MPI-Based Tensor Network Library

Semester Thesis by Simon Etter

Advisors: Robert Gantner, Vladimir Kazeev
and Prof. Dr. Christoph Schwab

6 May 2014

ETH Zürich

Contents

1 Introduction

In many fields of computational science and engineering, the need arises to work with tensors $A \in \mathbb{C}^{n_1 \times \ldots \times n_d}$ whose dimension d is "large", i.e. larger than the well-known special cases $d = 1$ (vectors) and $d = 2$ (matrices). Explicitly handling such high-dimensional objects is very difficult as both memory requirements and computational complexity scale with at least $\prod_{i=1}^{d} n_i \leq n^d$, where n is an upper bound on the n_i. This exponential scaling is called the curse of dimensionality as it renders explicit tensor algorithms unaffordable in more than three dimensions. In recent years, an approach based on so-called tensor decompositions has been developed to the point where it allows to trade accuracy for computational efficiency both theoretically as well as algorithmically. The common idea of these decompositions is the separation of variables, i.e. the representation of a single high-dimensional tensor as a product of several tensors of small dimensions. This product of tensors can be depicted as a tensor network [1]. The corresponding representations of the data are called tensor network formats. The two most essential examples of tensor network formats are the Tensor Train (TT) format [2, 3] and the Hierarchical Tensor Representation (HTR) [4, 5] illustrated in Figure 1.1. A far more detailed introduction to tensor networks can be found in the book by Hackbusch [1]. A literature survey with a strong emphasis on applications is given in [6].

Depending on the structure of the data, tensor network formats can dramatically reduce the data complexity of tensors while losing only very little in accuracy. Nevertheless, such compressed tensors may still require memory on the order of 10 GB and more, and basic operations like addition, dot products or truncation can take several minutes to complete. There is thus the need to speed up computations by making the most out of today's hardware. Tensor network algorithms typically spend the most time in calls to BLAS functions, thus their single-core optimization reduces to linking against an optimized BLAS library. The next level of optimization is then to parallelize the algorithms over multiple CPUs. As it turns out, not all tensor network formats lend themselves for parallelization. For example, important algorithms on TT-formatted tensors cannot be efficiently parallelized because they require the vertices to be processed

(a) TT format

(b) HTR

Figure 1.1: A six-dimensional tensor in the TT format and the HTR.

in a strict left-to-right or right-to-left order. On the other hand, HTR algorithms require only that parent vertices must be processed before their children or vice versa, but they don't impose any order on vertices which are not ancestors/descendants of each other. This leaves a certain degree of freedom to distribute the vertex processing over multiple processing units.

The main objective of this semester thesis is to pioneer the high-performance parallel implementation of the orthogonalization and truncation of high-dimensional tensors in the HTR. For that purpose, we developed basic data structures and functionality for a C++ library based on MPI. In Chapter 2, we discuss the software design considerations underlying the library. In Chapter 3, we analyze the scaling of tensor network algorithms on tree-structured networks. First, we provide a theoretical analysis. Then, we match our conclusions against experimental results obtained with our library.

The extensible classes idiom and the `full_tensor` class were developed by the author already before the start of this semester thesis. For completeness of presentation, we nevertheless present these two topics here.

2 Software Design

2.1 Extensible Classes

2.1.1 Considerations

In object-oriented programming, one associates algorithms (functions) with the data structures (classes) they operate on by making the functions members of the classes. Usually, the set of class member functions is rather small and fixed at an early stage in the software development process. It is a peculiarity of linear algebra that we have a few data structures which have a large number of algorithms operating on them. Consider matrices as an example: They are associated with algorithms for various arithmetic operations, solving linear systems of equations, determining matrix decompositions, etc. When translating such a many-algorithms data structure into a class, new issues arise that are not present in the usual class design process:

- We want the set of algorithms to be extensible.

  As there are many algorithms, we must expect them to be implemented at different times by different developers. It is therefore important to have a good scheme for adding new functions.

- We want to split the functions into different modules.

  Apart from operating on the same data structure, the algorithms are not related. We therefore would like to keep their implementations as cleanly separated as possible.

- We want to have a clean interface between data structure and algorithms.

  Since the amount of code written on top of the basic data structure code will be fairly large, we should abstract away the implementation details of the data structure and provide a clean interface to the algorithms instead.

2.1.2 Solution

The classic way of solving the above problems is to implement a class `data_structure` and implement the algorithms as functions taking `data_structure` objects as argument. However, for syntax reasons we wanted the algorithms to be member functions, i.e. `alg(data)` should better be written as `data.alg()`. The latter syntax is used extensively in the Eigen library [7], which served as a role model for our project. Unfortunately, in C++ it is not possible to make the algorithms member functions of `data_structure`

and at the same time satisfy the above objectives. We therefore need a new idiom which is illustrated below.

```
// ----------------
// data_structure.h
// ----------------
class data_structure_implementation {
public:
    // Public interface
protected:
    // Algorithms-specific interface
private:
    // Implementation details
};

#define BEFORE_CLASS
#include "extensions.h"
#undef BEFORE_CLASS
class data_structure : public data_structure_implementation {
    #define IN_CLASS
    #include "extensions.h"
    #undef IN_CLASS
}
#define AFTER_CLASS
#include "extensions.h"
#undef AFTER_CLASS

// ------------
// extensions.h
// ------------
#include "algorithm1.h"
#include "algorithm2.h"
// etc

// ------------
// algorithm1.h
// ------------
#ifdef BEFORE_CLASS
// Includes, non-member function declarations, helper structs, etc.
#endif /* BEFORE_CLASS */

#ifdef IN_CLASS
// Member functions
#endif /* IN_CLASS */
```

```
#ifdef AFTER_CLASS
// Non-member function definitions
#endif /* AFTER_CLASS */
```

The trick here is to use the preprocessor to assemble the class declaration at compile time. All algorithms are implemented in their own file, cleanly separated from each other. Once the preprocessor runs through `data_structure.h`, it reads each algorithm file three times, but due to the macro definitions it copies a different section of the files to `data_structure.h` in each run. This allows to add or remove member functions to/from `data_structure` by simply adding or removing a line in `extensions.h`.

Introducing a class `data_structure_implementation` allows the data structure to hide its implementation details from the extensions.

2.2 The `full_tensor` Class

The first actual example of a data structure with many associated algorithms in our project are the full tensors. All the operations on them can be formulated easily and efficiently according to the following pattern:

- Reshape the tensor to a matrix
- Perform some operations on the matrix
- (Reshape the result back into a tensor)

The advantage of this pattern is that it allows us to forward most performance-critical operations to a highly optimized BLAS library. We thus organize the `full_tensor` class according to the extensible classes idiom, and provide matricization as the basic interface for the algorithms to access the data structure. By matricization we mean the reformulation of the tensor in a format understandable by the BLAS.

It remains to decide on a storage format for efficient handling of matricization requests. The straightforward solution is to store a d-dimensional tensor A with mode sizes $n_0, \ldots, n_{d-1}$ in a long vector `a` of size $\prod_{j=0}^{d-1} n_j$ such that $\texttt{a}\left[\sum_{j=0}^{d-1} \prod_{k=0}^{j-1} n_k i_j\right] = A(i_0, \ldots, i_{d-1})$. However, such a storage format does not support matricization requests very well, as we will show next.

