Restructuring Expression Dags for Efficient Parallelization

Martin Wilhelm

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

Abstract

In the field of robust geometric computation it is often necessary to make exact decisions based on inexact floating-point arithmetic. One common approach is to store the computation history in an arithmetic expression dag and to re-evaluate the expression with increasing precision until an exact decision can be made. We show that exact-decisions number types based on expression dags can be evaluated faster in practice through parallelization on multiple cores. We compare the impact of several restructuring methods for the expression dag on its running time in a parallel environment.

1 Introduction

In Computational Geometry, many algorithms rely on the correctness of geometric predicates, such as orientation tests or incircle tests, and may fail or produce drastically wrong output if they do not return the correct result [9]. The Exact Geometric Computation Paradigm establishes a framework for guaranteeing exact decisions based on inexact arithmetic, as it is present in real processors [13]. In accordance with this paradigm, many exact-decisions number types have been developed, such as `leda::real` [4], `Core::Expr` [5, 14], `Real_algebraic` [7] and `LEA` [1]. All four named number types are based on arithmetic expression dags, i.e., they store the computation history in a directed acyclic graph and use this data structure to adaptively (re-)evaluate the expression when a decision has to be made.

While all of these number types are able to make exact decisions, they are also very slow compared to standard floating point arithmetic. Therefore continuous effort is taken to make these data types more efficient. However, none of these number types implements a strategy to take advantage of multiple cores yet. In this work, we show that multithreading can improve the performance of expression-dag-based number types by presenting the design of a multithreaded implementation for `Real_algebraic`, as well as experimental results comparing it to its single-threaded version.

To enable an efficient parallelization, we implement several restructuring methods for the underlying data structure and compare them with respect to their effect on multithreading. We aim for using these techniques in a general purpose number type, for which the user need not worry about implementation details. Therefore we look specifically at situations in which restructuring increases the running time. We propose a new approach to avoid some of these situations and thus to lower the risk of worsening the performance.

1.1   Preliminaries

An (arithmetic) expression dag is a rooted ordered directed acyclic graph that is either
1. A single node containing a number or
2. A node representing a unary operation ($\sqrt[n]{\ }$, $-$) with one, or a binary operation ($+, -, *, /$) with two, not necessarily disjoint, arithmetic expression dags as children.

The number type `Real_algebraic` is based on the concept of accuracy-driven computation[1] [13]. Applying operations leads to the creation of an expression dag instead of the calculation of an approximation. When a decision has to be made, the maximum accuracy needed for the decision to be exact is determined and each node is (re)computed with a precision that is sufficient to guarantee the desired final accuracy[2].

Let val(E) be the value represented by an expression dag E. When an approximation for val(E) is computed, then for each node v in E, approximations for the children of v, and consequently for all of its descendants, must be computed before v itself can be processed. Hence the computations we have to perform are highly dependent on each other. Whether the evaluation of an expression dag can be efficiently parallelized is therefore largely determined by its structure. Generally, a shallower, more balanced structure leads to less dependencies and can be expected to facilitate a more efficient parallel evaluation.

1.2   Related work

Few attempts have been made to restructure arithmetic expression dags. Richard Brent showed in 1974 how to restructure arithmetic expression trees to guarantee optimal parallel evaluation time [2]. In 1985, Miller and Reif improved this strategy by showing how the restructuring process itself can be done in optimal time [6]. In our previous work, arithmetic expression dags are restructured to improve single-threaded performance by replacing tree-like substructures, containing only additions or only multiplications, by equivalent balanced trees [11]. We call this method *AM-Balancing.*

We implement a variation of Brent's approach for tree-like substructures in arithmetic expression dags and compare it with AM-Balancing. Furthermore, we refine the algorithm based on practical observations. We do not use the strategy by Miller and Reif, since restructuring the expression dag is very cheap compared to the evaluation of bigfloat operations and therefore only minor performance gain, if any[3], is to be expected.

2   Design

In this section we briefly describe our implementation of parallel evaluation and restructuring for the dag-based number type `Real_algebraic`. A more detailed description of the parallelization can be found in the associated technical report [12].

2.1   Parallelization

The running time for the evaluation of expression dags is dominated by the execution of bigfloat operations. They usually sum up to around 95% of the total running time. Therefore

---

[1] Also known, less accurately, by the name "Precision-driven computation"
[2] Actually, this is an iterative process with increasing accuracy and checks for exactness after each iteration.
[3] Considerable effort would be needed to even neutralize the overhead from creating and managing different threads.

we focus on parallelizing bigfloat operations and allow for serial preprocessing. Accuracy-driven evaluation is usually done recursively with possible re-evaluations of single nodes. For an efficient parallelization, it is necessary to eliminate recalculations to avoid expensive lock-usage. Therefore we assign the required precision to each node in a (serial) preprocessing step and afterwards evaluate the nodes in topological order as proposed by Mörig et al. [8].

