ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Seminar for Applied Mathematics SAM

Parallel ALS Algorithm for Solving Linear Systems in the Hierarchical Tucker Representation

S. Etter

Research Report No. 2015-25

August 2015
Latest revision: May 2016

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

PARALLEL ALS ALGORITHM FOR SOLVING LINEAR SYSTEMS IN THE HIERARCHICAL TUCKER REPRESENTATION

SIMON ETTER

Abstract. Tensor network formats are an efficient tool for numerical computations in many dimensions, yet even this tool often becomes too time- and memory-consuming for a single compute node when applied to problems of scientific interest. Intending to overcome such limitations, we present and analyse a parallelisation scheme for algorithms based on the *Hierarchical Tucker Representation* which distributes the network vertices and their associated computations over a set of distributed-memory processors. We then propose a modified version of the *alternating least squares* (ALS) algorithm for solving linear systems amenable to parallelisation according to the aforementioned scheme and highlight technical considerations important for obtaining an efficient and stable implementation. Our numerical experiments support the theoretical assertion that the parallel scaling of this algorithm is only constrained by the dimensionality and the rank uniformity of the targeted problem.

1. Introduction. Computations in high dimensions are notoriously difficult due to the *curse of dimensionality*: if an algorithm requires n data points to solve a problem in one dimension, then solving the analogous problem in d dimensions typically requires n^d data points which becomes prohibitive very quickly. *Tensor network ansätze* like the *hierarchical Tucker representation* (HTR) from [11, 7] or its simpler special case, the *tensor train* (TT) format from [24, 22], avoid this curse by cleverly exploiting the structure in the data for compression such that the costs of working with n^d data points scale only with n times d times some low-order polynomial in the *rank parameter* r. Heuristically, this parameter measures the "structuredness" of the data and can be shown to be independent of d for many important special cases, see e.g. [8, 15, 23]. We further refer to [10] for a textbook and to [16, 9] for literature surveys regarding tensor network formats.

In the present paper, we consider high-dimensional linear systems of equations $Ax = b$ where the unknowns x have n^d entries such that they can only be feasibly handled in compressed tensor network form. The *alternating least squares* (ALS, also known as one-site DMRG) algorithm from [13, 19] and its various extensions like the *density matrix renormalisation group* (DMRG) algorithm from [13, 19] or the *ALS + steepest descent* (ALS(SD)) and *alternating minimal energy* (AMEn) algorithms from [4] are among the most effective to solve problems of this type, yet even these computational tools require parallelisation when applied to the large-scale problems from science and engineering. This paper presents a novel parallelisation scheme for such ALS-type algorithms which, to the author's knowledge, is the first one to have a serial fraction growing sublinearly in the problem size parameter d, meaning that its parallel scalability grows with increasing problem dimensionality.

This favourable property is brought about by basing our algorithm on the HTR which, as we motivate next on a fairly abstract level, is intrinsically better suited for parallelisation than the TT format. Any non-trivial algorithm, in particular the solution of linear systems, requires gathering some information from all vertices of the network, and this step can only be carried out efficiently if the information is passed on from one vertex to its neighbour like the baton in a relay race. Examples of such information gathering steps are the orthogonalisation and, in case of the HTR, computation of the Gramians for truncation (see [22, 7]), and the computation of the projected operators and right-hand sides for the ALS algorithm. In the TT case, the longest distance between two vertices is $\mathcal{O}(d)$ and it is therefore not possible to

reduce the runtime of the information gathering step below $\mathcal{O}(d)$ through parallelisation. In contrast, the longest distance in the HTR based on a balanced mode tree is only $\mathcal{O}(\log(d))$ and all basic algorithms (addition, dot product, orthogonalisation and truncation) achieve the resulting lower bound of $\mathcal{O}(\log(d))$ on the parallel runtime out of the box.

Nevertheless, it is possible to parallelise ALS-type algorithms based on the TT format to some extent, as has been shown in [26]. In the mental picture given above, the parallelisation scheme proposed there exploits that information needs to be exchanged repeatedly in ALS-type algorithms, and by pipelining up to $\mathcal{O}(d)$ such exchange rounds one obtains an algorithm scaling up to $\mathcal{O}(d)$ processors after some initial phase. The fundamental problem of the TT format remains, however, and expresses itself in that the first exchange round, requiring $\mathcal{O}(d)$ computational effort, cannot be parallelised beyond two processors, namely one for each end of the TT chain.

The remainder of this paper is organised as follows. Section 2 introduces the notation necessary for our presentation, and in section 3 we discuss the parallel implementation and analyse the parallel scaling of a fairly general class of HTR algorithms. Section 4 reviews the serial HTR ALS algorithm, highlights the changes required to make this algorithm amenable to the above-mentioned parallelisation scheme and presents numerical experiments verifying the scaling behaviour predicted in section 3.

2. Notation. To the author's knowledge, this is the first manuscript to give a detailed account of the HTR-based ALS algorithm. It should therefore come as no surprise that this endeavour requires a novel notation which will be developed in the following two subsections. Since the result will be fairly remote from the notation found elsewhere, we discuss the advantages of our approach in subsection 2.3.

2.1. Tensors. The definition of tensors used in this paper is based on a generalised concept of tuples obtained as follows.

DEFINITION 1 (Tuple). *Let D be an arbitrary set, and $(A_k)_{k\in D}$ a family of sets parametrised by the elements of D. A* tuple *is a function $t : D \to \bigcup_{k\in D} A_k$, $k \mapsto t_k$ such that $t_k \in A_k$ for all $k \in D$.*

This is a proper generalisation since the more common definition of tuples as ordered sets is retained as the special case $D := \{1, \dots, n\}$ for some $n \in \mathbb{N}$. We denote the set of such tuples, i.e. the Cartesian product of the A_k, by $\bigtimes_{k\in D} A_k$ and define $A^D := \bigtimes_{k\in D} A_k$ for the case $A_k = A$ for all $k \in D$. Two tuples $t^{(1)} \in \bigtimes_{k\in D^{(1)}} A_k$, $t^{(2)} \in \bigtimes_{k\in D^{(2)}} A_k$ with two disjoint sets $D^{(1)}$, $D^{(2)}$ can be combined into a new tuple $t \in \bigtimes_{k\in D^{(1)}\cup D^{(2)}} A_k$ by writing $t := t^{(1)} \times t^{(2)}$.

For our purposes, a tensor is a composition of tuples in the above sense.

DEFINITION 2 (Tensor). *Let D be an arbitrary set of* modes *(also known as* dimensions *or* directions*) with corresponding mode sizes $n_k \in \mathbb{N}$ and index sets $[n_k] := \{0, \dots, n_k - 1\}$ for all $k \in D$. A* tensor *is an element from the Cartesian product $\mathbb{K}^{\bigtimes_{k\in D}[n_k]}$, i.e. it is a tuple with elements from $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$ indexed by tuples from $\bigtimes_{k\in D}[n_k]$ which are themselves indexed by $k \in D$.*

We introduce the following conventions.

- We assume every mode k to have an implicitly defined mode size n_k such that mentioning it in the definition of a tensor space becomes redundant. We therefore abbreviate $\mathbb{K}^{\bigtimes_{k\in D}[n_k]}$ to $\mathbb{K}(D)$.
- In contrast to all other tuples, we subscript the index tuples $i_D \in \bigtimes_{k\in D}[n_k]$

with their domain of definition D because this in turn allows us to define that an index i_D shall always be taken from $\bigtimes_{k\in D}[n_k]$ even if we do not explicitly introduce i_D as such.
- We write $x(i_D)$ instead of x_{i_D} to denote the evaluation of a tensor $x \in \mathbb{K}(D)$ at i_D for better readability.

The addition and scalar multiplication of tensors is defined element-wise,

$$(x+y)\,(i_D) := x\,(i_D) + y\,(i_D)\,, \quad (\alpha x)\,(i_D) := \alpha\, x\,(i_D)$$

for all $x, y \in \mathbb{K}(D)$ and $\alpha \in \mathbb{K}$. The inner product and norm on $\mathbb{K}(D)$ are the standard Euclidean inner product and norm

$$(x,y) := \sum_{i_D} \overline{x(i_D)}\, y(i_D), \quad \|x\| := \sqrt{(x,x)},$$

where $\overline{z}$ denotes complex conjugation if $z \in \mathbb{C}$ and is to be ignored for $z \in \mathbb{R}$.

The *mode product* defined next is the straightforward generalisation of the matrix product for tensors.

Definition 3 (Mode Product). *Let $x \in \mathbb{K}(M \cup K)$, $y \in \mathbb{K}(K \cup N)$ be two tensors such that M, K and N are pairwise disjoint. The expression xy defines a new tensor $z \in \mathbb{K}(M \cup N)$ whose entries are given by*

$$z\,(i_M \times i_N) := \sum_{i_K} x\,(i_M \times i_K)\; y\,(i_K \times i_N)\,.$$

K may also be empty in which case the mode product becomes the tensor product [10, §1.1.1]. The same operation has already been introduced in [1] under the name "tensor-times-tensor" (`ttt`) product. We sometimes call the mode product a *tensor contraction* to emphasise its property of merging several tensors into a single one.