Assume we are given a three-dimensional tensor and need to return a $((1),(0,2))$-matricization, i.e. the second mode should be viewed as the row mode and the combination of the first and third mode as the column mode. All existing BLAS libraries require the input matrices to have constant strides in both row and column direction, i.e. given a pointer `p` pointing to $A(i,j)$, there have to exist two numbers `row_stride` and `column_stride` independent of i, j such that `p + row_stride` points to $A(i+1,j)$ and `p + column_stride` to $A(i,j+1)$. In our example, `row_stride` is equal to n_1 and

thus meets the requirements. On the other hand, `column_stride` is given by

$$\texttt{column\_stride} = \begin{cases} 1 & j \bmod n_0 < n_0 - 1 \\ n_0 n_1 & \text{otherwise} \end{cases}$$

i.e. it depends on the position j. Therefore, to return the requested matricization we first have to permute the data.

It is clear that for $d > 2$ there is no storage format which can serve all matricization requests without permuting the data. But to keep the number of permutations low, we use the following generalization of the naive format: We introduce a permutation $\pi : [d] \to [d]$ with $[d] := \{0, \dots, d-1\}$. The elements of A are then stored in `a` according to $\texttt{a}\left[\sum_{j=0}^{d-1} \prod_{k=0}^{j-1} n_{\pi(k)} i_{\pi(j)}\right] = A(i_0, \dots, i_{d-1})$. When the user asks for a (μ, ν)-matricization, we update π to either of (μ, ν) or (ν, μ) and update the data in `a` accordingly. In the former case, we then have `row_stride` $= 1$ and `column_stride` $= \prod_{j=1}^{|\mu|} n_{\mu(j)}$, in the latter `row_stride` $= \prod_{j=1}^{|\nu|} n_{\nu(j)}$ and `column_stride` $= 1$. Like this, if the user asks for n equal matricizations in a row, we have to permute at most once, whereas the naive format requires up to n permutations.

2.3 The Tensor Network Classes

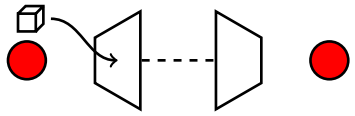
2.3.1 Considerations

In tensor networks, both the data as well as the computational workload are naturally associated to single vertices of the network. We therefore parallelize our code by distributing the vertices over the MPI processes. As we will see later, finding a good distribution is not a trivial task even when assuming uniform costs for each vertex. If additionally the vertex costs could become unbalanced, we might need to adapt the distribution accordingly to achieve good performance. It is thus best to implement the graph data structure and the algorithms on them such that they can work with a general vertex distribution.

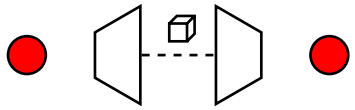
Another aspect in which we want to maintain generality is the graph structure. Despite our primary focus lying on tree-type and, in particular, HTR networks, our code can be extended to work with any kind of network. If at some point in the future new promising types of tensor networks appear, it will thus be easy to implement them within our library.

2.3.2 The `edge_port` Class

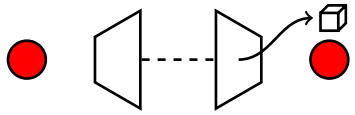
In tensor network algorithms, the flow of data is constrained to the edges of the network. Since vertices are distributed over the processes, we can distinguish between edges having both endpoint vertices on the same process or on two different processes. While the forwarding of the data has to be handled differently for these two kinds of edges, all we care about when implementing the algorithms is that the data eventually appears on the other side. We therefore introduce an `edge_port` class to abstract away the



(a) The left vertex pushes a piece of data into its end of the edge.



(b) The data is stored within the edge. If the edge is between vertices on different processes, the data is passed asynchronously between the processes.



(c) The vertex to the right retrieves the data from its side of the edge.

Figure 2.1: Illustration of message passing through `edge_ports`.

implementation detail of how the data is transferred. Since the sender and receiver could be the same process, these `edge_ports` have to work much like a service hatch: if you need to pass data from the "kitchen" to the "dining room", you push it into your end of the edge represented by a local `edge_port`. There, it waits until the receiver (which may be the same process as the sender) is ready to retrieve the data from its side of the edge. This process is illustrated in Figure 2.1.

2.3.3 Organization Of The Classes

An easy and efficient format for distributed graphs are adjacency lists. In this format, each process stores a list of its local vertices and each vertex stores a list of the edges to which it belongs. The Parallel Boost Graph Library [8] is an example of an existing library implementing adjacency lists. To represent such a format in C++, we use five classes whose relationships are depicted in Figure 2.2.

When implementing a specific tensor network format, we might need to add new functions and fields to the tensor network classes, or redefine the behaviour of some functions. For example, if the network is a rooted tree we might need to mark one edge per vertex as the edge to the parent vertex, and rewrite the `dot` operation which can be implemented easier and more efficiently in this special case. The promoted object-oriented solutions to this problem are inheritance and polymorphism. In order to add or modify members of a class `base`, we introduce a new class `derived` inheriting from `base` and implement the changes there. The use of the built-in polymorphism mechanism in

C++ (i.e. the `virtual` keyword) is problematic, however, due to at least the following two reasons.

- Performance overhead

  If a function `foo()` is declared `virtual`, each call to `foo()` is resolved at run time even if we could already tell at compile time which version of `foo()` has to be called. This overhead can significantly decrease performance.

- Limited interoperability with templates

  A `virtual` function cannot be a template. This can become a problem, e.g. if we want to have an overwritable base implementation of the addition between operands of different numeric types.

The *Curiously Recurring Template Pattern* (CRTP) is a C++ programming idiom which can serve as a substitute to polymorphism in many cases while avoiding the above problems. The basic idea of CRTP is to pass the type of the derived class as a template parameter to the base, which can then resolve the function calls already at compile time.

```
template<class Derived>
struct base {
    void foo() {
        // Emulate polymorphism by forwarding calls
        // to base<>::foo() to Derived::foo()
        static_cast<Derived*>(this)->foo();
    }
};

struct derived : base<derived> {
    void foo() {
        // Actual implementation of base<>::foo()...
    }
};
```

We face the exceptional situation that we don't have just one inheritance graph, but rather we have five inheritance graphs which are highly interdependent. For illustration, imagine we have a `tensor_network_base` class and want to derive a `tree_tensor_network_base` class from it. In general, only modifying `tensor_network_base` will not be enough since we might also want to add new features to other classes, e.g. the `vertex_base` class. Therefore, we derive a `tree_vertex_base` class from `vertex_base`. At this point, however, we must have a way to tell the `tensor_network_base` class to use `tree_vertex_base` instead of `vertex_base`, as it is the `tensor_network_base` class which manages the vertices. To be precise, it is actually up to the `vertex_container_base` class to manage the vertices, but for simplicity we ignore that intermediate step.