In our approach, we assign one task to every bigfloat operation. Tasks that can be solved independently are then sorted into a task pool, where they may be executed in parallel. Each dependent task is assigned a dependency counter, which gets reduced as soon as the tasks on which they depend are finished. When the dependency counter reaches zero, the task gets sorted into the task pool. With this strategy we reduce the shared data to a minimum, such that atomic operations can be facilitated at critical points to eliminate race conditions.

The maximum number of threads working in the task pool can be adjusted, depending on the hardware available. Our tests are run with a maximum number of four threads simultaneously working on the tasks plus the main thread, which stays locked during the computation.

2.2 Restructuring

In AM-Balancing, so-called "operator trees", i.e., connected subgraphs consisting of operator nodes with only one predecessor, are replaced by equivalent balanced trees if they represent a large sum or a large product. Since sums and products are associative, this can be done without increasing the total number of operations and therefore without a large risk of significantly increasing the running time, aside from some subtleties [11].

In this work we extend AM-Balancing, based on the tree restructuring by Brent [2]. In Section 2.2.2 we consider operator trees consisting of all basic arithmetic operations except roots and reduce their depth by continuously splitting long paths and moving the lower half to the top. In Section 2.2.3 we describe a modification to this algorithm, which avoids some steps that are particularly expensive.
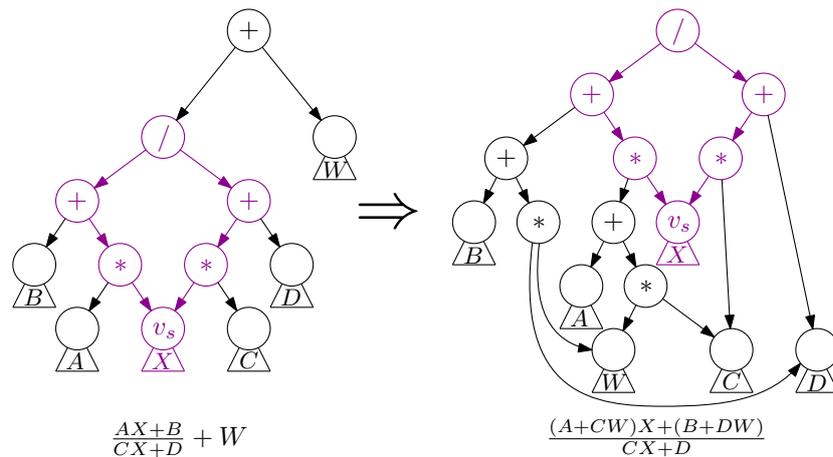
2.2.1 Notation

Let E be an expression dag. We call a subgraph T an *operator tree* if T is a maximal connected subgraph of E, consisting of nodes of type $\{+, -, *, /, \text{- (unary)}\}$ such that no node except the root of T has more than one parent. We denote the set of operator nodes having one of the allowed operator types V_{op}. We call the children of the leaves of an operator tree T its *operands* and let $\phi(T)$ denote their number.

An operator tree is always a tree and all operator trees in an expression E are disjoint. Therefore each node $v \in V_{\mathrm{op}}$ is part of exactly one operator tree $T(v)$. For each $v \in V_{\mathrm{op}}$ we call the operands of $T(v)$ that are part of the subtree rooted at v the *operands of v* and denote their number by $\phi(v)$. We call a path from the root of an operator tree to a leaf *critical* if each node on the path has at least as many operands as its siblings.

2.2.2 Move-To-Root Restructuring

We briefly describe our variation of Brent's algorithm, which we refer to as *Move-To-Root (MTR) Restructuring*. Each operator tree T in an expression dag E is restructured separately. In each node v in T we store $\phi(v)$, i.e., the number of operands of v. Then we search for a split node v_s on a critical path of T, such that $\phi(v_s) \leq \frac{1}{2}\phi(T) < \phi(\mathrm{parent}(v_s))$.

Let X be the subexpression at v_s. We restructure T, such that it now represents an equivalent expression of the form $\frac{AX+B}{CX+D}$ with (maybe trivial) subexpressions A, B, C, D.

Figure 1 Incorporating an addition into an expression of the form $(AX + B)/(CX + D)$. The main structure of the expression is highlighted before and after the restructuring. The expression dag grows by two additional multiplications and one additional addition due to the denominator.

Starting with the expression X at v_s, we raise v_s to the top of the tree while maintaining the form $\frac{AX+B}{CX+D}$. We say that we *incorporate* the operations along the way from v_s to the root into the expression. The restructuring needed to incorporate an addition is shown exemplarily in Figure 1.

