subexpression represented by this node will be destroyed during balancing. If a full copy of the node would be created instead, this may lead to an exponential blowup for highly self-referential structures (cf. Fig. 1).

A similar observation as for the second condition (cf. Observation 1) can be made. If few operator nodes have more than one parent, the overall impact on the expression depth is small.
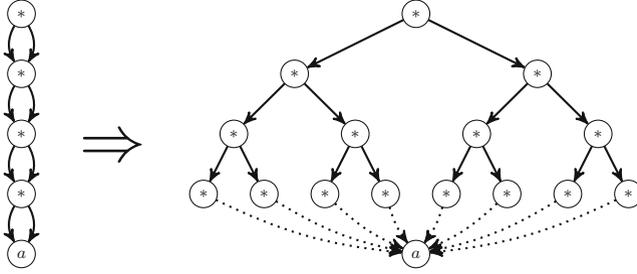
Fig. 1. An expression dag computing a^{16} and its exponential expansion that results from resolving multiple references through copying.

Observation 2. *Let E be an expression dag consisting of n additions or n multiplications and $n + 1$ bigfloats. Let d be the expression depth of E after its evaluation. If at most k operator nodes of E have more than one reference, then $d \leq k\lceil \log \frac{n}{k} \rceil$.*

We balance an operation by combining two operands to a new operand until only one node is left by treating the operand vector like a queue (cf. Algorithm 2). Note that this strategy does not preserve the evaluation order of the operands if the number of operands is not a power of two. This can have consequences for the running time and may obfuscate the experiments. If operand order is of importance, it can be preserved by inserting dummy nodes with values 0 for addition and 1 for multiplication up to the next power of two.

Algorithm 2. The operator tree is restructured by discarding all nodes except the root and building a new balanced operator tree bottom-up by repeatedly combining the two smallest subtrees to a new tree.

Data: operand vector *operands*, operation type $\circ$, root node *root*
$size = operands.size()$;
for $i = 0$ **to** $size - 2$ **do**
  | $operands.add(new\ Node(operands[2i], operands[2i + 1], \circ))$
end
$root.left = operands[2 * size - 2]$;
$root.right = operands[2 * size - 1]$;

---

4   Experiments

All experiments are run on an Intel Core i5 660 with 8 GB RAM under Ubuntu 16.04 LTS. We use Boost interval arithmetic as floating-point-filter and MPFR bigfloats for the bigfloat arithmetic. The code is compiled using g++ 5.4.0 with C++11 on optimization level `O3` and linked against Boost 1.62.0 and MPFR 3.1.0. Test results are averaged over 25 runs each if not stated otherwise. The variance for each data point is negligible.

We will perform two simple experiments to evaluate our strategy. In our first experiment we compute the sum of the square roots of the natural numbers 1 to n with accuracy q.

```
template <class NT> void sum_of_sqrts(const int n, const long q){
  NT sum = NT(0);
  for (int i = 1; i <= n; ++i) {
    sum += sqrt(NT(i));
  }
  sum.guarantee_absolute_error_two_to(q);
}
```

The second test computes the generalized binomial coefficient

$$\binom{\sqrt{13}}{n} = \frac{\sqrt{13}(\sqrt{13}-1)\cdots(\sqrt{13}-n+1)}{n(n-1)\cdots 1}$$

with accuracy q.

```
template <class NT> void bin_coeff(const int n, const long q){
  NT b = sqrt(NT(13));
  NT num = NT(1); NT denom = NT(1);
  for (int i = 0; i < n; ++i) {
    num *= b - NT(i);
    denom *= NT(i+1);
  }
  NT bc = num/denom;
  bc.guarantee_absolute_error_two_to(q);
}
```

For each test we compare four different implementations. We distinguish between no balancing (def), balancing only addition (add), balancing only multiplication (mul) and balancing both addition and multiplication (all).