To resolve this problem, we introduce proxy classes for all five tensor network classes which are templated on a class parameter `NetworkType`. We then have a `tensor_network`

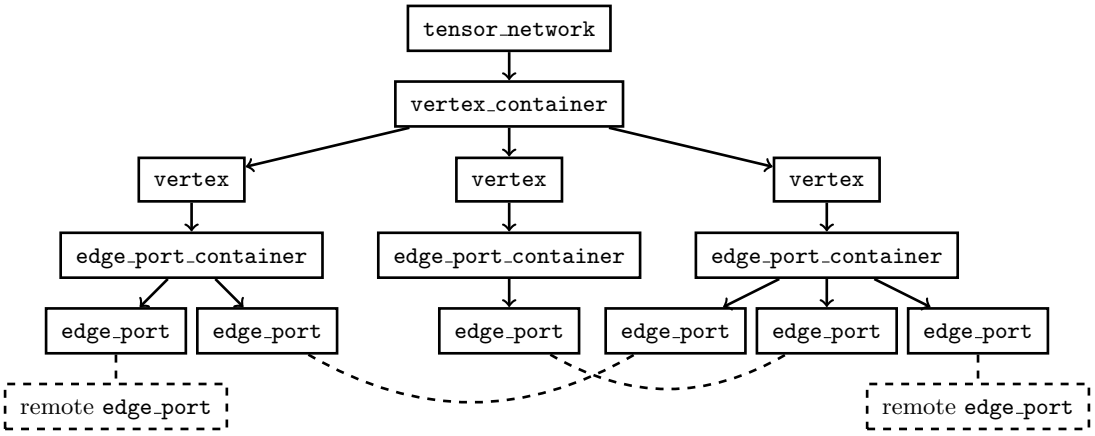


Figure 2.2: Composition of objects of different tensor network classes. Arrows indicate the ownership of the source object by the destination object. The dashed lines symbolize the links between the `edge_ports`.

`<general>` and a `tensor_network<tree>` class and likewise proxy classes for all other tensor network classes. Additionally, all of the above `*_base` get templated on `Network Type` as well. Eventually, we let the proxy classes inherit from the corresponding base classes. So `tensor_network<general>` inherits from `tensor_network_base<general>`, whereas `tensor_network<tree>` inherits from `tree_tensor_network_base<tree>` which in turn inherits from `tensor_network_base<tree>`. An instantiation of `tensor_network_base` therefore knows whether it stores a general network or a tree network and can reference the appropriate vertex class through `vertex<NetworkType>`. See also Figure 2.3 for a schematic representation of the relations described above. Note that our scheme is identical in functionality and very similar in spirit to the CRTP, while the actual implementation differs in that we let the derived classes serve as proxies and forward function calls to these proxies implicitly through inheritance.

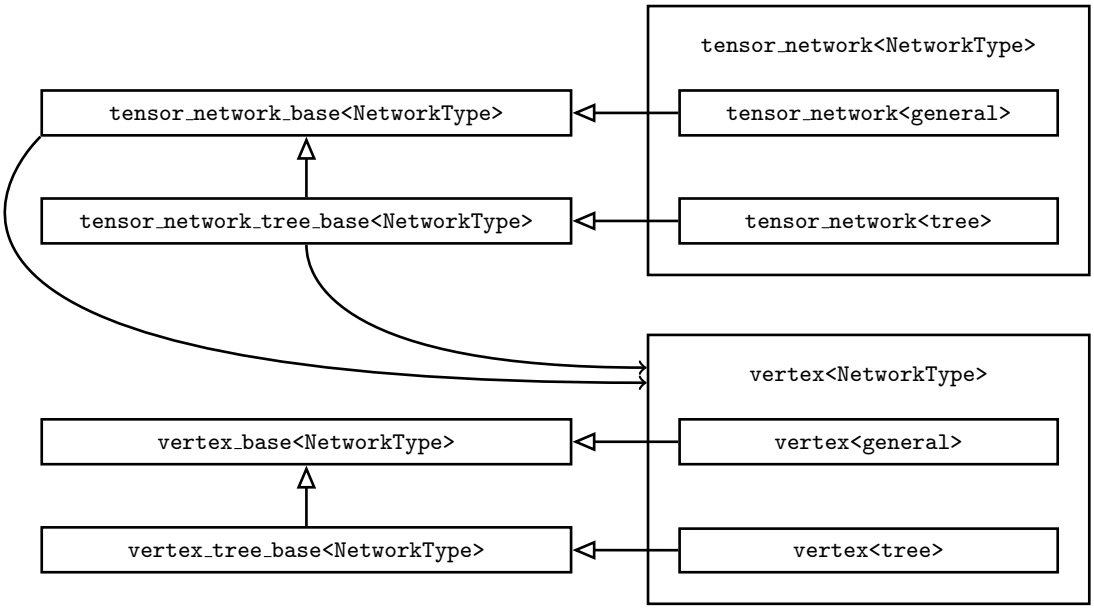


Figure 2.3: Excerpt of the inheritance diagram for the tensor network classes. The arrows always connect classes with the same template parameter `NetworkType`.

3 Performance

3.1 Theoretical Scaling

3.1.1 Introduction

All tree tensor network algorithms naturally deal with one vertex at a time, but they impose different constraints on the order in which the vertices have to be processed. Based on these constraints, we can split the algorithms into two classes:

- Perfectly parallel algorithms

  The vertices can be processed in arbitrary order.
  Examples: addition, truncation (after the gramians have been computed)

- Tree parallel algorithms

  Child vertices have to be processed before the parent vertices (leaves-to-root algorithms) or vice versa (root-to-leaves algorithms).
  Examples: the dot product, orthogonalization, computation of the gramians

For sufficiently large numbers of vertices (i.e. sufficiently fine granularity of the job sizes compared to the overall workload), the runtime of perfectly parallel algorithms obviously scales with $\frac{1}{p}$ where p is the number of processes. On the other hand, for tree parallel algorithms such perfect scaling is not possible. In the remainder of this section, we derive a formula for the optimal speedup achievable with a tree parallel algorithm in a model situation satisfying the following assumptions.

Assumption 1. d denotes the number of leaf vertices, p the number of processors available. We assume $\lfloor \log_2 d \rfloor \geq \lceil \log_2 p \rceil$.

Assumption 2. The tensor represented by the network has exactly d modes. All of them have the same mode size n, and there is exactly one such free mode per leaf vertex.

Assumption 3. All ranks have the same size k.

Assumption 4. The tree is a balanced binary tree, i.e. the longest branch is at most one vertex longer than the shortest one.

Assumption 5. Processing a leaf or root vertex takes no time. Processing an interior vertex takes one time unit.

Assumption 6. Processing a vertex is an atomic operation, i.e. it cannot be split into smaller steps.

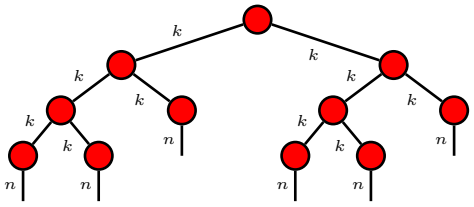


Figure 3.1: A tensor network satisfying the assumptions of Subsection 3.1.1. The edge labels indicate the ranks (k) and mode sizes (n).