After restructuring, the length of the path from the root to v_s is reduced to a constant. The same applies to the respective root nodes for the expressions A, B, C and D. The algorithm then recurses on these nodes, i.e., the operator trees representing X, A, B, C, D are restructured.

2.2.2.1   Comparison to Brent's algorithm

Like MTR Restructuring, Brent's original algorithm searches for a split node v_s on a critical path and raises it to the top of the expression tree. However, it does so in a more sophisticated manner by repeatedly splitting the path from the root to v_s in half and at the same time balancing the "upper half" of the tree, i.e., the part of the tree that is left when removing the subtree rooted at v_s. MTR Restructuring uses a simpler approach, which moves the split node to the top first and balances the remaining parts afterwards. It is easier to implement, but leads to an increased depth when subexpressions are reused, since nodes cannot be restructured as part of a larger operator tree if they have multiple parents. We chose this approach, since it is better suited to test possible improvements and can still be expected to behave similar to Brent's algorithm in many situations.

2.2.3   Parameterization of MTR Restructuring

Different operations on bigfloats have different costs[4]. In the algorithm introduced in the previous section, divisions in the upper half of a critical path are risen to the top. Since divisions are expensive, this is beneficial if two divisions fuse and one of them can be replaced by a multiplication. However, each addition or subtraction that is passed adds one or two expensive multiplications to the expression (cf. Figure 1).

If this affects a large number of additions and subtractions, the benefit of raising the division vanishes. If the number of cores and therefore the expected gain from an optimal

---

[4] In our tests on `mpfr` bigfloats, multiplications take on average 10-20 times as long as subtractions or additions and divisions take 1.5-2 times as long as multiplications. Interestingly, this behavior appears

---

Algorithm 1: The main part of Parameterized MTR Restructuring. A counter for the number of additions and subtractions above the current node is maintained, which may lead to a split at division nodes. If the current node is a split node, an expression of the form $(AX + B)/(CX + D)$ gets initialized. Otherwise the recursion continues and the node gets incorporated into the expression of the child (cf. Figure 1). Finally the root node (i.e. the operator tree) gets replaced by the new expression dag and the subexpressions get restructured.

---

1 Function restructure(*root***):**
2   **if** *root can be restructured* **then**
3 $exp = \text{raise}(root, root, 0)$
4   set *root* to *exp*
5   restructure(*exp.A*); restructure(*exp.B*); restructure(*exp.C*);
           restructure(*exp.D*); restructure(*exp.X*)
6   **end**

7

8 Function raise(*node, root, counter***):**
9   Create new Expression *exp*
10   **if** $\phi(node) \leq \phi(root)/2$ **or** *node is division and counter* $> THRESHOLD$ **then**
11   *exp*.init(*node*)
12   **else**
13   **if** *node is addition or subtraction* **then**
14   *counter++*
15   **end**
16   **if** *node is division* **then**
17   *counter = 0*
18   **end**
19   **if** *node has no right child* **or** $\phi(node.left) \geq \phi(node.right)$ **then**
20 $exp = \text{raise}(node.left, root, counter)$
21   **else**
22 $exp = \text{raise}(node.right, root, counter)$
23   **end**
24   *exp*.incorporate(*node*)
25   **end**
26   return *exp*

---

parallelization is small, it might be worthwile to allow for an increased depth to save multiplications. Our modified algorithm works the same as the algorithm described in Section 2.2.2, except it counts the number of additions and subtractions along a critical path and splits at a division node if their number surpasses a certain threshold, even if the division node still contains more than half of the operands of the operator tree. If a division node is passed while the counter is still smaller than the threshold, the counter is reset to zero, since then the additions and subtractions above the division node cause additional multiplications anyway. We refer to this strategy as *Parameterized MTR Restructuring*. A sketch of the main method is shown in Algorithm 1.

---

to be almost independent of the size of the bigfloats.

In the worst case, we may split at a linear number of division nodes if between two division nodes are just enough additions or subtractions to pass the threshold. Then the height of the operator tree, and therefore the running time with arbitrarily many processors, grows by a linear amount. However, each split increases the height of the tree only by a constant. We expect a similar strategy to be applicable to Brent's original algorithm, where divisions are risen to the top as well. Since a more complex counting procedure would be necessary to adapt the parameterization, this hypothesis is not evaluated in more detail in this paper.

3 Experiments

We perform several experiments on a dual core machine (named `dual_core`) with an Intel Core i5 660, 8GB RAM and a quad core machine (named `quad_core`) with an Intel Core i7-4700MQ, 16GB RAM. For `Real_algebraic` we use Boost interval arithmetic as floating-point-filter and MPFR bigfloats for the bigfloat arithmetic. The code is compiled on Ubuntu 17.10 using `g++` 7.2.0 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. The variance for each data point is small (the total range is usually in $\pm 5\%$ of its value). In each experiment the real value of the respective expression is computed to $|q| = 50000$ binary places.