The sum-of-square-roots test as well as the binomial coefficient test provide simple examples for when balancing can be of use (cf. Fig. 2). Obviously balancing multiplication does not have a positive effect on the sum-of-square-roots test
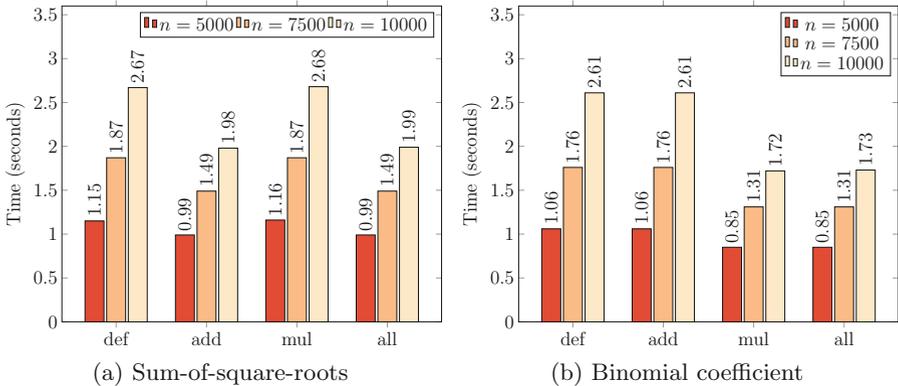

(a) Sum-of-square-roots   (b) Binomial coefficient

Fig. 2. Performance gain through balancing for `sum_of_sqrts` and `bin_coeff` with a requested accuracy of $q = 50000$ for different values of n.

and balancing addition does not have a positive effect on the binomial coefficient computation. There is a small overhead in these cases due to the traversal of the dag. The overhead vanishes in `all`, since the same procedure is used for both addition and multiplication.
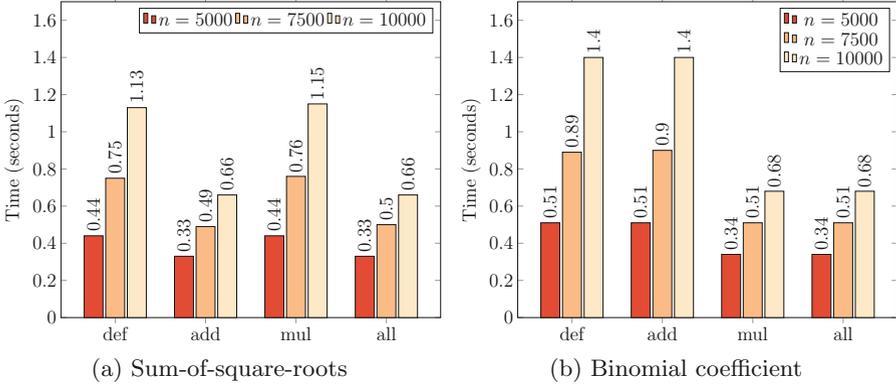

Fig. 3. Performance gain through balancing for `sum_of_sqrts` and `bin_coeff` with a requested accuracy of $q = 25000$. The relative gain is larger than for $q = 50000$ (cf. Fig. 2).

The relative benefit of balancing increases if the precision increase due to the number of operands is large relative to the requested accuracy for the result. Figure 3 shows the performance gain through balancing for a requested accuracy of $q = 25000$. With 10000 operands, the relative gain is about 42% for `sum_of_sqrts` and 51% for `bin_coeff` compared to 26% and 34% for $q = 50000$. The theoretical analysis from Sect. 2 predicts that the absolute performance gain

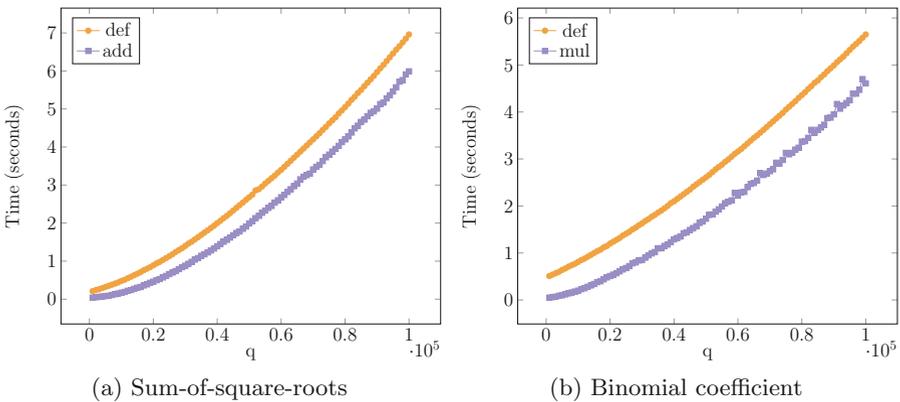

Fig. 4. Absolute performance gain through balancing for `sum_of_sqrts` and `bin_coeff` for $n = 10000$ with different requested accuracies (average over five runs). The absolute gain is almost independent of q. The relative gain decreases.

primarily depends on the number of the operands that can be balanced and is independent from the requested accuracy. The experimental results largely confirm this assumption as shown in Fig. 4. Since balancing is done before the first evaluation, the overhead due to the balancing procedure only depends on the size of the expression dag and the number of operands.