A tensor network satisfying all the above assumptions is depicted in Figure 3.1. Interior vertices store third order tensors of size k^3, whereas the leaves and the root store matrices of size nk or k^2, respectively. The relative runtime costs in Assumption 5 are thus approximately correct for large ranks k. Assumption 6 is violated in the current implementation: The processing of a vertex v is split into smaller operations, namely the preparation and consumption of messages sent over the edges incident to v. Unfortunately, this discrepancy between the model and the implementation can produce non-negligible differences in the results one obtains. On the other hand, it is clear that predictions based on the more coarse-grained model give a lower bound on the more fine-grained reality, and we will see in Subsection 3.2.1 that the bound is sharp in most cases. Finally, a model without Assumption 6 is much more difficult to treat, and it is not clear whether such a more complicated model would be worth the effort given that other assumptions are not realistic either.

3.1.2 Optimal Scaling Of Tree Parallel Algorithms

Given Assumptions 1 to 6, we can visualize the execution of a parallel algorithm in a table where the entry in the qth row and ith column denotes which interior vertex is processed by process q in time step i. We call such a table a *vertex schedule*. The overall runtime of the algorithm is given by the rightmost non-empty entry in the vertex schedule. A root-to-leaves algorithm requires that every parent vertex appears to the left of all its child vertices, and a leaves-to-root algorithm likewise requires the opposite order. The following theorem justifies that we merge root-to-leaves and leaves-to-root algorithms in the more general class of tree parallel algorithms:

Theorem 1 (Equivalence of leave-to-root and root-to-leaves algorithms)**.** *The optimal runtime of a leaves-to-root algorithm on p processes is equal to the optimal runtime of a root-to-leaves algorithm on the same number of processes.*

Proof. Let $R(p)$ be the optimal runtime of a root-to-leaves algorithm, and $L(p)$ the optimal runtime of a leaves-to-root algorithm, and assume $R(p) < L(p)$. Then, we can construct a leaves-to-root schedule with runtime $R(p)$ by flipping all rows in the optimal root-to-leaves schedule which contradicts the optimality of $L(p)$. We therefore have $L(p) \leq R(p)$. The similar argument in the opposite direction yields $R(p) \leq L(p)$, thus $L(p) = R(p)$. □

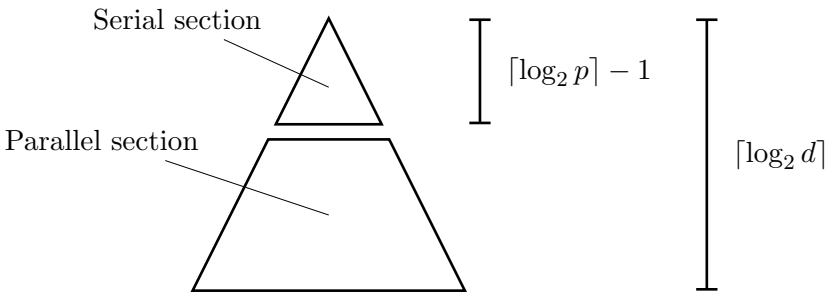


Figure 3.2: Splitting of the tree into serial and parallel sections.

To derive the optimal runtime of a tree parallel algorithm, it is useful to split the tree into a serial section and a parallel section as depicted in Figure 3.2. The serial section contains the vertices on all complete levels with fewer than p vertices, and the parallel section the vertices on the remaining levels. We compute the optimal runtime of a tree parallel algorithm for both sections, and eventually combine the results to the overall runtime.

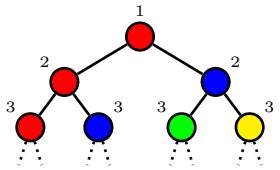


Figure 3.3: Execution history of a tree parallel algorithm in the serial section. Every vertex label indicates the step at which the vertex is processed; the colors distinguish between the processes.

Theorem 2 (Serial section runtime)**.** *The optimal runtime of a tree parallel algorithm in the serial section is given by* $\lceil \log_2 p \rceil - 1$.

Proof. Clearly, we cannot do better than parallelizing the vertex processing level-wise. The runtime is then given by the number of levels in the serial section, which is $\lceil \log_2 p \rceil - 1$. □

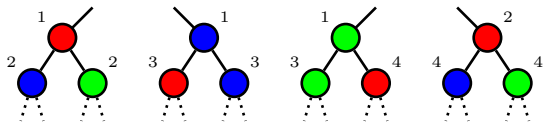


Figure 3.4: Execution history of a tree parallel algorithm in the parallel section. Every vertex label indicates the step at which the vertex is processed; the colors distinguish between the processes.

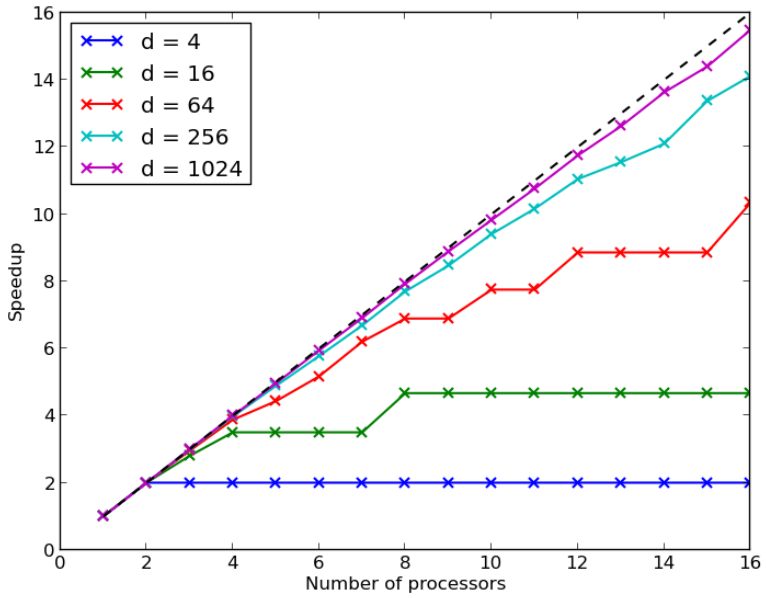


Figure 3.5: Optimal scaling of tree parallel algorithms.

Theorem 3 (Parallel section runtime)**.** *The optimal runtime of a tree parallel algorithm in the parallel section is given by* $\lceil \frac{d-2^{\lceil \log_2 p \rceil}}{p} \rceil$.

Proof. The whole tree contains $d-2$ interior vertices, $2^{\lceil \log_2 p \rceil}-2$ of which are in the serial section. There are thus $d-2^{\lceil \log_2 p \rceil}$ interior vertices to be processed in the parallel section. We see that the above runtime is the same as the runtime of a perfectly parallel algorithm, therefore it is surely a lower bound on the optimum. We prove that this runtime is indeed achievable for a tree parallel algorithm by showing that we can keep all p processes busy for all but the last time steps.