We analyze four different restructuring strategies: No restructuring (`def`), AM-Balancing (`amb`), MTR Restructuring (`mtr`) and Parameterized MTR Restructuring with a threshold of five (`mtr[5]`). In all of the four strategies the evaluation is done in topological order to avoid distortion of the results. For each strategy we compare the results with and without multithreading (`m`).

3.1 Binomial coefficient

The AM-Balancing method is particularly effective if the expression contains large sums or large products. It was conjectured that applying this restructuring method makes an expression dag more suitable for a parallel evaluation. We calculate the generalized binomial coefficient
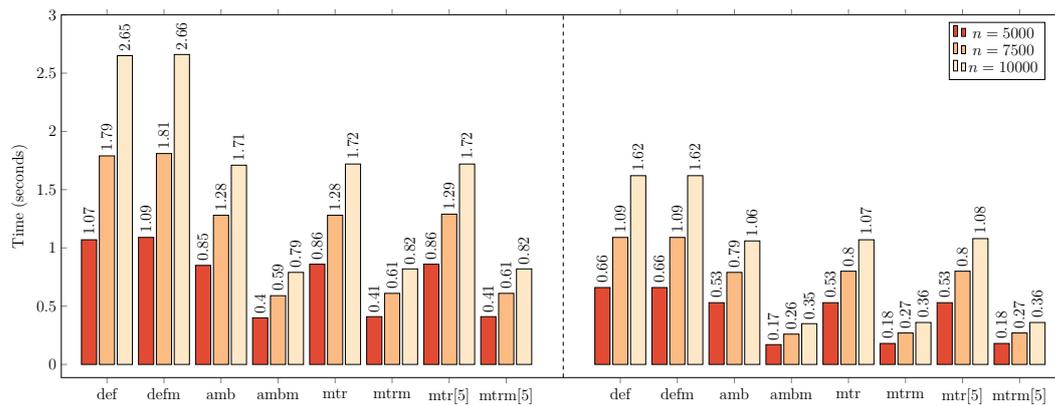
$$\binom{\sqrt{13}}{n} = \frac{\sqrt{13}(\sqrt{13}-1)\cdots(\sqrt{13}-n+1)}{n(n-1)\cdots 1}$$

iteratively as in the AM-Balancing paper (cf. [11]).

```
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);
}
```

In this method, both the numerator and the denominator of `bc` are large, sequentially computed products. Both of them can be balanced without adding additional operations because of the associativity of the multiplication. We run `bin_coeff` in our test environment.

The results are shown in Figure 2. Switching to multithreading while retaining the structure of the expression dag does not have a positive effect on the performance, since the operator nodes are highly dependent on each other. Applying AM-Balancing does not only directly increase the performance, but also makes the structure much more favorable for parallel evaluation. On the dual core machine the maximal possible performance gain is achieved. With a quad core we still get an improvement, but only by a factor of about 2.8.

Figure 2 The results of the binomial coefficient test for `dual_core` (left) and `quad_core` (right). The original structure of the expression dag is not suited for a parallel evaluation. AM-Balancing leads to a beneficial structure. MTR Restructuring and Parameterized MTR Restructuring have a similar effect as AM-Balancing.

For large additions and large multiplications, MTR Restructuring behaves similar to AM-Balancing in the sense that it builds an (almost) balanced tree. So, unsurprisingly, the running times for MTR Restructuring and Parameterized MTR Restructuring closely resemble the results for AM-Balancing.
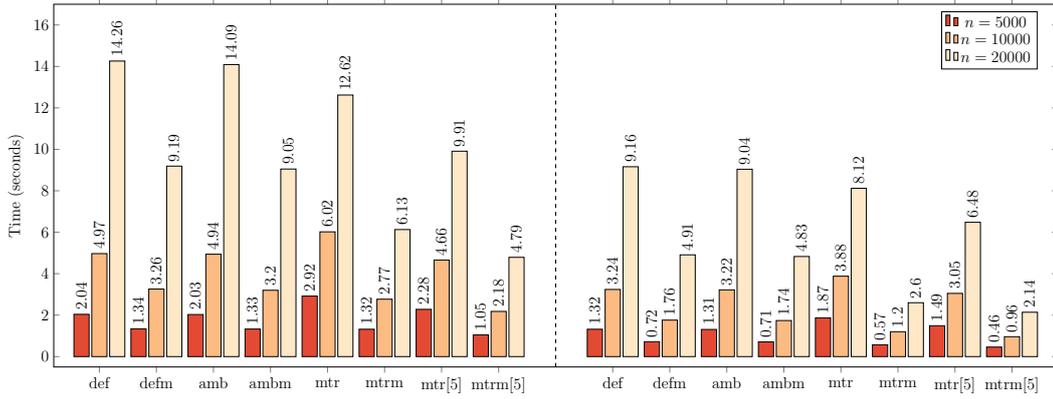
3.2 Random operations