If we interpret a square matrix as a linear operator on $\mathbb{K}(\{k\})$, it has two modes associated with the mode symbol k, namely one which is multiplied with the k-mode of the input vector and one which yields the mode of the output vector. We incorporate this into our notation as follows. Given some mode symbol k, we introduce two new mode symbols $\mathrm{R}(k)$ and $\mathrm{C}(k)$ with $n_{\mathrm{R}(k)} := n_{\mathrm{C}(k)} := n_k$ called *row* and *column mode* of k, respectively, and further define $D^2 := \mathrm{R}(D) \cup \mathrm{C}(D)$. Multiplication with row/column modes follows special rules which generalise the rules of the matrix product: a column mode $\mathrm{C}(k)$ is only multiplied with a k- or $\mathrm{R}(k)$-mode appearing to the right of the tensor carrying the $\mathrm{C}(k)$-mode, and similarly, a row mode $\mathrm{R}(k)$ is only multiplied with a k- or $\mathrm{C}(k)$-mode appearing to the left. If in the resulting tensor there is only either a row mode $\mathrm{R}(k)$ or a column mode $\mathrm{C}(k)$ left, we rename it to k. The simple matrix-vector product Ax with $A \in \mathbb{K}(\{k\}^2)$ and $x \in \mathbb{K}(\{k\})$ is thus to be read as follows. Because x stands to the right of A, the k-mode of x is multiplied with the $\mathrm{C}(k)$-mode of A, yielding an intermediate result in $\mathbb{K}(\{\mathrm{R}(k)\})$ whose $\mathrm{R}(k)$-mode is then renamed to simply k since it appears without an accompanying $\mathrm{C}(k)$-mode. We thus get a final result in $\mathbb{K}(\{k\})$ as expected. Note that the stripping of row/column modes only takes place if the corresponding column/row mode has been multiplied over. For example, the mode product of $x \in \mathbb{K}(\{\mathrm{R}(k)\})$ and $y \in \mathbb{K}(\{\mathrm{C}(\ell)\})$ with two distinct modes k, ℓ lies in $\mathbb{K}(\{\mathrm{R}(k), \mathrm{C}(\ell)\})$, not in $\mathbb{K}(\{k, \ell\})$.

Given a tensor $x \in \mathbb{K}(D)$ and two modes $k \in D$, $k' \notin D$, we let $x^{(k \to k')}$ denote the tensor in $\mathbb{K}((D \setminus \{k\}) \cup \{k'\})$ obtained after relabelling the k-mode to k'. Most commonly, we want to *tag* a mode k with a function-like symbol f (e.g. the

row/column mode tags R and C), a case for which we introduce the abbreviation $x^{(\mathrm{f}(k))} := x^{(k\to\mathrm{f}(k))}$. The main application of this notation is to change which modes are multiplied. For example, we must write $x^{(\mathrm{R}(k))}\, y^{(\mathrm{C}(k))} \in \mathbb{K}(\{k\}^2)$ to denote the outer product of two vectors $x, y \in \mathbb{K}(\{k\})$, whereas $xy \in \mathbb{K}$ yields their inner product up to conjugation of x in the complex case.

We conclude this subsection with a number of technical remarks and definitions.

Remark 4 (Mode Product Properties.). *The mode product is*

- *distributive over tensor addition.*
- *commutative unless one of the multiplied modes is a row or column mode.*
- *associative as long as the sets of multiplied modes are disjoint.*

An example where the mode product is not associative is given by the product of three tensors $x, y, z \in \mathbb{K}(D)$. Depending on how we put the parentheses, we get three different results $(yz)x \in \operatorname{span}\{x\}$, $(xz)y \in \operatorname{span}\{y\}$ or $(xy)z \in \operatorname{span}\{z\}$.

Definition 5 (Identity Tensor). *Let D be some mode set. The* identity tensor *$\mathbb{I}_D$ is the tensor in $\mathbb{K}(D^2)$ such that $\mathbb{I}_D\, x = x$ for all $x \in \mathbb{K}(D)$.*

Definition 6 ((Conjugate) Transposed Tensor). *Let $A \in \mathbb{K}(D \cup \mathrm{R}(E) \cup \mathrm{C}(F))$ be a tensor with some (potentially empty) row and column mode sets $\mathrm{R}(E)$, $\mathrm{C}(F)$. We define*

$$A^T := A^{(\mathrm{R}(E)\to\mathrm{C}(E),\, \mathrm{C}(F)\to\mathrm{R}(F))}, \qquad A^* := \overline{A^T}.$$

Definition 7 (Inverse Tensor). *Let $A \in \mathbb{K}(D^2)$ be an invertible tensor operator. The* inverse tensor *A^{-1} is the unique tensor in $\mathbb{K}(D^2)$ such that $AA^{-1} = \mathbb{I}_D$.*

Definition 8 (Orthogonal Tensor). *Let $x \in \mathbb{K}(D)$ be a tensor and $M \subset D$ a mode set. x is called* M-orthogonal *if $\overline{x}^{(\mathrm{R}(M))}\, x^{(\mathrm{C}(M))} = \mathbb{I}_M$.*

Definition 9 (Tensor Orthogonalisation). *Let $x \in \mathbb{K}(D)$ be a tensor and $k \in D$ some mode such that $\prod_{\ell\in D\setminus\{k\}} n_\ell \geq n_k$. The symbol $\mathrm{QR}_k(x)$ denotes a pair $(b \in \mathbb{K}(D),\, c \in \mathbb{K}(\{k\}^2))$ of tensors such that $x = bc$ and b is $\{k\}$-orthogonal.*

Definition 10 (Tensor SVD [3]). *Let $x \in \mathbb{K}(D)$ be a tensor and $k \in D$ some mode such that $\prod_{\ell\in D\setminus\{k\}} n_\ell \geq n_k$. The symbol $\mathrm{SVD}_k(x)$ denotes a triplet*

$$\big(b \in \mathbb{K}(D),\, s \in \mathbb{K}(\{k\}^2),\, d \in \mathbb{K}(\{k\}^2)\big)$$

such that $x = bsd$, b and d are $\{k\}$- and $\{\mathrm{R}(k)\}$-orthogonal, respectively, and s has the diagonal structure

$$s(i_{\{\mathrm{R}(k)\}} \times i_{\{\mathrm{C}(k)\}}) = \begin{cases} s(i_{\{k\}}) & \text{if } i_{\{\mathrm{R}(k)\}} = i_{\{\mathrm{C}(k)\}} =: i_{\{k\}}, \\ 0 & \text{otherwise,} \end{cases}$$

for some $s(i_{\{k\}}) \geq 0$ called singular values*.*

Remark 11. *We use the letters b, c and b, s, d rather than the more common q, r and u, s, v to denote the factors in the tensor QR decomposition and SVD because many of the latter will be used to refer to vertices in trees.*

Remark 12. *Tensor orthogonalisation and SVD can be conveniently implemented via the matrix QR decomposition and SVD. Both procedures cost $\mathcal{O}\left(n_k^2 \prod_{\ell\in D\setminus\{k\}} n_\ell\right)$ floating-point operations [6, §5.4.5].*

REMARK 13 (Tensor Network Diagrams). *Complicated mode products are sometimes more conveniently expressed in a graphical notation obtained as follows. Draw each tensor as a vertex with an outgoing edge for each of its modes. Connecting two edges implies contracting the corresponding modes. Instructive examples of this notation can be found in [13, Figure 2.1].*

2.2. Hierarchical Tucker Representation. The gist of the tensor network approach is to split a single, high-dimensional tensor $x \in \mathbb{K}(D)$ into a product $x = \prod_{v \in V} x_v$ of many low-dimensional tensors x_v. The template according to which the tensor is split is provided by the *mode tree*.

DEFINITION 14 (Mode Tree). *A triplet (V, E, D) consisting of an undirected tree $(V, E \subseteq \{u - v \mid u, v \in V\})$ and a function D mapping each vertex $v \in V$ to some (possibly empty) mode set $D(v)$ is called a* mode tree *if the $D(v)$ are disjoint after stripping the row and column modes.*

We use the notation $u - v$ rather than the more common $\{u, v\}$ to denote unordered pairs $u - v = v - u$ because the expression $\mathrm{f}(\{u, v\})$ for tagging an edge $\{u, v\}$ with a function-like symbol f could be misinterpreted as $\mathrm{f}(\{u, v\}) = \{\mathrm{f}(u), \mathrm{f}(v)\}$. The condition on $D(v)$ is required to make the formula $\prod_{v \in V} x_v$ to be introduced Definition 15 well defined. An example of two mode sets which are disjoint but not disjoint after stripping the row/column modes is given by $\{\mathrm{R}(k)\}$, $\{\mathrm{C}(k)\}$.

The elements of $D(v)$ are called *free modes* at $v \in V$. Depending on the context, the symbol D may also refer to the set of all free modes $D := \bigcup_{v \in V} D(v)$. The set of edges incident to $v \in V$ is denoted by $E(v)$.