By Theorem 1, it is enough to consider a root-to-leaves algorithm. In the first step, we process p vertices from the first level in the parallel section (note that there have to be at least p vertices in the first level due to the definition of the parallel section). In the second step, we process the remaining $2^{\lceil \log_2 p \rceil}-p$ vertices in the first level, and let the leftover processes work on the vertices from the second level. Since we processed p vertices from the first level in the first step, there are $2p$ vertices ready for processing on the second level, i.e. we can occupy all p processes in the second step. In the third step, we process p vertices from the second level. Finally, the arguments can be iterated until the last step. □

Theorem 4 (Runtime and scaling of tree parallel algorithms)**.** *The optimal runtime of*

a tree parallel algorithm on p processes is

$$T(p) = \lceil \log_2 p \rceil - 1 + \left\lceil \frac{d - 2^{\lceil \log_2 p \rceil}}{p} \right\rceil = O\left(\log_2 p + \frac{d}{p}\right)$$

The best possible speedup obtainable with a tree parallel algorithm is

$$\frac{T(1)}{T(p)} = \frac{d-2}{T(p)} = O\left(\frac{d}{\log_2 p + \frac{d}{p}}\right)$$

Proof. Consider a root-to-leaves algorithm. Clearly, it takes the algorithm at least $\lceil \log_2 p \rceil - 1$ steps to finish a vertex on the bottom level of the serial section. From the proof of Theorem 2 we conclude that all vertices in the serial section can be processed within that time but no vertex on the last level can be processed before the last time step. Therefore, by the time we start processing the first vertex in the parallel section, we can directly use the schedule of Theorem 3, and thus the optimal overall runtime is the sum of the optimal runtimes of the two sections. □

3.1.3 Optimized Vertex Distribution

Since sending data between processes is expensive and the vertices can be up to k^3 in size, we distribute the vertices to the processes once and then stay with that distribution throughout the computations. This raises the need to to find a distribution which allows all algorithms to be executed with reasonable performance. In particular, such a distribution should satisfy the following conditions.

Condition 1. The interior vertices should be distributed evenly over the processes (necessary condition for optimal scaling of perfectly parallel algorithms).

Condition 2. The interior vertices in the parallel section should be distributed evenly over the processes (necessary condition for optimal scaling of tree parallel algorithms).

Condition 3. Each parent vertex should have at least one child vertex on the same process. This allows to benefit from the asynchronous communication between a parent and its children in the presence of latency.

By evenly distributed, we mean that the process owning the most interior vertices owns at most one more than the process owning the least.

It turns out that the above conditions cannot be satisfied all at the same time: Conditions 1 and 2 together imply that the interior vertices in the serial section also have to be evenly distributed up to a maximal imbalance of two interior vertices. Since there are $2^{\lceil \log_2 p \rceil} - 2 < 2p - 2$ interior vertices in the serial section, this means that a process can receive at most 3 interior vertices from the serial section. But, to satisfy Condition 3, we need to place at least $\lceil \log_2 p \rceil - 2$ (depth of the serial section minus root vertex) interior vertices from the serial section onto the same process. A similar argument shows that in fact already Conditions 1 and 3 are mutually contradicting.

Violating Condition 1 increases the runtime of perfectly parallel algorithms to the one of tree parallel algorithms, i.e. it introduces an additional $O(\log_2 p)$ term. On the other hand, violating Condition 3 means that on each level we may have to wait some additional time for the inter-process communication to occur, thus on the whole tree we have again an $O(\log_2 p)$ term. We therefore expect that it doesn't make much of a difference whether we choose to satisfy Conditions 1 and 2 or 2 and 3. We devised and implemented an algorithm which given a d and p constructs a balanced binary tree and a vertex distribution such that 2 and 3 are satisfied. An example vertex distribution produced by it is depicted in Figure 3.6.

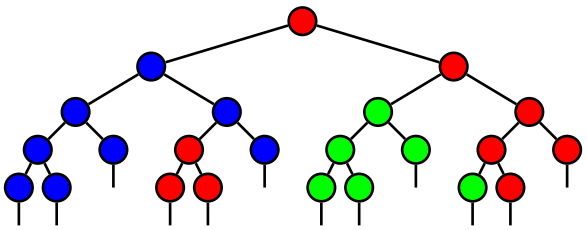


Figure 3.6: Example vertex distribution for $d = 12$ and $p = 3$.

3.2 Experimental Results

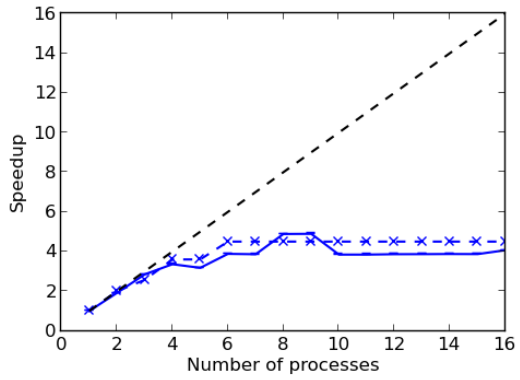
All benchmarks were carried out on four Quad-Core AMD Opteron$^{\text{TM}}$ 8356 processors (2.3 GHz).

3.2.1 Empirical Scaling

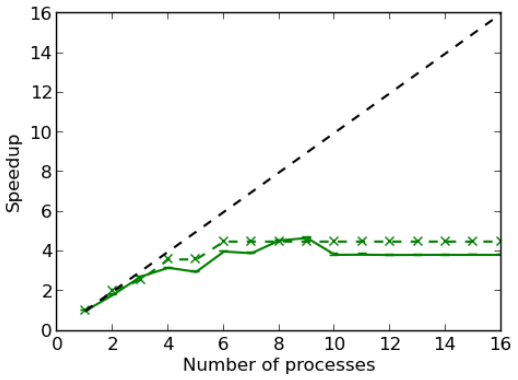
To verify the theoretical predictions from the previous section, we show the scaling of the addition and dot product in Figure 3.7. We would like to make the following comments on the obtained results.

- Even though the addition is a perfectly parallel algorithm, the fact that we distribute the vertices in a way optimized for tree parallel algorithms reduces its scaling to the one of tree parallel algorithms (see the discussion in 3.1.3). This justifies that we compare the scaling of the addition with the optimal scaling of tree parallel algorithms.

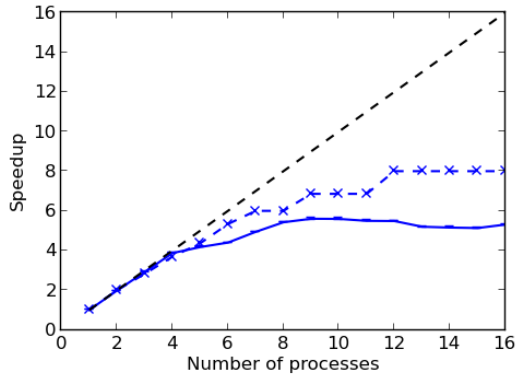
- For high processor counts, the algorithms scale notably worse than the theoretical predictions. In the case of the addition, this can be justified by noting that tensor network addition is a strongly memory-bound algorithm, and memory bandwidth does not scale linearly with the number of processors. In the case of the dot product, the suboptimal scaling is probably due to the fact that we use Boost.MPI [9] inefficiently: Currently, each send and receive does an unnecessary local copy of all the transmitted data. Benchmarks showed that these copies reduce the communication performance by up to a factor 10. We therefore aim to resolve this problem in the future.