5   Caveats

When restructuring an expression dag there are some potential pitfalls one should be aware of. Changing the structure of an expression dag leads to a change in evaluation order, which may in turn influence the performance. Other hurdles are even more subtle, since they result from implementation details of the underlying bigfloat arithmetic. We show examples, where this leads to problems for balancing. However, the caveats are not restricted to balancing, but apply to restructuring attempts in general.

5.1   Evaluation Order

When evaluating a dag-based number type recursively, a slight change in expression order can have an unexpectedly high impact on the evaluation time [7]. Balancing the dag may have a negative impact on the optimal expression order. One example where this may occur is the computation of the geometric sum $\sum_{i=0}^{n} r^i$ with $r < 1$.

```
template <class NT> void geometric_sum(const int n, const long q){
  NT r = sqrt(NT(13)/NT(64));
  NT ri = NT(1); NT s = ri;
  for (int i=0; i<n; ++i){
    ri *= r;
    s  += ri;
  }
  s.guarantee_absolute_error_two_to(q);
}
```

We call the multiplication node m_i resulting from the i-th multiplication *deeper* than the node m_j resulting from the j-th multiplication if $i < j$ and *shallower* if $j < i$. If m_j is shallower than m_i then m_j is an ancestor of m_i in the expression dag. When balancing the expression dag the accuracy needed at the deeper multiplication nodes decreases, while the accuracy needed at the shallower nodes increases. Since in geometric_sum the shallower multiplication nodes depend on the deeper ones, the balancing actually increases the final accuracy needed at the deeper multiplication nodes by an amount logarithmic in the total number of additions. To make things worse, the deeper nodes are still evaluated first (with low precision) and therefore need to be recursively re-evaluated for every shallower multiplication node, leading to a quadratic number of evaluations (Fig. 5).

Note, that this does not happen for the linear computation order if we assume the following increase in accuracy (cf. Sect. 2):
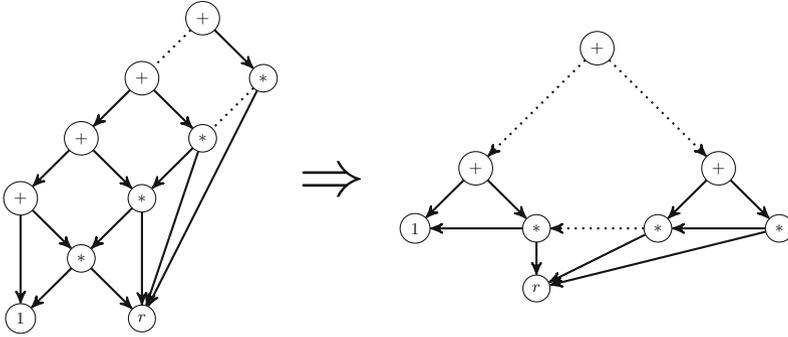
Fig. 5. Expression dags for `geometric_sum` before and after balancing. After balancing, all multiplication nodes are on the same level, with the deeper ones evaluated first, inducing a quadratic number of evaluation steps.

– To evaluate $z = x + y$ with accuracy q, both x and y must be accurate up to $q + 2$ digits.
– To evaluate $z = x * y$ with accuracy q, x must be accurate up to $q + 2 + \lceil \log |y| \rceil$ and y must be accurate up to $q + 2 + \lceil \log |x| \rceil$ digits.

Since for $r < 1$ also $r^i \leq 1$ the increase in accuracy is the same for addition and multiplication. Therefore with linear computation order the multiplication nodes do not need to be re-evaluated after their initial evaluation. If $r > 1$ the linear dag and the balanced dag show similar behavior.[2]

To avoid extensive recomputations, we can compute a topological order and determine the final accuracy needed at each node before recomputing it [7]. We implement this strategy and compare it with recursive evaluation. The standard recursive evaluation procedure essentially works as depicted in Algorithm 3. At each node the needed accuracy of its children is ensured and the value at this node gets recomputed. Nodes can get recomputed several times if they have more than one parent.