We will employ the usual graph-related terminology. In particular:

- Given a vertex $v \in V$, we define $\mathrm{neighbour}(v) := \{u \in V \mid u - v \in E\}$.
- Given a pair of vertices $u, v \in V$, we write $\mathrm{path}(u, v)$ to refer to the unique set of vertices connecting u and v (recall (V, E) is a tree). That is, $\mathrm{path}(u, v) = \{v\}$ if $u = v$, otherwise it is the unique set satisfying $u, v \in \mathrm{path}(u, v)$ and

$$\#\,\mathrm{neighbour}(w) \cap \mathrm{path}(u, v) = \begin{cases} 1 & \text{for } w = u, v, \\ 2 & \text{for } w \in \mathrm{path}(u, v) \setminus \{u, v\}. \end{cases}$$

  Throughout this document, $\#S$ denotes the number of elements in a finite set S.
- Given a *root* $r \in V$ and a vertex $v \in V$, we define

$$\begin{aligned} \mathrm{parent}(v \mid r) &:= \mathrm{path}(v, r) \cap \mathrm{neighbour}(v), \\ \mathrm{child}(v \mid r) &:= \mathrm{neighbour}(v) \setminus \mathrm{parent}(v \mid r), \\ \mathrm{sibling}(v \mid r) &:= \mathrm{child}(\mathrm{parent}(v \mid r) \mid r) \setminus \{v\}, \\ \mathrm{descendant}(v \mid r) &:= \{u \in V \mid v \in \mathrm{path}(u, r)\}, \\ \mathrm{depth}(v \mid r) &:= \#\,\mathrm{path}(v, r) - 1, \\ \mathrm{height}(v \mid r) &:= \max\{\mathrm{depth}(u \mid r) \mid u \in \mathrm{descendant}(v \mid r)\} - \mathrm{depth}(v \mid r), \\ \mathrm{leaf} &:= \{v \in V \mid \#\,\mathrm{neighbour}(v) = 1\}, \\ \mathrm{interior}(r) &:= V \setminus (\{r\} \cup \mathrm{leaf}). \end{aligned}$$

  Note that $\mathrm{parent}(v \mid r)$ is a singleton unless $v = r$. We will therefore often treat it as a vertex $\mathrm{parent}(v \mid r) \in V$ rather than a set of vertices

[1] Put differently, $\mathrm{height}(v)$ is the longest distance from v to any leaf in $\mathrm{descendant}(v)$.

parent$(v \mid r) \subseteq V$, as we already did in the definition of sibling. In this convention, both parent as well as sibling become undefined if $v = r$.

The analogous concept to mode trees in the literature is the *dimension partition tree* from [7, 10]. These have three additional properties.

- There is an a-priori chosen root $r^\star \in V$.
- The tree is a proper binary tree, i.e. $\#\,\mathrm{child}(v \mid r^\star) \in \{0, 2\}$ for all $v \in V$.
- Only the leaf vertices $v \in \mathrm{leaf}$ have free modes, and each leaf vertex has exactly one free mode. Symbolically,

$$\#D(v) = \begin{cases} 1 & \text{if } v \in \mathrm{leaf}, \\ 0 & \text{otherwise.} \end{cases}$$

We call a mode tree satisfying these constraints *standard*. Furthermore, a standard tree is called *balanced* if

$$\max_{v \in \mathrm{leaf}} \mathrm{depth}(v \mid r^\star) - \min_{v \in \mathrm{leaf}} \mathrm{depth}(v \mid r^\star) \leq 1.$$

Definition 15 (HTR Network). *Let (V, E, D) be a mode tree. A tuple of tensors $x \in \mathrm{HTR}(V, E, D) := \bigtimes_{v \in V} \mathbb{K}(E(v) \cup D(v))$ is called an* HTR network. *It represents a tensor $x := \prod_{v \in V} x_v \in \mathbb{K}(D)$ which we denote by the same symbol x. The mode sizes n_e of the edge modes $e \in E$ are called* ranks *and may differ for two different networks $x, y \in \mathrm{HTR}(V, E, D)$. We therefore clarify which ranks are meant by writing $n_e(x)$, $n_e(y)$.*

In a straightforward continuation of the above notation, we write

$$\mathrm{HTR}(V, E, D^2) := \bigtimes_{v \in V} \mathbb{K}(E(v) \cup D(v)^2)$$

to denote the space of HTR-formatted linear operators.

2.3. Discussion. Two key advantages of our notation are related to the role of the root in HTR algorithms, which is to define the vertex with respect to which we orthogonalise or compute reduced quantities like Gramians or contracted subtrees (to be introduced in Definition 29). While this root remains fixed in the HTR orthogonalisation and truncation algorithms from [7, 10], the ALS algorithms described below move it through the network. Describing such *rerootings* in terms of the dimension partition tree construction from [7, 10] is difficult since there the representation of the tree structure is very much dependent on the particular choice of the root, and relating representations of the same tree with different roots is fairly complicated. In the mode tree construction given above, on the other hand, the root is simply an additional parameter such that rerooting a tree is as simple as redefining this parameter.

It is a common pattern in [7, 10] that expressions take different forms depending on the type of vertex we are considering. An illustrative example is provided by the orthogonalisation Algorithm 3 from [7] where four different cases are distinguished depending on whether the vertex is a leaf or an interior one and whether it is the first or second child of its parent. Because of the aforementioned rerooting in the ALS algorithm, we also have to orthogonalise with respect to vertices which are not the a priori chosen root of the dimension partition tree. Consequently, we have to orthogonalise over each edge not only in leaves-to-root but also in root-to-leaves direction and introduce further rules regarding the treatment of the root. In the notation from [7],

the number of cases to distinguish would therefore rise to ten, a number we consider large enough to best be avoided. In contrast, our notation leverages the symmetries of the problem and thereby allows to treat all these cases at once, see the discussion of orthogonalisation algorithms in subsection 4.1 and in particular Definition 27. The same problem also occurs when discussing the assembly of contracted subtrees, where once again our notation greatly reduces the number of cases to distinguish.

Another important feature of our notation is the conciseness of tensor contractions. To illustrate this point, let us compare our formula $x = \prod_{v \in V} x_v$ relating a tensor $x \in \mathbb{K}(D)$ to its HTR representation $(x_v)_{v \in V} \in \mathrm{HTR}(V, E, D)$ with its counterpart [10, (11.26)]. Both notations introduce families of summation indices (i_{u-v} in our case, $\ell[\alpha]$ in [10]), but we then shorten the formula by treating the basis and coefficient tensors $b^{(j)}$, $c^{(\alpha,\cdot)}$ in a uniform manner and hiding the indices as well as the corresponding summations in the definition of the mode product. We believe the latter point to be justified because we expect readers with prior knowledge of tensor network techniques to interpret formulae like $\prod_{v \in V} x_v$ correctly even without knowing the details of our mode product construction such that its precise definition only serves to confirm their guess. On the other hand, removing the boilerplate symbols emphasises the nontrivial statements, namely that the product runs over all vertices and not subsets thereof as in (2) (Environment Tensor) or Definition 25 (Subtree Tensor). Achieving this conciseness is particularly important when discussing local LSEs and contracted subtrees where even in the much simpler TT case the formulae become fairly lengthy, cf. [19, §3.4] and [13, §5.1].

3. Parallelisation of HTR Algorithms. Most HTR algorithms exhibit a common algorithmic structure to be pointed out in subsection 3.1. It is this structure which will allow us to make fairly general statements regarding the parallel scalability of HTR algorithms as well as their parallel implementation later in this section.

3.1. Common Structure.

Definition 16 (Tree Traversing Algorithm). *Given a tree (V, E), an algorithm is called* tree traversing *if there exists a partially ordered set S of ordered pairs (u, v) involving neighbouring vertices $u, v \in V$ such that the algorithm can be formulated as follows.*

1: **for** each $(u, v) \in S$ **do**
2: On u: Prepare a message m
3: Transfer m from u to v
4: On v: Consume m
5: **end for**

The for-loop on line 1 traverses through the pairs such that if a pair $p \in S$ is visited before another pair $p' \in S$, then either $p \leq p'$ or p and p' are incomparable in the partial order of S.

Definition 17 (Root-to-Leaves Algorithm). *A tree traversing algorithm is called* root-to-leaves *if there exists a vertex $r \in V$ such that the set S and the partial order defined thereon are given by*

$$S := \{(\mathrm{parent}(v \mid r), v) \mid v \in V \setminus \{r\}\},$$
$$(\mathrm{parent}(v \mid r), v) \leq (\mathrm{parent}(u \mid r), u) \quad :\Longleftrightarrow \quad v \in \mathrm{path}(u, r).$$

Definition 18 (Leaves-to-Root Algorithm). *A tree traversing algorithm is called* leaves-to-root *if there exists a vertex $r \in V$ such that the set S and the partial order*

defined thereon are given by

$$S := \{(v, \text{parent}(v \mid r)) \mid v \in V \setminus \{r\}\},$$
$$(v, \text{parent}(v \mid r)) \leq (u, \text{parent}(u \mid r)) \quad :\iff \quad u \in \text{path}(v, r).$$

DEFINITION 19 (Parallel Tree Traversing Algorithm). *A tree traversing algorithm is called* parallel *if the preparation/consumption of messages not ordered by the partial order on S and occurring on different vertices can be executed concurrently.*

DEFINITION 20 (Tree Parallel Algorithm). *An algorithm consisting of one or more parallel root-to-leaves and/or leaves-to-root parts is called* tree parallel.

The orthogonalisation and truncation procedures from [7] as well as the computation of the inner product are all tree parallel algorithms, and it will be the topic of section 4 to develop a tree parallel version of the ALS algorithm. This category therefore includes all major HTR algorithms.

3.2. Theoretical Parallel Scaling. We next analyse the parallel scaling of tree parallel algorithms based on the following assumptions.