A different behavior of the restructuring methods is to be expected if algorithms use many different operators in varying order. We simulate this behavior by performing random operations on an expression.

```cpp
template <class NT> void random_operations(const int n, const long q, ↵
    const int FADD = 1, const int FSUB = 1, const int FMUL = 1, const int↵
    FDIV = 1){
    std::random_device rd; std::mt19937 gen(rd());
    std::uniform_int_distribution<> idis(0,FADD+FSUB+FMUL+FDIV-1);
    std::exponential_distribution<> rdis(1);

    double r; NT result = NT(1);
    for (int i = 0; i < n; ++i) {
        const int nbr = idis(gen);
        do { r = rdis(gen); } while (r == 0);

        if (nbr < FADD) result += sqrt(NT(r));
        else if (nbr < FADD+FSUB) result -= sqrt(NT(r));
        else if (nbr < FADD+FSUB+FMUL) result *= sqrt(NT(r));
        else result /= sqrt(NT(r));
    }
    result.guarantee_absolute_error_two_to(q);
}
```

We exploit two kinds of randomness in this test. First, we randomly choose one of the operators $\{+, -, *, /\}$. The parameters `FADD`, `FSUB`, `FMUL` and `FDIV` determine their respective fractions of the total number of operators. If all of them are set to one, the operations are equally distributed. Second, we randomly choose a real positive number on which to apply the operator. We use an exponential distribution with a mean of one instead of a uniform
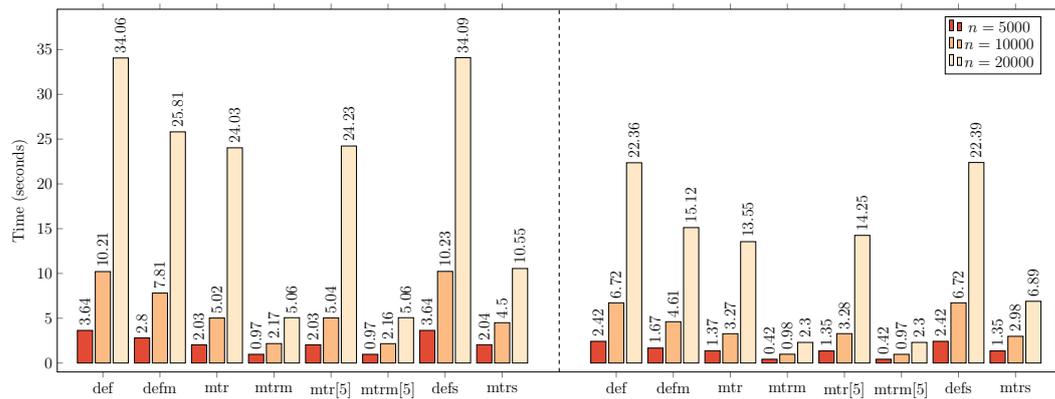
Figure 3 Results for performing random operations uniformly distributed on `dual_core` (left) and `quad_core` (right). AM-Balancing has no significant effect. MTR Restructuring and Parameterized MTR Restructuring worsen the single-threaded performance for small numbers of operands, but improve the multithreaded performance. Parameterized MTR Restructuring performs better than MTR Restructuring in all cases.

distribution. `Real_algebraic` behaves differently for very large or very small numbers. In a uniform distribution most of the random numbers are larger than one. Therefore through repeated multiplication (division) numbers get very large (very small). Since we want to be able to compare the actual costs of multiplications and divisions, we have to avoid distorting the results through side effects of the experiment.

In our test we use the square roots of the random floating point numbers we get to generate numbers with an infinite floating point representation. When performing a division, `Real_algebraic` must check whether the denominator is zero. So it must decide at which point the established accuracy is sufficient to guarantee that the value of an expression is zero. This decision is made by comparing the current error bound to a separation bound, which is a number $\text{sep}(X)$ for an expression X, for which $|\text{value}(X)| > 0 \Rightarrow |\text{value}(X)| > \text{sep}(X)$. The separation bound we use is a variation of the separation bound by Burnikel et al. [3, 10]. It relies on having a meaningful bound for the algebraic degree of an expression, which we cannot provide in an expression with a lot of square roots. However, since we know that none of the denominators we get during our experiments can become zero, we can safely set the separation bound to zero and stop computing as soon as we can separate our denominator from zero with an error bound.