Algorithm 3. Evaluating an expression dag by recursively increasing the accuracy of the children before recomputing the current operation.

Data: requested accuracy q
if *error is larger than* 2^{-q} **then**
  compute needed accuracy for children
  recurse on children with their respective accuracy
  recompute
end

When evaluating topologically we determine a topological order for all inexact nodes and compute the maximum accuracy needed for those nodes.

---

[2] `Real_algebraic` usually overestimates the exponent by one, therefore in our tests r is chosen to be smaller than 0.5.

Afterwards we recompute the nodes with their maximum accuracy (cf. Algorithm 4). By following this procedure we can guarantee that no node is recomputed more than once during one evaluation of the expression dag.

Algorithm 4. Evaluating an expression dag by finding a topological order and determining the maximum accuracy needed at each node before recomputing them.

Data: requested accuracy q
if *error is larger than* 2^{-q} **then**
    $top =$ all inexact nodes in topological order
    for $i = 1$ **to** $|top|$ **do**
        | update the required error for the children of $top[i]$
    end
    for $i = |top|$ **downto** 1 **do**
        if $top[i].error > top[i].requested\_error$ **then**
            | recompute $top[i]$
        end
    end
end

We execute the geometric sum experiment with the four balancing strategies from before. Furthermore for each of these strategies we evaluate either recursively (r) or in topological order (t).

As the results in Fig. 6 show, balancing the expression dag destroys a favorable evaluation order when computing the geometric sum. Switching to a topological evaluation order negates this effect. Note that the performance loss due

Fig. 6. Balancing additions leads to a massive increase in running time for `geometric_sum` with $q = 50000$ by creating a bad evaluation order. Topological evaluation solves the problem.

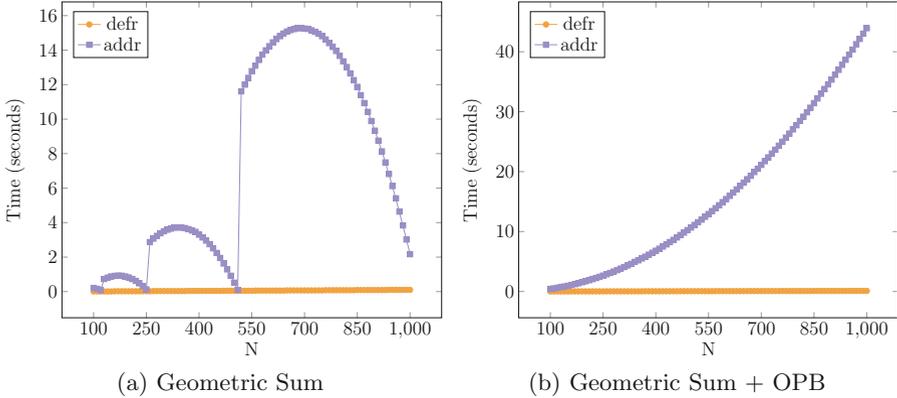(a) Geometric Sum   (b) Geometric Sum + OPB

Fig. 7. Comparison of the behavior of `geometric_sum` for increasing n with the balancing procedure from Algorithm 2 and with order preserving balancing (OPB). The original procedure leads to jumps in running time, while order preserving balancing produces the expected quadratic behavior ($q = 50000$, averaged over 5 runs each).

to the logarithmic increase of precision in the balanced case is too small to show in the measurements.

The values for n have been chosen to show spikes in the running time. As pointed out in Sect. 3 our balancing algorithm does not necessarily preserve the order of the operands. If the shallowest multiplication node is evaluated first, this leads to an optimal evaluation order. Figure 7 compares our implementation with an order-preserving balancing strategy.[3]

The algorithm we use to build a balanced tree results in large jumps when stepping from $2^k - 1$ to 2^k operands ($k \in \mathbb{N}$). With $2^k - 1$ operands the previously rightmost operand, i.e. the shallowest multiplication node, becomes the leftmost operand in the balanced tree and therefore the evaluation order is optimal. With 2^k operands the previous operand order is preserved by the algorithm and is therefore the worst possible. If preserving order is enforced, the quadratic increase in running time is evident.