ASSUMPTION 21. *Let* `A` *be a tree parallel algorithm consisting of a single root-to-leaves/leaves-to-root part running on a standard balanced mode tree (V, E, D) with root $r^\star$ equal to the root r from Definition 17/18. We assume:*

- *The operations on a vertex $v \in V$ can only be run once all incoming messages have been received. These local operations cannot be further parallelised, and the outgoing messages can only be sent once all operations on vertex v have finished.*
- *It takes* `A` *one time unit to prepare/consume all messages at an interior vertex $v \in \text{interior}(r^\star)$, and no time at the root $r^\star$ or a leaf $v \in \text{leaf}$.*
- *Transferring messages takes no time.*

The first assumption simplifies the model in that it allows to associate all operations with vertices instead of endpoints of edges. In the following, we will therefore use the expression "to process vertex v" to denote the consumption of all incoming and the preparation of all outgoing messages on a vertex $v \in V$. The second assumption is derived from the fact that for a standard mode tree, the interior vertex tensors are three-dimensional and therefore typically have many more elements than the root or leaf tensors which are only two-dimensional. Its main implication is that we can split the time dimension into discrete, equally sized time steps which we will index by the zero-based integer $t \in \mathbb{N}$. All of these assumptions are only approximately satisfied in practice. The idea behind the theory developed next is therefore not to explain the scaling behaviour of tree parallel algorithms in all details, but rather to serve as a reasonably accurate reference against which the empirically observed scaling can be compared.

LEMMA 22 (Equivalence of Leaves-to-Root and Root-to-Leaves Algorithms). *Let* `RtL` *and* `LtR` *be parallel root-to-leaves/leaves-to-root algorithms satisfying Assumption 21, and let $T_{\texttt{A}}(p)$ denote the optimal runtime of algorithm* `A` *running on p processors. Then, $T_{\texttt{RtL}}(p) = T_{\texttt{LtR}}(p)$ for all processor counts p.*

Proof. In order to run a tree parallel algorithm, we need to specify for each time step t and each processor q the vertex $v(t, q)$ which is to be processed, if any. We call such a function $v(t, q)$ a *vertex schedule*. Let $v_{\texttt{RtL}}(t, q)$ be an optimal vertex schedule for `RtL`, i.e. $v_{\texttt{RtL}}(t, q)$ is compatible with the constraints from Definition 17

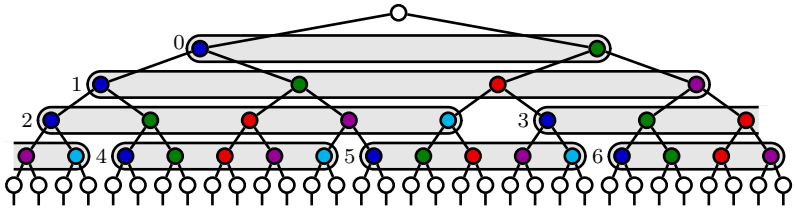


Fig. 1: Example vertex schedule for $d = 32$, $p = 5$. The colors distinguish between processors and the grey beams group the vertices according to the time step during which they are processed.

and leads to the optimal parallel execution time $T_{\texttt{RtL}}(p)$. One easily verifies that $v_{\texttt{LtR}}(t, q) := v_{\texttt{RtL}}(T_{\texttt{RtL}}(p) - 1 - t, q)$ is a valid vertex schedule for `LtR` and has the same execution time, which proves $T_{\texttt{RtL}}(p) \geq T_{\texttt{LtR}}(p)$. Applying the same argument in the opposite direction proves $T_{\texttt{RtL}}(p) \leq T_{\texttt{LtR}}(p)$ which yields the claim. ∎

THEOREM 23 (Scaling of Tree Parallel Algorithms). *Let `A` be an algorithm satisfying Assumption 21 and set $d := \#D$. The optimal runtime of `A` on $p \leq \lfloor \frac{d}{2} \rfloor$ processors is given by*

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

and the optimal parallel speedup is

$$S(p) := \frac{T(1)}{T(p)} = O\left(\frac{d}{\log_2 p + \frac{d}{p}} \right). \tag{1}$$

Providing $p > \lfloor \frac{d}{2} \rfloor$ processors does not yield additional speedup.

Proof. Clearly, the largest number of vertices we can process during a single time step is $\#\{v \in V \mid \text{height}(v \mid r^\star) = 1\} = \lfloor \frac{d}{2} \rfloor$ which proves the last statement. We therefore assume $p \leq \lfloor \frac{d}{2} \rfloor$ in the remainder of this proof. By Lemma 22, it is further sufficient to consider only a root-to-leaves algorithm. We propose the following vertex schedule for this case (see also Figure 1): at time $t < t(p) := \lceil \log_2 p \rceil - 1$, pick any $2^{t+1} < p$ processors and let these process the vertices at depth $t + 1$ (the $+1$ takes into account that we do not need to allocate time for processing the root). We thus process $2^{t(p)+1} - 2$ interior vertices until time $t(p)$. For times $t \geq t(p)$, enumerate the remaining $d - 2^{t(p)+1}$ interior vertices starting from depth $t(p) + 1$ and proceeding in breadth-first order, then let processor $q \in \{0, \ldots, p-1\}$ process vertex $(t - t(p)) \cdot p + q$ in that order. If no such vertex exists, i.e. if $(t - t(p)) \cdot p + q \geq d - 2^{t(p)+1}$, then the processor will wait for at most one time step until all other processors have finished.

It is easily seen that this vertex schedule satisfies the constraints imposed by a root-to-leaves algorithm. Since processing one vertex renders at most two further vertices ready for processing and we start with two ready vertices at time $t = 0$, an upper bound for the number of vertices we can process at any time $t \in \mathbb{N}$ is 2^{t+1}. Up to time $t(p)$, we meet this limit in every time step. From time $t(p)$ onwards, we keep all processors busy until the list of remaining vertices is exhausted, which takes $t'(p) := \left\lceil \frac{d - 2^{t(p)+1}}{p} \right\rceil$ time steps. The runtime of $t(p) + t'(p) = T(p)$ is therefore both achievable as well as optimal. ∎

If a tree parallel algorithm consists of more than one root-to-leaves/leaves-to-root part, we assume that the parts must be run one after the other and cannot overlap. Again, this assumption is not necessarily satisfied in practice, but it simplifies the argument and provides a reasonable approximation to reality. The optimal runtime of the algorithm on p processors is then given by $\sum_{i=1}^{n} c_i\, T(p)$, where n denotes the number of such parts and the c_i take into account that each part may require a different unit time per interior vertex. When computing the optimal parallel speedup, the c_i factor out and cancel, therefore (1) is still valid even in this more general setting.

3.3. Parallel Implementation. The proof of Theorem 23 presented a parallelisation scheme for HTR algorithms in the idealised setting of Assumption 21. In practice, however, several complications arise such that this scheme may not apply or the optimality guarantee given by Theorem 23 may no longer be valid:

- The mode tree may be non-standard and/or not balanced.
- The HTR ranks may be non-uniform such that the uniform-cost-per-interior-vertex assumption is not satisfied.
- Inter-process communication costs may be non-negligible.

This subsection discusses a set of strategies for handling such issues. Following the structure exposed in Definition 16, we assume for this purpose that an HTR algorithm is given as a list of jobs (namely the preparation or consumption of messages) each of which is associated with a vertex of the mode tree and may depend on the completion of other jobs before being run. Parallelising such an algorithm then amounts to specifying for each job a) on which processor and b) when it is to be executed such that all dependency constraints are met and the overall runtime is minimised.

To settle the "where" question, we let the user specify a *vertex distribution*, a function $q(v)$ mapping each vertex $v \in V$ to a processor $q(v) \in \{0, \dots, p-1\}$ by whom the jobs associated with vertex v are to be run. The rationale for this design choice is the observation that devising vertex distributions delivering decent performance requires much insight into the problem at hand and is therefore best done on a case-by-case basis. Assigning all jobs of vertex v to the same processor $q(v)$ allows to store all the data associated with v exclusively on processor $q(v)$ and therefore reduces the need for communication.

The "when" question, on the other hand, is answered by the *longest path first* (LPF) scheduling algorithm [14] introduced next. Assume we know for each vertex $v \in V$ the time $t(v)$ it takes to execute its associated jobs. We then define the *weighted depth*

$$\operatorname{depth}_t(v \mid r) := \sum_{u \in \operatorname{path}(v,r) \setminus \{v\}} t(u)$$

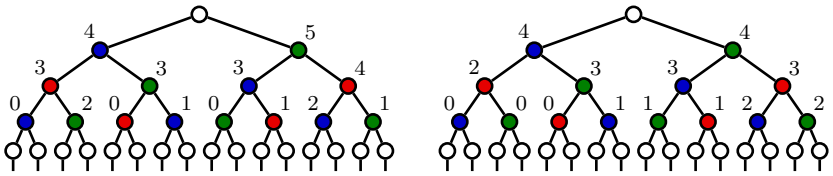
and the *weighted height*

$$\operatorname{height}_t(v|r) := \max\{\operatorname{depth}_t(u|r) + t(u) \mid u \in \operatorname{descendant}(v|r)\} - \operatorname{depth}_t(v|r) - t(v).$$

We further define the *local connected component* $C(v) \subseteq V$ of a vertex $v \in V$ to be the largest connected component such that $v \in C(v)$ and $q(u) = q(v)$ for all $u \in C(v)$, and finally set $\text{local root}(v \mid r) := \arg\min_{u \in C(v)} \operatorname{depth}(u \mid r)$.