The results for a uniform distribution of operators are shown in Figure 3. Although the structure of the dag is largely unsuited for parallelization, we can see a significant difference between the single-threaded and the multi-threaded variant even without restructuring due to the parallel evaluation of the square root operations. AM-Balancing shows no effect at all on the overall performance.

For smaller inputs, MTR Restructuring has a negative effect on the single-threaded performance. The predominant cause of this is the propagation of divisions as described in Section 2.2, which results on average in about nine multiplications per processed division. Parameterized MTR Restructuring reduces this ratio to about 5.5 multiplications per processed division by leaving about one tenth of them unprocessed. For large inputs, MTR Restructuring has a positive effect even without multithreading due to the decrease in depth of the main operator tree and, consequently, a decrease in the maximum accuracy needed for the square root operations in its leaves (cf. [11]).

Figure 4 Test results for random operations with mostly divisions on `dual_core` (left) and `quad_core` (right). MTR Restructuring causes an enormous increase in performance. The large jump between single- and multithreading can be partially ascribed to a slight change in implementation that shows a positive effect on the restructured expression dag. Applying this change without multithreading leads to the results shown in `defs` and `mtrs`. Parameterized MTR Restructuring behaves similar to MTR Restructuring.
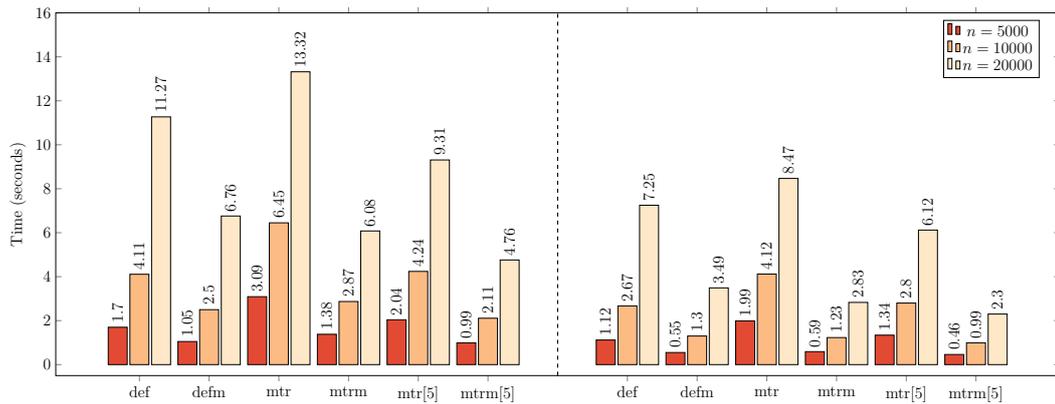
In the case of multithreading, both approaches based on Brent's algorithm show the desired effect, increasing the speedup for the dual core from around 1.7 to an optimal 2 and for the quad core from around 1.8 to about 2.8. As a consequence, they are able to beat direct parallelization in every test case. Parameterized MTR Restructuring performs slightly better than MTR Restructuring due to the increase in single-threaded performance while maintaining the same speedup.

3.3 Random operations with mostly divisions

Raising divisions to the top is bad only if many additions and subtractions are passed during the procedure. If instead divisions can be combined and therefore replaced by multiplications, the overall effect is positive. We test this by shifting the operator distribution such that nine out of ten operators are divisions, i.e., by setting `FADD = 1`, `FSUB = 1`, `FMUL = 1`, `FDIV = 27`. Since AM-Balancing shows no differences compared to the default number type, we exclude it from further experiments.

Considering the results shown in Figure 4 it becomes evident that restructuring alone reduces the running time significantly in this situation. When switching to multithreading we get an even bigger improvement, which, however, can not be explained by parallelizing alone. Instead about half of the improvement stems from an implementation detail when computing the separation bound for separating denominators from zero in the multithreaded version. The change usually leads to a slight overhead, but also happens to prevent a slow-down if many checks have to be made. It is explained in detail in the associated technical report [12]. After restructuring we have many big denominators, for which a separation bound must be computed and therefore we get an improvement. The default number type, on the other hand, does not benefit from the new separation bound computation strategy. The data points for `defs` and `mtrs` represent the test results for single-threaded evaluation with the change in separation bound computation applied.

In contrast to our previous test, Parameterized MTR Restructuring does not perform better than MTR Restructuring, since there are few to none situations in which the condition for the improvement gets triggered. However, more importantly, the modified algorithm also does not perform worse than the original one.

Figure 5 Test results for random operations with few divisions on `dual_core` (left) and `quad_core` (right). MTR Restructuring worsens the single-threaded performance and is not able to outperform direct multithreading for smaller inputs. Parameterized MTR Restructuring peforms better in all tests, although having a worse speedup factor on the quad core.