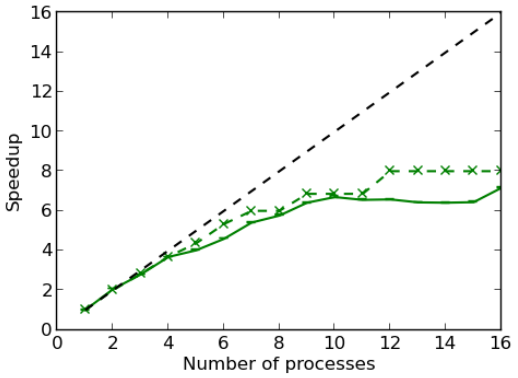
(a) Addition, $d = 20$



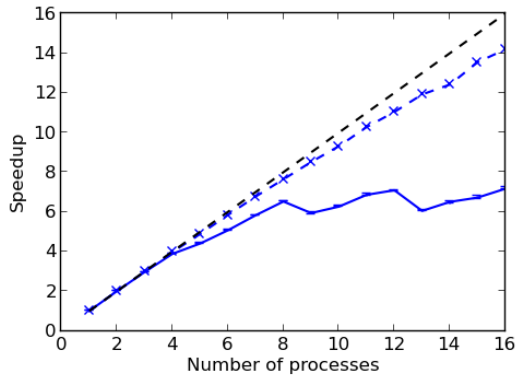
(b) Dot product, $d = 20$



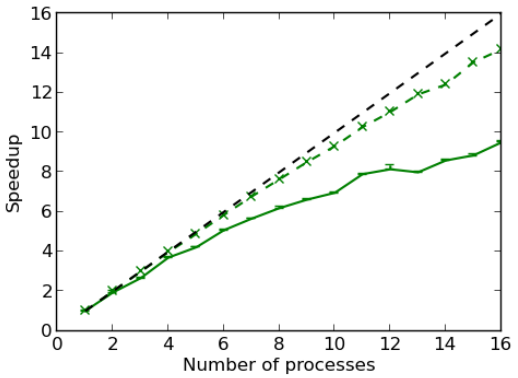
(c) Addition, $d = 50$



(d) Dot product, $d = 50$

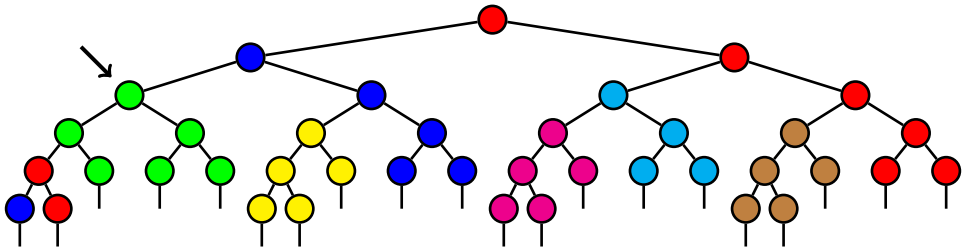


(e) Addition, $d = 300$

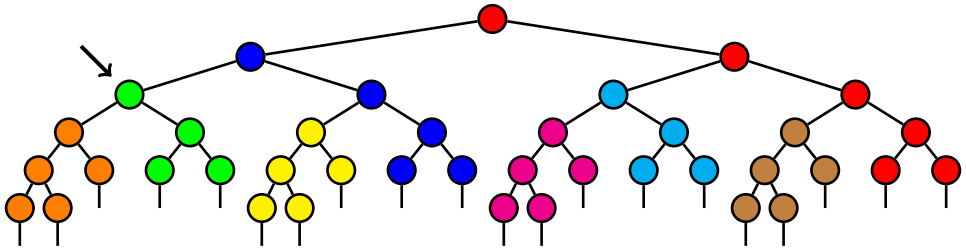


(f) Dot product, $d = 300$

Figure 3.7: Strong scaling of addition (left) and dot product (right) on binary trees with d leaves, uniform mode sizes and ranks $n = k = 64$ and random initial data. The dashed lines denote the optimal scaling of a tree parallel algorithm as stated in Theorem 4. The solid lines show the median speedups over 10 runs, and the error bars the 20% and 80% quantiles.



(a) Vertex distribution for $d = 20$ and $p = 7$.



(b) Vertex distribution for $d = 20$ and $p = 8$.

Figure 3.8

- For $d = 20$ and $p = 8, 9$, the speedup is larger than what is predicted by theory. To explain this, consider the corresponding vertex distributions shown in Figure 3.8. Even though the outcome is similar for both the addition and dot product, we have to look at the two cases separately:

  Addition: For $p = 7$, the process with the most interior vertices and thus the bottleneck is the red one with four interior vertices. In the case when $p = 8$, the red process has one interior vertex less which makes the addition run faster.

  Dot product: The mismatch between theory and experiment is due to the wrong model assumption that the vertex processing is an atomic operation (see Assumption 6 in Subsection 3.1.1). In the case when $p = 8$, the model assumes that the orange process first has to process its two interior vertices before the green one can start processing the vertex indicated by the arrow, which gives a total of three time steps for this subtree. What actually happens in the implementation is that the green process consumes the message from the right son of the marked vertex already while the orange process is still working. Once the orange process is done, the green process then only has to consume the message from the left son and prepare the message to the parent, which is one consume operation less than what the model assumes. For the dot product, it is reasonable to expect that the preparation and consumption of messages each take $\frac{1}{3}$ time units. The indicated subtree is then processed in only $2 + \frac{2}{3}$ time units, and the overall speedup is $\frac{18}{3+\frac{2}{3}} \approx 4.91$, while the model predicts a speedup of $\frac{18}{4} = 4.5$.

  For $p = 7$, a similar effect does not occur because the green process has enough

work to do on its own and never waits for the message from the red process. Therefore, in this case the model prediction is exact.

3.2.2 Sine Series Example

We also test the performance of our code in the more realistic example of evaluating the truncated sine series of a d-dimensional analytic function on $[0,1]^d$,

$$f(x_1,\ldots,x_d) = \sum_{(i_1,\ldots,i_d)\in\Delta(d,m)} \alpha(i_1,\ldots,i_d) \prod_{k=1}^{d} \sin(2\pi i_k x_k)$$

We set $\alpha(i_1,\ldots,i_d) = b^{\,d-\sum_{k=1}^{d} i_k}\,\omega(i_1,\ldots,i_d)$ with some fixed $b > 1$ and $\omega(i_1,\ldots,i_d)$ uniformly distributed in $[-1,1]$. $\Delta(d,m)$ denotes the d-dimensional discrete simplex

$$\Delta(d,m) := \left\{ (i_1,\ldots,i_d) \in \mathbb{Z}^{\geq 1} \mid \sum_{k=1}^{d} i_k \leq m \right\}$$

One can show that $|\Delta(d,m)| = \binom{m}{m-d}$. We choose m such that $\Delta(d,m)$ contains exactly the points $(i_1,\ldots,i_d)$ for which $b^{\,d-\sum_{k=1}^{d} i_k} > \frac{\varepsilon}{b}$ with some target accuracy $\varepsilon > 0$, i.e. $m = d - \lfloor \log_b(\varepsilon) \rfloor$ ($\lfloor\cdot\rfloor$ denotes rounding towards $-\infty$). For $d = 1$, this bounds the L^2-norm of the dropped terms in the infinite sine series by $\frac{\varepsilon}{\sqrt{2\,(1-b^{-2})}}$. If we use the same definition of m in higher dimensions, the prefactor of ε in the error due to the truncation of the series grows rapidly with d, so in principle we should compensate for that by increasing m. On the other hand, the number of terms to sum grows as $\mathcal{O}\left(\frac{m^d}{d!}\right)$ which quickly renders the summation effectively impossible. We therefore use the same definition of m for all d, even though this means that ε does not correspond to a bound on the series truncation error.