Let each processor manage a list of ready jobs, i.e. jobs which are not blocked by dependencies on other jobs. Once a processor finishes a job, it waits until this list becomes non-empty and then chooses the job to work on next according to either of the following rules, depending on the type of the algorithm.

Leaves-to-root: Pick any ready job on one of the vertices $v \in V$ which maximise $\operatorname{depth}_t(\text{local root}(v \mid r) \mid r)$.



(a) Suboptimal schedule. (b) Optimal schedule.

Fig. 2: Vertex distribution for which LPF scheduling may not deliver optimal performance. Let the algorithm in question be leaves-to-root and Assumption 21 hold. The colors distinguish between processors and the numbers indicate the time step at which the vertices are processed. Both of the shown schedules are valid LPF schedules, yet the left one has a runtime of six time steps while the right one requires only five.

Root-to-leaves: Pick any ready job on one of the vertices $v \in V$ which maximise

$$\max\{\text{height}_t(u \mid r) + t(u) \mid u \notin C(v) \wedge \text{parent}(u \mid r) \in C(v)\}.$$

Let us motivate this scheduling at the example of a leaves-to-root algorithm. On the one hand, we note that the order in which processor q executes the jobs within one of its local connected components $C \subseteq V$ does not matter as the runtime will in any case be $\sum_{v \in C} t(v)$. What does matter, however, is the time t at which q sends the message from the local root u of C to $\text{parent}(u \mid r)$ since this provides a lower bound $t + \text{depth}_t(u \mid r)$ on the overall runtime. The above algorithm greedily minimises this bound. While LPF scheduling does not necessarily achieve the optimal runtime, see Figure 2, we believe that the pathological cases are rare and the resulting loss in performance outweighed by the simplicity of the algorithm.

4. Parallel ALS Algorithm.

4.1. Review of the Serial Algorithm. The *alternating least squares* (ALS) algorithm from [13, 19] tackles the linear system of equations (LSE) $Ax = b$ where the operator $A \in \text{HTR}(V, E, D^2)$, the right-hand side $b \in \text{HTR}(V, E, D)$ and an initial guess $x \in \text{HTR}(V, E, D)$ for the solution $A^{-1}b$ are all represented in HTR. Given such an x and a vertex $r \in V$, let us define the *environment tensor*

$$U_r(x) := \left(\prod_{u \in V \setminus \{r\}} x_u \right) (\text{C}(E(r))) \,. \tag{2}$$

This quantity corresponds to the matrix Q_k from [19] and the retraction operator $P_{i,1,x}$ from [13]. Relabelling the edge modes $E(r)$ of $U_r(x)$ to column modes $\text{C}(E(r))$ is a purely notational trick enabling us to use the more familiar and shorter conjugate transpose notation rather than having to relabel the modes in later formulae, e.g. (3). This notation makes the fairly natural assumption that D does not contain any row or column modes such that transposing the environment tensor only relabels $\text{C}(E(r)) \to \text{R}(E(r))$.

In its simplest form, the ALS algorithm reads as follows.

Algorithm 1 ALS Algorithm

1: **repeat**
2: **for** vertex $r \in V$ **do**
3: Update x_r to the solution of the *local LSE*

$$U_r(x)^* A\, U_r(x)\, x_r = U_r(x)^*\, b. \tag{3}$$

4: **end for**
5: **until** convergence

The very simple structure of Algorithm 1 was brought about by ignoring two important technical constraints:

- The local LSE (3) can only be solved numerically if the condition number $\kappa\left(U_r(x)^* A\, U_r(x)\right)$ of the local matrix is reasonably small.
- The ALS algorithm is only computationally feasible if the local matrix and right-hand side can be assembled efficiently.

The remainder of this subsection will be devoted to demonstrating how to satisfy these constraints in the HTR case. A similar endeavour has already been undertaken in [17], but we need to give further details in order to prepare for the discussion in subsection 4.2.

The following concepts of HTR orthogonality allow us to address the concerns regarding the condition number.

Definition 24 (r-Orthogonality). *An HTR network $x \in \mathrm{HTR}(V, E, D)$ is called* r-orthogonal *with $r \in V$ if $U_r(x)$ is $\mathrm{C}(E_r)$-orthogonal.*

Definition 25 (Subtree Tensor). *Let $x \in \mathrm{HTR}(V, E, D)$ be an HTR network and $v, r \in V$, $v \neq r$, two vertices. We define the* subtree tensor $S_{v|r}(x)$ *through*

$$S_{v|r}(x) := \left(\prod_{u \in \mathrm{descendant}(v|r)} x_u \right) \left(\mathrm{C}(v - \mathrm{parent}(v|r))\right).$$

The reason for relabelling the parent edge mode $v - \mathrm{parent}(v \mid r)$ of $S_{v|r}(x)$ to a column mode $\mathrm{C}(v - \mathrm{parent}(v \mid r))$ is again to simplify later formulae, cf. the remark after (2).

Definition 26 (Strong r-Orthogonality). *An HTR network $x \in \mathrm{HTR}(V, E, D)$ is called* strongly r-orthogonal *if all subtree tensors $S_{v|r}(x)$ with $v \in V \setminus \{r\}$ are $\{\mathrm{C}(v - \mathrm{parent}(v \mid r))\}$-orthogonal.*

It is easily verified that a strongly r-orthogonal network is also r-orthogonal: the environment tensor can be written in terms of the subtree tensors as

$$U_r(x) = \prod_{v \in \mathrm{neighbour}(r)} S_{v|r}(x),$$

and by the $\{\mathrm{C}(v - \mathrm{parent}(v \mid r) = v - r)\}$-orthogonality of the $S_{v|r}(x)$ we have

$$\begin{aligned} U_r(x)^*\, U_r(x) &= \prod_{v \in \mathrm{neighbour}(r)} S_{v|r}(x)^*\, S_{v|r}(x) \\ &= \prod_{v \in \mathrm{neighbour}(r)} \mathbb{I}_{\{v-r\}} = \mathbb{I}_{E_r}. \end{aligned}$$

The above definition of strong r-orthogonality is simply a rewrite of the orthogonality concept from [7, Definition 3.5] in our root-invariant notation. The reason why we need to distinguish between r-orthogonality and strong r-orthogonality will become apparent when discussing parallelisation in subsection 4.2.

If x is r-orthogonal and A Hermitian, Theorem 4.1b) in [13] allows us to bound the condition number of the local LSEs by $\kappa\left(U_r(x)^* A\, U_r(x)\right) \leq \kappa(A)$. Our aim is therefore to *r-orthogonalise* x before solving the local LSE at r, i.e. to transform the vertex tensors $(x_v)_{v\in V}$ such that x becomes r-orthogonal but the represented tensor $\prod_{v\in V} x_v$ remains unchanged. We next reformulate and extend the orthogonalisation algorithm from [7, Alg. 3] to obtain an efficient scheme for iteratively orthogonalising an HTR network $x \in \mathrm{HTR}(V, E, D)$ with respect to all its vertices $r \in V$ as required by the ALS Algorithm 1. This scheme will make use of the following vertex-wise operation.

Definition 27 (Vertex Orthogonalisation). *Let $x \in \mathrm{HTR}(V, E, D)$ be an HTR network and $u - v \in E$ an edge.* Orthogonalisation of x_v with respect to $u - v$ *is defined as the following operation:*

1: $(b, c) := \mathrm{QR}_{u-v}(x_v)$
2: $x_v := b$
3: $x_u := c x_u$

Note that vertex orthogonalisation does not modify the represented tensor since we have $\tilde{x}_v \tilde{x}_u = bcx_u = x_v x_u$ where x_u, x_v and $\tilde{x}_u, \tilde{x}_v$ denote the vertex tensors before and after the orthogonalisation step, respectively. The trick to strongly r-orthogonalise an HTR network is to orthogonalise its vertices in the right order, which is leaves-to-root:

Algorithm 2 Strong r-Orthogonalisation

1: **for** $v \in \mathrm{neighbour}(r)$ **do** Recurse(u) **end for**
2: **function** Recurse(v)
3: **for** $u \in \mathrm{child}(v \mid r)$ **do** Recurse(u) **end for**
4: Orthogonalise x_v with respect to $v - \mathrm{parent}(v \mid r)$ // *Definition 27*
5: **end function**

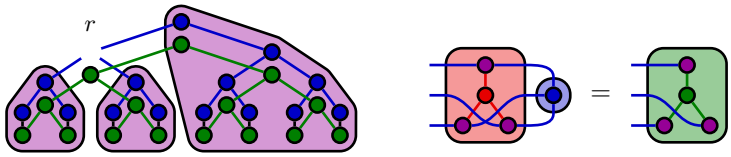
Correctness of this algorithm follows easily by induction in leaves-to-root direction. Once an HTR network is strongly orthogonal with respect to any vertex $r \in V$, we can move this orthogonal centre around using the following theorem.