3.4 Random operations with few divisions

With few divisions, compared to the number of additions and subtractions, we should expect the opposite effect from the previous experiment. MTR Restructuring should lead to an decrease in singlethreaded performance and Parameterized MTR Restructuring should perform better than MTR Restructuring. We set our input parameters to `FADD = 3, FSUB = 3, FMUL = 3, FDIV = 1`, such that only one out of ten operations is a division.
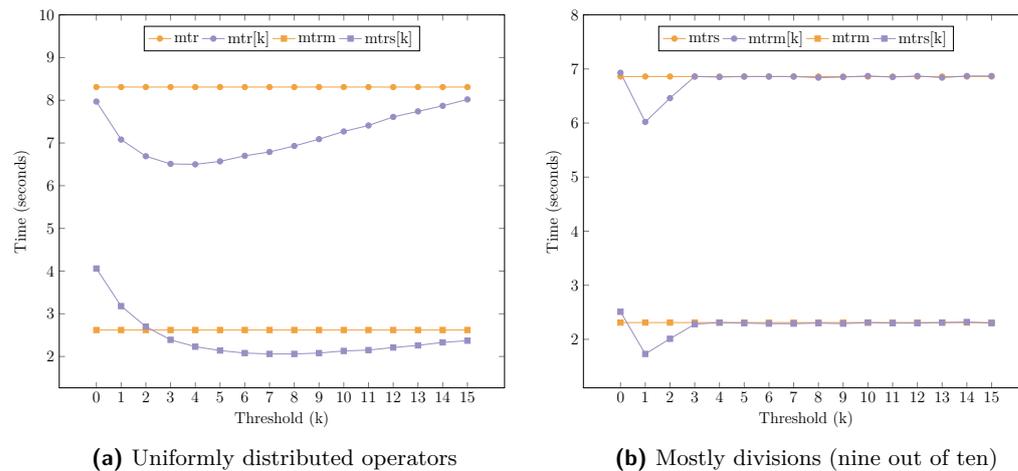
Our test results confirm these expectations (cf. Figure 5). For small inputs, MTR Restructuring performs worse than no restructuring even in a parallel environment. Parameterized MTR Restructuring on the other hand performs better than the default in all parallel tests except for the test with the smallest number of operands on a dual core. This effect strengthens when the number of divisions further decreases.

However, it should be noted that while on a dual core the speedup through parallelization is optimal for both variants, on a quad core MTR Restructuring allows for a speedup of about 3, whereas Parameterized MTR Restructuring only reaches a speedup of around 2.7. The modified algorithm leaves about forty percent of the divisions untouched for this operator distribution, which manifests in a significant effect on the expression dag's degree of independence.

3.5 Parameter-dependence of Parameterized MTR Restructuring

In the experiments in the previous sections we set the threshold k for Parameterized MTR Restructuring to five without an explanation. In this section we make evident that, while there is always an optimal choice for this parameter, most choices are not actually bad compared to standard MTR Restructuring.

The threshold indicates the number of additions and subtractions that are affected by incorporating a division node into the new structure and therefore sets the benefit from raising a division to the top in relation to the number of additional multiplication nodes it causes (cf. Section 2.2.3). If $k = 0$, restructuring never incorporates divisions in its expressions, therefore all expressions are of the form $AX + B$. For $k = 1$ divisions are only incorporated if they are followed exclusively by divisions, multiplications or negations. With increasing k it becomes less frequent that a division cannot be passed during restructuring, leading to Parameterized MTR Restructuring behaving more and more similar to MTR Restructuring.

(a) Uniformly distributed operators   **(b)** Mostly divisions (nine out of ten)

■ **Figure 6** Comparison of different parameter choices for Parameterized MTR Restructuring with $n = 20000$ random operations on `quad_core`. For large values of k, the parameterized approach becomes increasingly similar to MTR Restructuring. For small values of k the original approach is slightly better in the multithreaded version. The optimal choice for k depends on the division ratio.

In Figure 6 the running time for the experiments from Section 3.2 and Section 3.3 for different values of k are shown. For the second experiment, we use the modified separation bound computation strategy to ensure comparability (cf. Section 3.3). The results demonstrate that most of the choices for k improve the performance of the algorithm. Also, they confirm that for high k the parameterized approach is almost identical to the original algorithm.

For $k = 0$ the parameterized approach performs worse in both experiments in the parallel version, although slightly increasing the single-threaded performance for uniformly distributed operators. Surprisingly, Parameterized MTR Restructuring is faster than MTR Restructuring for $k = 1$ when nine out of ten operators are divisions, despite in this case restructuring tends to replace divisions by multiplications, which then can be balanced due to their associativity. However, since only one out of fifteen operations is an addition or subtraction, the loss of independence is small compared to the gain from avoiding additional multiplications.