We represent a d-dimensional function $g(x_1,\ldots,x_d)$ by its point values on an equidistant tensor-product mesh on the unit hypercube $[0,1)^d$ using 2^l grid points in each dimension, $l \in \mathbb{Z}^{\geq 0}$. This transforms $g(x_1,\ldots,x_d)$ into a tensor $G(i_1,\ldots,i_d) := g(x_{i_1},\ldots,x_{i_d})$ with $x_i := 2^{-l}\,i$ and $i_k \in \{0,\ldots,2^l-1\}$. We further quantize the *physical* dimensions i_k, i.e. we interpret each i_k, $k = 1,\ldots,d$, as a long index made up of the so-called *virtual* dimensions $j_{k,\kappa} \in \{0,1\}$, $\kappa = 1,\ldots,l$. The idea of quantization in the context of tensor-structured representations was introducted in [10, 11, 12, 13]. G then becomes a tensor with $d \times l$ modes defined by

$$G(j_{1,1},\ldots,j_{1,l}\,;\,\ldots\,;\,j_{d,1},\ldots,j_{d,l}) := g\left(\sum_{\kappa=1}^{l} 2^{\kappa-l-1} j_{1,\kappa},\ldots,\sum_{\kappa=1}^{l} 2^{\kappa-l-1} j_{d,\kappa}\right)$$

This G is stored in a tensor network based on a balanced binary tree with one leaf for each $j_{k,\kappa}$. If we traverse through the leaves from left to right, we encounter the modes in the following order: $j_{1,1}, j_{1,2}, \ldots, j_{1,l}\,;\, j_{2,1}, j_{2,2}, \ldots, j_{2,l}\,;\,\ldots\,;\, j_{d,1}, j_{d,2}, \ldots, j_{d,l}$. Throughout this example, we use $l = 10$. If $g(x_1,\ldots,x_d) = \prod_{k=1}^{d} \sin(2\pi i_k x_k)$, the ranks

of G in this format are bounded by 2 iff d is a power of 2 and by 4 for general d. This result (namely, the former bound) was shown in [14] for TT networks and is easy to adapt to the HTR networks considered here.

If we naively add two tensor networks, the ranks of the resulting tensor network are the sums of the ranks of the two summands. In our case, this means that the ranks of F (i.e. the tensor network representing f) grow linearly in the number of terms which we add. Since memory consumption and the runtime of the addition scale cubically in the rank sizes, we truncate F after every 10 terms added to it with a relative error tolerance of $10\,\varepsilon/\binom{m}{m-d}$. Once we summed up all terms, we truncate F once more with a relative accuracy of ε. For the tree tensor network truncation to be numerically stable, the square of the relative error tolerance has to be larger than the machine precision `eps`. The commonly used C++ `double` floating point type is 8 byte wide and has `eps` $\approx 2.2\times10^{-16}$ on the machine and compiler we use. We therefore cannot prescribe tolerances $< 10^{-8}$ and, to be on the safe side, we had better choose the tolerances one or two order of magnitudes above this limit. As we want to truncate with tolerances $10\,\varepsilon/\binom{m}{m-d}$ ranging down to $10^{-3}\,\varepsilon$, this condition can be rather restrictive on ε. We therefore run the entire benchmark using the 16 byte `long double` type with `eps` $\approx 1.1 \times 10^{-19}$.

In Table 3.1, we report on the outcome of the above procedure. We observe that the ranks are much smaller than the theoretical bound of $2\,|\Delta(d,m)|$, even though they do grow monotonically with $|\Delta(d,m)|$ (see Figure 3.9). Regarding the performance, we observe similar patterns as in Subsection 3.2.1. In order for the problem to be reasonably parallelizable, we need sufficiently high dimensionality d, and the scaling gets worse for larger processor counts p. In addition, there is also one new issue: contrasting to the model assumptions from Section 3.1, the ranks increase towards the top of the tree, see Figure 3.10. This further limits parallelizability as the vertex processing can be distributed over fewer processes the closer we get to the root.

d	b	ε	$\lvert\Delta(d,m)\rvert$	k_{max}	k_{eff}	$T(1)$ $[s]$	$T(2)$ $[s]$	$S(2)$	$T(4)$ $[s]$	$S(4)$	$T(8)$ $[s]$	$S(8)$	$T(16)$ $[s]$	$S(16)$
2	2	10^{-4}	120	22	5.58	16.17	9.38	1.7	8.52	1.9	8.19	2.0	9.67	1.7
2	4	10^{-4}	36	11	4.29	2.55	2.36	1.1	2.17	1.2	2.10	1.2	2.17	1.2
2	10	10^{-4}	15	7	3.58	0.72	1.46	0.5	1.39	0.5	1.38	0.5	1.42	0.5
2	2	10^{-6}	231	31	7.29	48.00	25.64	1.9	23.46	2.0	22.95	2.1	23.14	2.1
2	4	10^{-6}	66	17	5.14	6.95	4.63	1.5	4.14	1.7	4.02	1.7	4.09	1.7
2	10	10^{-6}	28	10	4.31	1.92	2.04	0.9	1.88	1.0	1.83	1.0	1.88	1.0
4	2	10^{-4}	3060	49	9.41	2478.23	1287.25	1.9	989.35	2.5	962.38	2.6	955.03	2.6
4	4	10^{-4}	330	15	5.16	84.78	45.09	1.9	27.18	3.1	24.98	3.4	24.14	3.5
4	10	10^{-4}	70	9	3.86	10.59	6.55	1.6	4.07	2.6	3.65	2.9	3.60	2.9
4	4	10^{-6}	1001	26	6.96	414.94	215.65	1.9	141.79	2.9	134.99	3.1	132.34	3.1
4	10	10^{-6}	210	13	5.00	48.13	26.38	1.8	15.46	3.1	14.03	3.4	13.46	3.6
8	4	10^{-4}	6435	69	11.22	12880.05	7005.80	1.8	5922.95	2.2	5486.87	2.3	5423.43	2.4
8	10	10^{-4}	495	17	4.98	203.85	108.15	1.9	62.56	3.3	42.38	4.8	39.20	5.2
8	10	10^{-6}	3003	45	8.60	3108.04	1668.23	1.9	1248.24	2.5	1080.97	2.9	1060.46	2.9
16	10	10^{-4}	4845	53	8.82	8531.14	4660.81	1.8	3393.91	2.5	2909.77	2.9	2689.30	3.2

Table 3.1: Ranks and runtimes of the sine series example. k_{max} denotes the largest rank in the final approximation to F, k_{eff} the effective rank computed as $\sqrt[3]{\frac{s}{d-2}}$ where s is the total number of coefficients stored in all interior vertices of the tree. $T(p)$ is the runtime on p processors, $S(p) := \frac{T(1)}{T(p)}$ the speedup compared to the single core runtime.