Theorem 28. *Let $x \in \mathrm{HTR}(V, E, D)$ be an HTR network and $r \in V$, $r' \in \mathrm{neighbour}(r)$ two vertices such that x is strongly r-orthogonal. After orthogonalising x_r with respect to $r - r'$, x is strongly r'-orthogonal.*

Proof. Because x was initially strongly r-orthogonal and the subtree tensors $S_{v|r}(x) = S_{v|r'}(x)$ with $v \in V \setminus \{r, r'\}$ are not affected by the vertex orthogonalisation, we only need to prove $S_{r|r'}(x)$ is $\{\mathrm{C}(r - \mathrm{parent}(r \mid r') = r - r')\}$-orthogonal. We can write $S_{r|r'}(x) = \left(\prod_{v\in\mathrm{child}(r|r')} S_{v|r}(x)\right) x_r(\mathrm{C}(r-r'))$, therefore we have

$$\begin{aligned} S_{r|r'}(x)^*\, S_{r|r'}(x) &= \overline{x_r}(\mathrm{R}(r-r')) \left(\prod_{v\in\mathrm{child}(r|r')} S_{v|r}(x)^*\, S_{v|r}(x) \right) x_r(\mathrm{C}(r-r')) \\ &= \overline{x_r}\left(\mathrm{R}(r-r')\right)\; x_r\left(\mathrm{C}(r-r')\right) = \mathbb{I}_{\{r-r'\}}. \end{aligned}$$

∎



(a) Contracted subtrees. (b) Local LSE.

Fig. 3: (a) Contracted subtrees $(x|b)_{v|r}$ for $v \in \text{neighbour}(r)$. (b) Local LSE in terms of the contracted subtrees $(x|A|x)_{v|r}$, $(x|b)_{v|r}$, $v \in \text{neighbour}(r)$. In both figures, vertices and edges of A are shown in red, of x in blue and of b in green. See Remark 13 for an introduction to tensor network diagrams.

In conclusion, stabilisation of the ALS Algorithm 1 requires us to make an initial call to the strong r-orthogonalisation Algorithm 2 and then let the orthogonality centre follow the vertex on which we solve the local LSE by using Theorem 28. Because of this orthogonalisation scheme, a single ALS iteration runs faster if we choose to visit the vertices in an order such that consecutive vertices are always neighbours, and it has been verified in [17] that the vertex order has no significant impact on the convergence as a function of the iteration count. In the following, we will therefore assume the fixed backtracking depth first search (bDFS) order shown in Figure 4a. Note that bDFS tree traversal visits a vertex several times, namely once per incoming edge, in contrast to what is implied by the for-loop in Algorithm 1, and simplifies to the sweeping scheme from [13, 19] in the TT case.

The key to the efficient assembly of the local LSEs are the *contracted subtrees*, which are the HTR analogues of the tensors Ψ_k, Φ_k in [19] and G_i (without the last vertex tensor A_i), H_i in [13].

Definition 29 (Contracted Subtrees). *Let*

$$x \in \mathrm{HTR}(V, E, D), \quad A \in \mathrm{HTR}(V, E, D^2), \quad y \in \mathrm{HTR}(V, E, D)$$

be HTR networks and $v, r \in V$, $v \neq r$, two vertices. We define the contracted subtrees

$$\begin{aligned} (x|y)_{v|r} &:= \prod_{u \in \text{descendant}(v|r)} \overline{x_u}\big(\mathrm{l}(E(v))\big)\, y_u\big(\mathrm{r}(E(v))\big), \\ (x|A|y)_{v|r} &:= \prod_{u \in \text{descendant}(v|r)} \overline{x_u}\big(\mathrm{l}(E(v))\big)\, A_u\big(\mathrm{m}(E(v))\big)\, y_u\big(\mathrm{r}(E(v))\big). \end{aligned} \tag{4}$$

The l, m and r tags in (4) stand for *left*, *middle* and *right* and serve to prevent the edge modes of different HTR networks from being multiplied.

The local matrix and the local right-hand side at a vertex $r \in V$ can be expressed in terms of the contracted subtrees as

$$\begin{aligned} U_r(x)^* A\, U_r(x) &:= \left(A_r\big(\mathrm{m}(E(r))\big) \prod_{v \in \text{neighbour}(r)} (x|A|x)_{v|r} \right) \begin{pmatrix} \mathrm{l}(E(r)) \to \mathrm{R}(E(r)), \\ \mathrm{r}(E(r)) \to \mathrm{C}(E(r)) \end{pmatrix}, \\ U_\alpha(x)^*\, b &:= \left(b_r\big(\mathrm{r}(E(r)\big) \prod_{v \in \text{neighbour}(r)} (x|b)_{v|r} \right) \big(\mathrm{l}(E(r)) \to E(r)\big), \end{aligned} \tag{5}$$

see also Figure 3, thus assembling the local LSE incurs little extra cost once the contracted subtrees are available. These in turn can be obtained efficiently through the recursion formulae

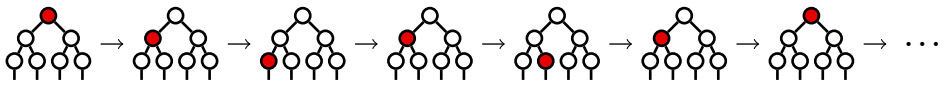
$$
\begin{aligned}
(x|y)_{v|r} &:= \overline{x_v}\big(\mathrm{l}(E(v))\big)\, y_v\big(\mathrm{r}(E(v))\big) \prod_{u\in\mathrm{child}(v|r)} (x|y)_{u|r}, \\
(x|A|y)_{v|r} &:= \overline{x_v}\big(\mathrm{l}(E(v))\big)\, A_v\big(\mathrm{m}(E(v))\big)\, y_v\big(\mathrm{r}(E(v))\big) \prod_{u\in\mathrm{child}(v|r)} (x|A|y)_{u|r}.
\end{aligned} \tag{6}
$$

The resulting algorithm for assembling the local LSEs has then exactly the same structure as the orthogonalisation scheme presented above: in a first step, we pick a starting vertex $r \in V$ and compute the contracted subtrees $(\star)_{v|r}$ for all $v \in \mathrm{neighbour}(r)$ ($\star$ stands for both $x|A|x$ and $x|b$), which if evaluated according to (6) requires to compute all contracted subtrees $(\star)_{v|r}$ for $v \in V \setminus \{r\}$. If we then move from r to one of its neighbours $r' \in \mathrm{neighbour}(r)$, we can reuse the $(\star)_{v|r} = (\star)_{v|r'}$ with $v \in \mathrm{child}(r' \mid r)$ such that the only contracted subtrees to compute anew are the two tensors $(\star)_{r|r'}$. Because we already have the $(\star)_{v|r}$ for $v \in \mathrm{sibling}(r' \mid r)$, this step involves only computations on r and has therefore a constant cost with respect to the network size. Proceeding further according to the bDFS tree traversal order from Figure 4a, we can continue in this manner for all vertices in the network.

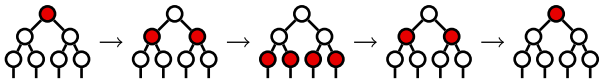
Algorithm 3 HTR ALS Algorithm

```
1:  Pick an arbitrary starting vertex r ∈ V
2:  Strongly r-orthogonalise x                          // Algorithm 2
3:  Compute (x|A|x)_{v|r} and (x|b)_{v|r} for all v ∈ V \ {r}   // Equation (6)
4:  repeat RECURSE(r, r) until convergence
5:  function RECURSE(r, p)
6:      for v ∈ child(r | p) do
7:          Solve the local LSE at r                    // Equations (3), (5)
8:          Orthogonalise x_r with respect to r − v     // Definition 27
9:          Compute (x|A|x)_{r|v} and (x|b)_{r|v}       // Equation (6)
10:         RECURSE(v, r)
11:     end for
12:     Solve the local LSE at r                        // Equations (3), (5)
13:     if r ≠ p then
14:         Orthogonalise x_r with respect to r − p     // Definition 27
15:         Compute (x|A|x)_{r|p} and (x|b)_{r|p}       // Equation (6)
16:     end if
17: end function
```

4.2. Parallelisation. An important feature of the local LSE (3) is that its matrix and right-hand side depend on all vertex tensors of x. We therefore must not modify x while one local solve is running, and in particular we cannot solve multiple local LSEs concurrently. In Algorithm 3, this is expressed by the fact that we must run the loop over the children (line 6) sequentially, because the computations for the local LSE and the contracted subtrees on lines 7, 9 depend on the contracted subtrees computed on line 15 in the subordinate calls to RECURSE. The ALS algorithm as presented in subsection 4.1 is therefore not parallelisable without algorithmic modifications eliminating the dependency between local LSE solves. Since already the



(a) Serial ALS Algorithm 3: backtracking depth first search (bDFS).



(b) Parallel ALS Algorithm 4: root-to-leaves followed by leaves-to-root.

Fig. 4: Tree traversal orders of the serial and parallel HTR ALS algorithm. The red vertices denote the ones on which we currently solve local LSEs.

serial ALS Algorithm 3 does not explicitly access all vertex tensors but rather uses cached contracted subtrees to assemble the local LSEs, eliminating this dependency is very easy: we simply drop the aspiration that the cached contracted subtrees must equal the ones computed from only the most recent vertex tensors. More precisely, we rearrange the loop on lines 6 to 11 in Algorithm 3 as follows.

1: Solve the local LSE at r // *Equations* (3), (5)
2: **for** $v \in \mathrm{child}(r \mid p)$ **do**
3: Compute $(x|A|x)_{r|v}$ and $(x|b)_{r|v}$ // *Equation* (6)
4: **end for**
5: **parallel for** $v \in \mathrm{child}(r \mid p)$
6: RECURSE(v, r)
7: **end parallel for**

The intended meaning is that we eliminate the dependencies in the second loop by precomputing the contracted subtrees on line 3 and using only these precomputed values on line 6, ignoring the fact that they become outdated with the first local LSE solve on this line. The resulting parallel tree traversal order is shown in Figure 4b.