5.2 Operands Matter

In some cases balancing can destroy a favorable dag structure independently from the evaluation order. We compute the telescoping product $\prod_{i=1}^{n-1} \frac{i+1}{i}$ through the following algorithm.

```cpp
template <class NT>
void telescoping_product(const int n, const long q){
  NT prod = NT(1);
  for (int i = 1; i < n; ++i) {
    prod *= NT(i+1)/NT(i);
  }
  prod.guarantee_absolute_error_two_to(q);
}
```

---

[3] This is implemented by inserting dummy nodes up to the next power of two.

In the experimental results shown in Fig. 8a, a performance decrease due to balancing is evident, which also cannot be corrected through a change in evaluation order. The reason for this effect is that the naive order enables the bigfloat arithmetic to make use of eliminating factors. Bigfloat multiplications involving integers[4] are less expensive[5]. In the original expression order the result of each multiplication is an integer and can be determined as such. Therefore, although significantly reducing the average need of accuracy, balancing has a negative effect on the performance.
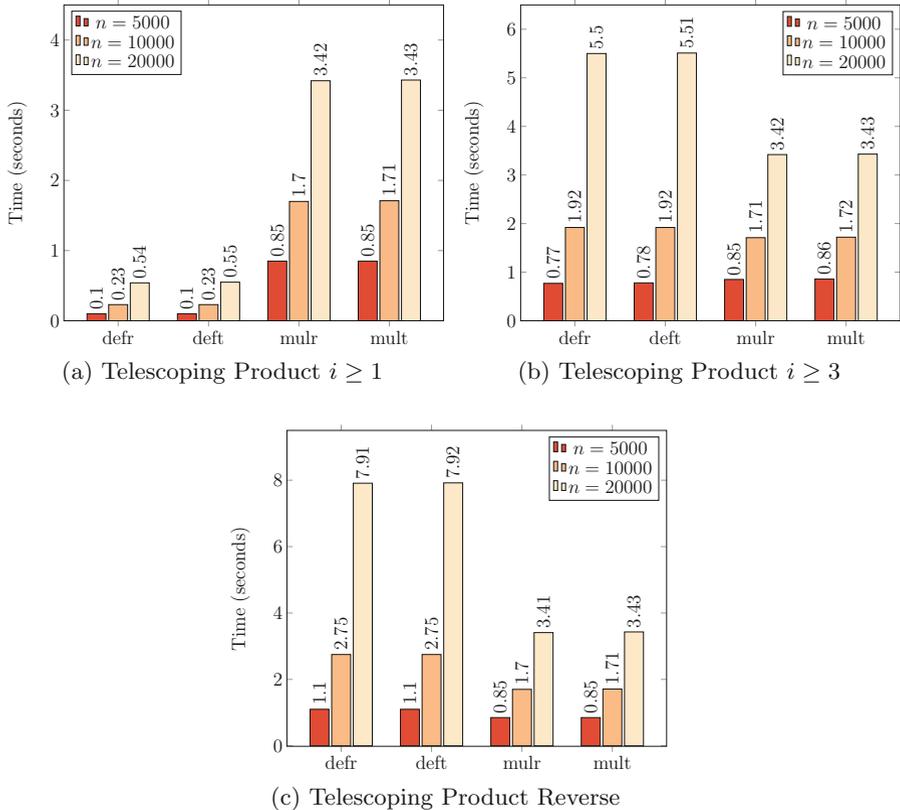

(a) Telescoping Product $i \geq 1$ (b) Telescoping Product $i \geq 3$

(c) Telescoping Product Reverse

Fig. 8. Performance of different variants of `telescoping_product` before and after balancing multiplications ($q = 50000$). In the original version balancing destroys a favorable order of the operands, which cannot be corrected for by switching to topological evaluation. When starting with $i = 3$, the order before balancing is less favorable and the performance gain of balancing outweighs the loss for larger n. For `telescoping_product_reverse` no favorable order is destroyed and balancing shows the expected net benefit.

---

[4] Or integers divided by a power of two.
[5] This behavior was confirmed with both `mpfr` and `leda` bigfloats.