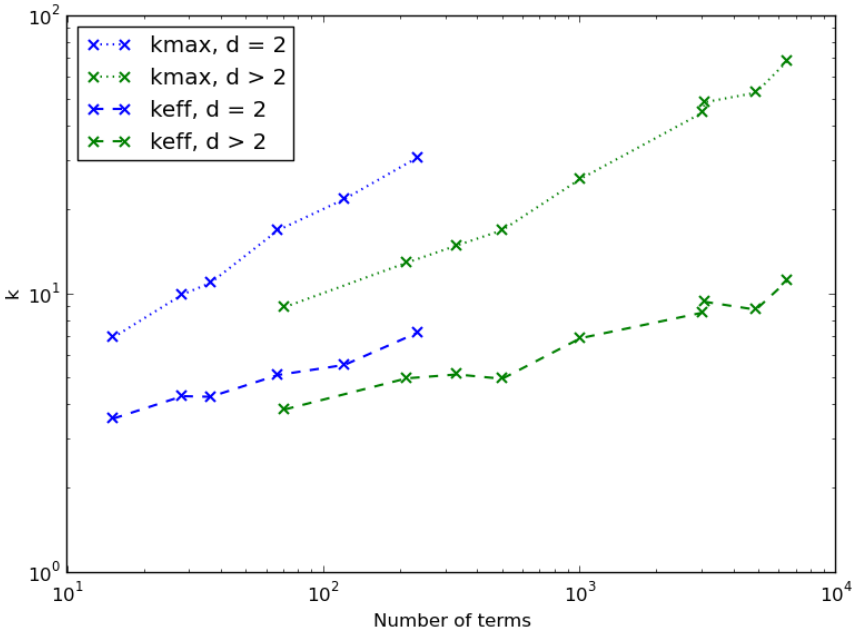
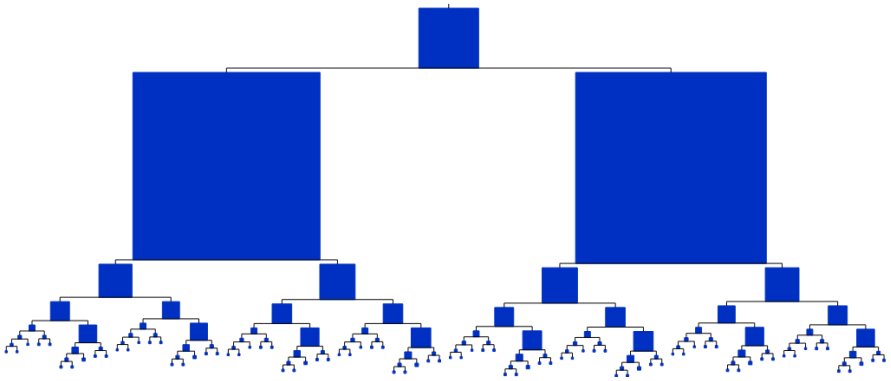
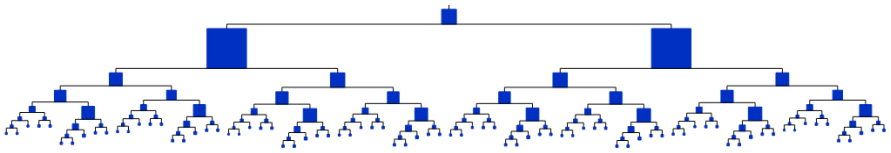


Figure 3.9: Ranks as a function of $|\Delta(d, m)|$



(a) $d = 8$, $b = 4$, $\varepsilon = 10^{-4}$



(b) $d = 8$, $b = 10$, $\varepsilon = 10^{-4}$

Figure 3.10: Illustration of the vertex tensor sizes for two cases from Table 3.1. The area of each square is proportional to the number of coefficients stored in the corresponding vertex.

4 Conclusion

We devised and implemented a framework for distributed-memory parallel computations on tensor networks. Among the algorithms provided so far are the copying and addition of general networks, and the dot product, orthogonalization and truncation of tree-structured networks.

We also studied the scaling of tensor network algorithms from the theoretical point of view as well as experimentally. We observed that some tensor formats like the TT one cannot be parallelized due to their inherently linear structure. On the other hand, networks based on balanced trees like the HTR are good candidates for parallelization. We have seen that even though tree network algorithms do not scale perfectly, their scaling approaches the perfect one for large dimensions d compared to the processor count p. Currently, the scaling of our implementation falls behind the theoretical predictions, but we hope to improve on this in future.

Bibliography

[1] Wolfgang Hackbusch. *Tensor Spaces and Numerical Tensor Calculus*. Vol. 42. Springer Series in Computational Mathematics. Springer, 2012.

[2] I. V. Oseledets and E. E. Tyrtyshnikov. 'Breaking the curse of dimensionality, or how to use SVD in many dimensions'. *SIAM Journal on Scientific Computing* 31.5 (Oct. 2009), pp. 3744–3759.

[3] I. V. Oseledets. 'Tensor-Train Decomposition'. *SIAM Journal on Scientific Computing* 33.5 (2011), 2295–2317.

[4] W. Hackbusch and S. Kühn. 'A New Scheme for the Tensor Representation'. *Journal of Fourier Analysis and Applications* 15.5 (2009). 10.1007/s00041-009-9094-9, pp. 706–722.

[5] L. Grasedyck. 'Hierarchical Singular Value Decomposition of Tensors'. *SIAM Journal on Matrix Analysis and Applications* 31.4 (2010), 2029–2054.

[6] Lars Grasedyck, Daniel Kressner and Christine Tobler. *A literature survey of low-rank tensor approximation techniques*. arXiv preprint 1302.7121. Feb. 2013.

[7] *The Eigen Linear Algebra Library*. URL: `http://eigen.tuxfamily.org` (visited on 14/04/2014).

[8] N. Edmonds, D. Gregor and A. Lumsdaine. *Parallel Boost Graph Library*. URL: `http://www.boost.org/doc/libs/1_55_0/libs/graph_parallel/doc/html/index.html` (visited on 29/03/2014).

[9] D. Gregor and M. Troyer. *Boost.MPI*. URL: `http://www.boost.org/doc/libs/1_55_0/doc/html/mpi.html` (visited on 27/04/2014).

[10] E. E. Tyrtyshnikov. 'Tensor approximations of matrices generated by asymptotically smooth functions'. *Sbornik: Mathematics* 194.5 (2003), pp. 941–954.

[11] I. Oseledets. 'Approximation of matrices with logarithmic number of parameters'. *Doklady Mathematics* 80.2 (Apr. 2009), pp. 653–654.

[12] Boris N. Khoromskij. $\mathcal{O}(d \log N)$*-Quantics Approximation of* N*-d Tensors in High-Dimensional Numerical Modeling*. Preprint 55. Max-Planck-Institut für Mathematik in den Naturwissenschaften, Sept. 2009, pp. 1–21.

[13] I. V. Oseledets. 'Approximation of $2^d \times 2^d$ matrices using tensor decomposition'. *SIAM Journal on Matrix Analysis and Applications* 31.4 (2010), pp. 2130–2145.

[14] I. V. Oseledets. 'Constructive representation of functions in tensor formats'. *Constructive Approximation* 37 (1 2013), pp. 1–18.