To justify why we put the local LSE solve before both loops, we note that the first loop in the above pseudocode-snippet reads x_r but does not generate new information which would influence the local LSE at r, while the second loop does generate such information but does not read x_r. Therefore, putting line 1 into the first loop would amount to solving the same problem multiple times, while putting it into the second loop would mean to solve intermediate problems whose solutions we do not need.

The above pseudocode-snippet does not yet include orthogonalisation because the parallel setting introduces the additional difficulty that we have to orthogonalise with respect to multiple vertices at the same time. We propose the following algorithm to achieve this.

DEFINITION 30 (Child-Orthogonalisation). *Let $x \in \mathrm{HTR}(V, E, D)$ be an HTR network and $r, p \in V$ two vertices.* Orthogonalisation of x_r with respect to its children relative to p *is defined as the following operation:*

1: **for** $v \in \mathrm{child}(r \mid p)$ **do**
2: $(b_v, s_v, d_v) := \mathrm{SVD}_{r-v}(x_r)$
3: $x_r := b_v\, s_v$
4: $x_v := d_v\, x_v$

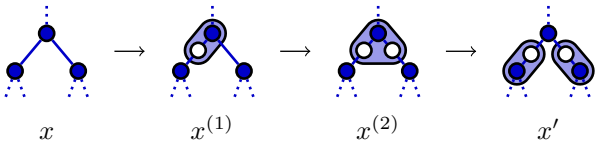


Fig. 5: Child orthogonalisation. The vertex tensors x_r, x_v, $v \in \mathrm{child}(r \mid p)$, of x are shown in blue, the tensors s_v in white. See Remark 13 for an introduction to tensor network diagrams.

5: **end for**
6: **for** $v \in \mathrm{child}(r \mid p)$ **do**
7: $\quad x_r := x_r\, s_v^{-1}$
8: $\quad x_v := s_v\, x_v$
9: **end for**

THEOREM 31. *Child-orthogonalisation does not modify the represented tensor* $x = \prod_{v \in V} x_v$.

Proof. Lines 3, 4 do not modify x since $b_v s_v d_v\, x_v = x_r x_v$, and the same holds for lines 7, 8 since $x_r s_v^{-1} s_v x_v = x_r x_v$. ∎

THEOREM 32. *Assume* $S_{v|r}(x)$ *is* $\{C(v-r)\}$*-orthogonal for all* $v \in \mathrm{neighbour}(r)$ *and some* $r \in V$.[2] *After orthogonalising* x_r *with respect to its children relative to some vertex* $p \in V$, $S_{r|v}(x)$ *is* $\{C(r-v)\}$*-orthogonal for all* $v \in \mathrm{child}(r \mid p)$.

Proof. Let us denote by x the original network and by x' the final network. We define $v_1, \ldots, v_c \in \mathrm{child}(r \mid p)$, $c := \#\,\mathrm{child}(r \mid p)$, to be the children of r in the order in which they appear in the first loop, and denote by $x^{(i)}$, $i = 1, \ldots, c$, the state of the network after the iteration $v = v_i$ of the first loop has been executed (see also Figure 5).

The proof splits into two parts.

1. We have
$$S_{v_i|r}(x^{(j)}) = \begin{cases} S_{v_i|r}(x) & \text{if } j < i, \\ \left(S_{v_i|r}(x)\, d_{v_i}^T\right)_{(C(v-r))} & \text{otherwise,} \end{cases}$$
for $i, j = 1, \ldots, c$. Since d_{v_i} and $S_{v_i|r}(x)$ are $\{\mathrm{R}(v_i - r)\}$- and $\{\mathrm{C}(v_i - r)\}$-orthogonal, respectively, $S_{v_i|r}(x^{(j)})$ is $\{\mathrm{C}(v_i - r)\}$-orthogonal for any $i, j = 1, \ldots, c$.
2. Let $u \in \mathrm{neighbour}(r)$. From the proof of Theorem 31 it follows that $S_{r|u}(x)$ is not modified by lines 3, 4 unless u is equal to the loop variable v. This proves
$$S_{r|v_i}(x^{(c)}) = \left(\prod_{u \in \mathrm{child}(r|v_i)} S_{u|r}(x^{(i)}) \right) (b_{v_i} s_{v_i})_{(C(r-v_i))}, \quad \forall i = 1, \ldots, c. \tag{7}$$

[2] This is equivalent to x being r-orthogonal up to scaling of the $S_{v|r}(x)$, $v \in \mathrm{neighbour}(r)$.

Arguing similarly for the second loop, we obtain

$$S_{r|v_i}(x') = \left(\prod_{u \in \text{child}(r|v_i)} S_{u|r}(x^{(i)}) \right) b_{v_i}(_{C(r-v_i)}), \qquad \forall i = 1, \dots, c. \tag{8}$$

b_{v_i} is $\{r - v_i\}$-orthogonal and the $S_{u|r}(x^{(i)})$ are $\{C(r-u)\}$-orthogonal by assumption for $u = \text{parent}(r \mid p)$ and by part 1 for $u \in \text{sibling}(v_i \mid p)$. Therefore, $S_{r|v_i}(x')$ is $\{C(r - v_i)\}$-orthogonal for all $v_i \in \text{child}(r \mid p)$. □

Definition 30 silently assumed that s_v is invertible, i.e. that no singular value is exactly 0. In our code, we ensure this condition by transforming the singular values $s_v(i_{v-r})$ with

$$s_v(i_{\{v-r\}}) := \max\{s_v(i_{\{v-r\}}), \texttt{eps}\, s_v(i_{\{v-r\}} = 0)\} \tag{9}$$

where `eps` denotes the machine precision and $s_v(i_{\{v-r\}} = 0)$ the largest singular value. Note that finite-machine precision may lead to a similar deviation between the exact singular values and their numerically computed counterparts such that the above transformation does not change the accuracy of the latter. We now analyse how such rounding influences the above results.

Theorem 31 relies on the identities $b_v s_v d_v = x_r$ and $s_v\, s_v^{-1} = \mathbb{I}_{\{v-r\}}$, both of which are satisfied up to machine precision when using the singular values from (9). Thus, the statement in Theorem 31 is valid up to machine precision as well.

In Theorem 32, the rounding in the singular values implies that (7) is satisfied up to a relative error of $\mathcal{O}(\texttt{eps})$. Multiplication with s_v^{-1} may then blow this error up such that the relative error in (8) is $\mathcal{O}(1)$, i.e. the $S_{r|v}(x)$ may not be $\{C(r-v)\}$-orthogonal at all. Luckily, we can limit the impact of rounding errors by interleaving orthogonalisation and subtree computation as follows.

Definition 33 (Child-Orthogonalisation & Contracted Subtree Computation). *Let $x \in \text{HTR}(V, E, D)$ be an HTR network and $r, p \in V$ two vertices.* Combined child-orthogonalisation and contracted subtree computation at x_r relative to p *is defined as the following operation:*

for $v \in \text{child}(r \mid p)$ **do**
  // Orthogonalise x_r with respect to v
  $(b_v, s_v, d_v) := \text{SVD}_{r-v}(x_r)$
  $x_r := b_v$
  $x_v := s_v\, d_v\, x_v$
  // Compute contracted subtrees
  Compute $(x|A|x)_{r|v}$ and $(x|b)_{r|v}$ *// Equation (6)*
  // Temporarily move the non-orthogonal factor to x_r
  $x_r := x_r\, s_v$
  $x_v := s_v^{-1}\, x_v$ *// (*)*
  // Update $(x|A|x)_{v|r}$ and $(x|b)_{v|r}$
  $(x|A|x)_{v|r} := \overline{d_v}\big((\text{l})\big)\, (x|A|x)_{v|r}\, d_v\big((\text{r})\big)$
  $(x|b)_{v|r} := \overline{d_v}\big((\text{l})\big)\, (x|b)_{v|r}$
end for
for $v \in \text{child}(r \mid p)$ **do**
  // Move the non-orthogonal factor to x_v again
  $x_r := x_r\, s_v^{-1}$

18: $x_v := s_v\, x_v$ // (*)
19: // *Update* $(x|A|x)_{v|r}$ *and* $(x|b)_{v|r}$
20: $(x|A|x)_{v|r} := s_\beta\big((\mathrm{l})\big)\ (x|A|x)_{v|r}\ s_\beta\big((\mathrm{r})\big)$ // (+)
21: $(x|b)_{v|r} := s_\beta\big((\mathrm{l})\big)\ (x|b)_{v|r}$ // (+)
22: **end for**

Here, (l) *stands for* $\mathrm{R}(v-r)\to\mathrm{R}(\mathrm{l}(v-r)),\mathrm{C}(v-r)\to\mathrm{C}(\mathrm{l}(v-r))$*, and* (r) *stands for* $\mathrm{R}(v-r)\to\mathrm{C}(\mathrm{r}(v-r)),\mathrm{C}(v-r)\to\mathrm{R}(\mathrm{r}(v-r))$.