If the product is computed starting with $i = 3$ only every third subexpression evaluates to an integer. While there are still some favorable structures getting disrupted by balancing the expression dag, the benefit of balancing surpasses the loss as the number of operands increases (cf. Fig. 8b). The effect vanishes if the product is computed in reverse order as depicted in the following algorithm.

```
template <class NT>
void telescoping_product_reverse(const int n, const long q){
  NT prod = NT(1);
  for (int i = n-1; i >= 1; --i) {
    prod *= NT(i+1)/NT(i);
  }
  prod.guarantee_absolute_error_two_to(q);
}
```

By this, none of the subexpressions involved evaluates to an integer and only a logarithmic amount of subexpressions evaluates to an integer divided by a power of two. The results of the experiment for the reverse case are shown in Fig. 8c. Now balancing has the expected positive effect on the overall performance. Note that, as expected, the forward loop starting with $i = 3$ without balancing takes approximately two third of the time of the reverse case.

5.3   Overhead

In all tests, except the telescoping product test with $i \geq 1$, the overhead for the balancing procedure as well as for the topological sorting was (usually much) less than 0.5% of the final running time. The running time of `telescoping_product` is unusually small compared to its number of operations, therefore the relative overhead of additional computations is higher. In this case the overhead amounts to less than 2% for balancing and less than 3% for topological sorting.

6   Conclusion

Balancing additions and multiplications in an expression dag can significantly reduce the computation time needed as demonstrated by the sum-of-square-roots test and the binomial coefficient test. The experimental data indicates that the overhead due to the balancing algorithm is small compared to the cost of the bigfloat operations. Balancing may cause changes in the evaluation order that lead to increased running time. Those issues can partially be addressed by switching to a topological evaluation, which can be done with small overhead as well.

We conclude that it is useful to provide a number type supporting balancing of expression dags in combination with topological evaluation. The use of this number type should be considered whenever an algorithm performs a large number of consecutive additions or multiplications. Switching a number type is usually less time-consuming than a deep analysis and adjustment of the used algorithm.

7 Future Work

In this paper we restricted restructuring of the dag to balancing additions and multiplications. Performance increase due to further restructuring is imaginable. Subtractions could easily be included in the balancing process by treating them like an addition and a negation and propagating the negations to the operands. It may also be useful to incorporate divisions into the multiplication balancing process. Since inversions are much more expensive than negations, it seems not feasible to replace them by a multiplication and an inversion. Instead a promising strategy might be to reduce the number of divisions by raising them to the root.

Balancing an expression dag makes its nodes more independent and therefore makes it more accessible for parallelization. Further restructuring with the goal of faster parallelization, e.g. expanding products, might be profitable.

References

1. Benouamer, M.O., Jaillon, P., Michelucci, D., Moreau, J.: A lazy exact arithmetic. In: Proceedings of the 11th Symposium on Computer Arithmetic, pp. 242–249 (1993)
2. Burnikel, C., Mehlhorn, K., Schirra, S.: The leda class real number (1996)
3. Dubé, T., Yap, C.: A basis for implementing exact geometric algorithms (extended abstract) (1993)
4. Fortune, S., van Wyk, C.J.: Efficient exact arithmetic for computational geometry. In: Proceedings of the Ninth Annual Symposium on Computational Geometry, pp. 163–172 (1993)
5. Karamcheti, V., Li, C., Pechtchanski, I., Yap, C.: A core library for robust numeric and geometric computation. In: Proceedings of the Fifteenth Annual Symposium on Computational Geometry, SoCG, pp. 351–359 (1999)
6. Mörig, M., Rössling, I., Schirra, S.: On design and implementation of a generic number type for real algebraic number computations based on expression dags. Math. Comput. Sci. **4**(4), 539–556 (2010)
7. Mörig, M., Schirra, S.: Precision-driven computation in the evaluation of expression-dags with common subexpressions: problems and solutions. In: Kotsireas, I.S., Rump, S.M., Yap, C.K. (eds.) MACIS 2015. LNCS, vol. 9582, pp. 451–465. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32859-1_39
8. Pion, S., Fabri, A.: A generic lazy evaluation scheme for exact geometric computations. Sci. Comput. Program. **76**(4), 307–323 (2011)
9. Schirra, S.: Robustness and precision issues in geometric computation. In: Sack, J.R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 597–632. Elsevier, Amsterdam (2000)
10. Yap, C.: Towards exact geometric computation. Comput. Geom. **7**, 3–23 (1997)
11. Yu, J., Yap, C., Du, Z., Pion, S., Brönnimann, H.: The design of core 2: a library for exact numeric computation in geometry and algebra. In: Fukuda, K., Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 121–141. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15582-6_24