The optimal choice for k depends on the ratio between divisions and additions/subtractions. If this ratio gets smaller, the optimal k increases. At the same time, the difference between different choices for k decreases. Therefore for a small ratio and smaller values of k the parameterized approach still performs better than the original version.

4 Summary

Multithreading can be an effective tool to speed up the performance of expression-dag-based number types. Applying MTR Restructuring to expression dags allows us to benefit from multithreading even when faced with a structure with a high number of dependencies, although bearing the risk of lowering the performance. AM-Balancing can create favorable structures with low risk of worsening the end result, but is not widely applicable. The parameterized version of MTR Restructuring lowers the risk of significant performance loss, while at the same time maintaining most of the benefits of the original algorithm. In a general purpose number type, we therefore suggest using Parameterized MTR Restructuring, with a sensible choice of k, if the evaluation should be done in parallel. For k, a small number greater than zero is advisable.

5   Future Work

This work addresses parallelization on multiple CPUs. While it seems unlikely that complex expression dags can be efficiently parallelized on a GPU, it may be possible to do so for the underlying bigfloats. Since bigfloat operations still constitute the most expensive part of exact-decisions number types, this may lead to a significant speedup. Furthermore in this work we only restructure tree-like subgraphs to avoid (possibly exponential) blow-up of our structure. However, with a larger number of cores it might be worthwile to split up some nodes with multiple parents to eliminate or shorten critical paths or at least weigh such nodes accordingly in the higher-level operator trees. Finally, the performance gain due to the parameterization cannot be fully attributed to interactions between divisions and additions. It may prove useful to extend the new strategy to consider multiplications over large sums.

―――― **References** ――――

1   Mohand Ourabah Benouamer, P. Jaillon, Dominique Michelucci, and Jean-Michel Moreau. A lazy exact arithmetic. In *11th Symposium on Computer Arithmetic, 29 June - 2 July 1993, Windsor, Canada, Proceedings.*, pages 242–249, 1993. `doi:10.1109/ARITH.1993.378086`.

2   Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

3   Christoph Burnikel, Stefan Funke, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt. A separation bound for real algebraic expressions. *Algorithmica*, 55(1):14–28, 2009. `doi:10.1007/s00453-007-9132-4`.

4   Christoph Burnikel, Kurt Mehlhorn, and Stefan Schirra. The leda class real number. Report MPI-I-1996-1-001, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1996.

5   Vijay Karamcheti, C. Li, Igor Pechtchanski, and Chee-Keng Yap. A core library for robust numeric and geometric computation. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry, Miami Beach, Florida, USA, June 13-16, 1999*, pages 351–359, 1999. `doi:10.1145/304893.304989`.

6   Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 478–489, 1985.

7   Marc Mörig, Ivo Rössling, and Stefan Schirra. On design and implementation of a generic number type for real algebraic number computations based on expression dags. *Mathematics in Computer Science*, 4(4):539–556, 2010. `doi:10.1007/s11786-011-0086-1`.

8   Marc Mörig and Stefan Schirra. Precision-driven computation in the evaluation of expression-dags with common subexpressions: Problems and solutions. In *6th International Conference on Mathematical Aspects of Computer and Information Sciences, MACIS*, pages 451–465, 2015. `doi:10.1007/978-3-319-32859-1_39`.

9   Stefan Schirra. Robustness and precision issues in geometric computation. In *Handbook of Computational Geometry*, pages 597–632. Elsevier, 2000.

10   Susanne Schmitt. Improved separation bounds for the diamond operator. Report ECG-TR-363108-01, Effective Computational Geometry for Curves and Surfaces, Sophia Antipolis, France, 2004.

11   Martin Wilhelm. Balancing expression dags for more efficient lazy adaptive evaluation. In *Mathematical Aspects of Computer and Information Sciences - 7th International Conference, MACIS 2017, Vienna, Austria, November 15-17, 2017, Proceedings*, pages 19–33, 2017. `doi:10.1007/978-3-319-72453-9_2`.

12   Martin Wilhelm. Multithreading for the expression-dag-based number type Real_algebraic. Technical Report FIN-001-2018, Otto-von-Guericke-Universität Magdeburg, 2018.

13   Chee-Keng Yap. Towards exact geometric computation. *Comput. Geom.*, 7:3–23, 1997. `doi:10.1016/0925-7721(95)00040-2`.

14   Jihun Yu, Chee-Keng Yap, Zilin Du, Sylvain Pion, and Hervé Brönnimann. The design of core 2: A library for exact numeric computation in geometry and algebra. In *Proceedings of the Third International Congress on Mathematical Software, ICMS*, pages 121–141, 2010. `doi:10.1007/978-3-642-15582-6_24`.