The two lines marked with (*) *may be omitted since the second undoes the effect of the first. The two lines marked with* (+) *may be dropped if* $(x|A|x)_{v|r}$ *and* $(x|b)_{v|r}$ *are updated anyway before being used again, as is the case in Algorithm 4.*

It is easily verified that the modifications applied to x in Definition 33 are equivalent to the ones in Definition 30, therefore Theorems 31 and 32 are also valid for combined child orthogonalisation and contracted subtree computation. The key idea of Definition 33 is to compute the contracted subtrees at a point (line 7) where $S_{r|v}(x)$ is $\{C(r-v)\}$-orthogonal up to errors of order $\mathcal{O}(\varepsilon)$. Since the local LSE are assembled based on these cached $(x|A|x)_{r|v}$, $(x|b)_{r|v}$ capturing accurately $\{C(r-v)\}$-orthogonal $S_{r|v}(x)$, it no longer matters that the final $S_{r|v}(x)$ are not accurately $\{C(r-v)\}$-orthogonal. We further remark that the updates to the contracted subtrees on lines 12, 13 and 20, 21 in Definition 33 are a consequence of the modifications done to x_v on lines 5, 10 and 18. In the serial Algorithm 3, such updates are not necessary because $(x|A|x)_{v|r}$, $(x|b)_{v|r}$ are updated anyway on line 15 before being used again.

The final parallel HTR ALS algorithm is summarised in Algorithm 4. It can be considered an adaption of the parallel TT DMRG from [26] to the HTR setting, which yields the advantage that our algorithm has a tree parallel scaling right from the beginning, in contrast to the TT DMRG algorithm which requires an almost serial initial phase. We would furthermore like to point out the similarity between the above orthogonalisation strategy and the HOSVD algorithm from [10, §11.3.3]. In fact, the singular values it computes are exactly the ones from [10] and could in principle be used for truncation, resulting in a scheme similar to the AMEn algorithm from [4]. We do not pursue this idea further, however, because interleaving the ALS and truncation steps complicates the algorithm and results in at most a factor two speedup. In practice, we have found the speedup to be much lower because most of the time is spent on solving the local LSEs rather than orthogonalisation and truncation.

4.3. Computational Costs. We conclude the above discussions by analysing the cost of the HTR ALS algorithm and pointing out the tricks to reduce this cost, as has been done for the TT ALS algorithm in [13, 19]. For this purpose, we assume (V, E, D) to be a standard mode tree and $x \in \mathrm{HTR}(V, E, D)$, $A \in \mathrm{HTR}(V, E, D^2)$, $b \in \mathrm{HTR}(V, E, D)$. We define

$$d := \#D, \quad n := \max_{k\in D} n_k, \quad r := \max_{e\in E} n_e(x), \quad R := \max_{e\in E} n_e(A), \quad R_b := \max_{e\in E} n_e(b).$$

The ranks of x are usually larger than the ranks of A or b, thus we will assume $R^a r^b \le R^{a'} r^{b'}$ if $a+b = a'+b'$ and $b \le b'$, and the analogous inequality for R_b. In the below terms involving both n as well as r, the rank symbol r refers to the ranks at the leaves and therefore satisfies $r \le n$. This allows us to order such terms according to $n^a r^b \le n^{a'} r^{b'}$ if $a + b = a' + b'$ and $a \le a'$.

In the following, we count for each part of the ALS algorithm — orthogonalisation, contracted subtree computations and local LSE solves — the number of floating-point operations (FLOP) arising during a single iteration, i.e. a single call to Recurse(r, r).

Algorithm 4 Parallel HTR ALS Algorithm

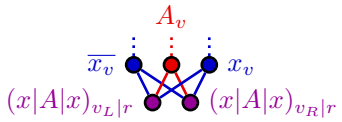
We assume an implicit cache of contracted subtrees. This cache is initialised on line 3 and only updated when we explicitly say so, namely on lines 8 and 16. Each time contracted subtrees are needed (i.e. when computing new contracted subtrees and when solving the local LSE), their values are read from the cache even if the cache is outdated.

```
1: Pick an arbitrary starting vertex r ∈ V
2: Strongly r-orthogonalise x                                  // Algorithm 2
3: Compute (x|A|x)_{v|r} and (x|b)_{v|r} for all v ∈ V \ {r}   // Equation (6)
4: repeat RECURSE(r, r) until convergence
5: function RECURSE(r, p)
6:     if child(r | p) ≠ {} then
7:         Solve the local LSE at r                            // Equations (3), (5)
8:         Child-orthogonalise and compute contracted
               subtrees at x_r relative to p                   // Definition 33
9:         parallel for v ∈ child(r | p)
10:            RECURSE(v, r)
11:        end parallel for
12:    end if
13:    Solve the local LSE at r                                // Equations (3), (5)
14:    if r ≠ p then
15:        Orthogonalise x_r with respect to r − p             // Definition 27
16:        Compute (x|A|x)_{r|p} and (x|b)_{r|p}               // Equation (6)
17:    end if
18: end function
```

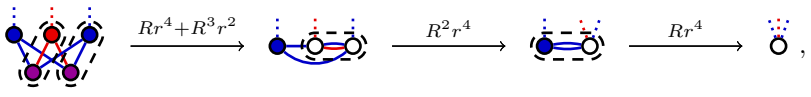
We state already here that the costs at the root are always negligible such that we do not have to discuss this special case repeatedly.

Orthogonalisation requires some constant number of QR decompositions, SVDs and mode multiplications per vertex, each of which costs $\mathcal{O}(nr^2)$ for leaves and $\mathcal{O}(r^4)$ for interior vertices, leading to a total cost of $\mathcal{O}(dr^4+dnr^2)$ FLOP. See also [7, Lemma 4.8] for a more detailed result regarding the cost of the strong r-orthogonalisation Algorithm 2.

The recursive computation of the **contracted subtrees** $(x|A|x)_{v|r}$ with $v \in \text{interior}(r)$, $r \in V$ and $\text{child}(v \mid r) = \{v_L, v_R\}$ requires evaluating

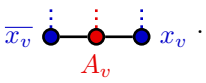


(see Remark 13 for an introduction to tensor network diagrams). As in Figure 3, vertices and edges of x are shown in blue and those of A in red. We propose to contract this network according to the sequence[3]



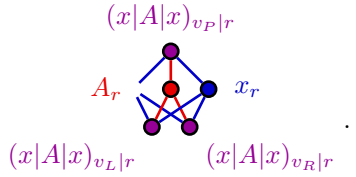
[3]The Matlab script from [25] was of great use to determine such contraction sequences.

which costs $\mathcal{O}(R^2r^4)$ FLOP. At the leaf vertices $v \in \text{leaf}$, the network to contract is given by

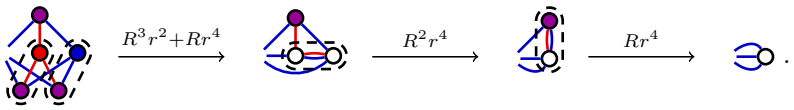


and can be evaluated in $\mathcal{O}(n^2Rr)$ FLOP. Both of these operations are invoked a constant number of times per vertex, thus computing the contracted subtrees $(x|A|x)_{v|r}$ costs $\mathcal{O}\left(dR^2r^4 + dn^2Rr\right)$ in total. The costs for computing $(x|b)_{v|r}$ are obtained similarly. We only state the final result, which is $\mathcal{O}\left(dR_br^3 + dnR_br\right)$.

The **local LSEs** are best solved using an iterative method like conjugate gradient or GMRES because on the one hand, the old, to-be-replaced vertex tensor often provides a good initial guess for the new, replacing one, and on the other hand, the special structure of the local matrix allows for an efficient matrix-vector product. To make the second point more concrete, we consider the matrix-vector product at an interior vertex $r \in \text{interior}(r^\star)$ with $\text{neighbour}(r) = \{v_P, v_L, v_R\}$, which amounts to contracting



see also Figure 3b. This can be done in $\mathcal{O}(R^2r^4)$ FLOP through



Similarly, we find the cost of the matrix-vector product at the leaves to be $\mathcal{O}(n^2Rr)$. The right hand side can be computed in $\mathcal{O}(R_br^3)$ (interior vertex) and $\mathcal{O}(nR_br)$ (leaf), respectively, which yields a total cost for solving the local LSE of

$$\mathcal{O}\left(d\,\rho\left(R^2r^4 + n^2Rr\right) + d\left(R_br^3 + nR_br\right)\right),$$

where ρ denotes the number of steps per local LSE required by the iterative solver.

4.4. The ALS(SD) Algorithm. The ALS algorithms presented above do not adapt the ranks of the iterate and therefore fail to produce a reasonably accurate solution if the initial ranks are chosen too small, but become unnecessarily costly if these ranks are overestimated. The ALS(SD) algorithm from [4] allows to easily endow the ALS scheme with rank-adaptivity by extending it with a steepest descent (SD) and a truncation step as follows.

Algorithm 5 ALS(SD) Algorithm

1: **repeat**
2: Compute residual approximation $z \approx b - Ax$
3: Update $x := x + z$
4: Run a single ALS iteration (Algorithm 3 or 4)
5: Truncate x
6: **until** convergence

In the numerical experiments presented below, the residual approximation is computed using a single iteration of the parallel ALS Algorithm 4 with fixed uniform rank $n_e(z) = 3$ for all $e \in E$ applied to the system $\mathbb{I}_D\, z = b - Ax$, see [4] for details. For the truncation step, we use the algorithm from [10, §11.4.2.1] choosing the ranks adaptively such that the original and truncated tensors $x, \tilde{x} \in \mathrm{HTR}(V, E, D)$ satisfy $\frac{\|x-\tilde{x}\|}{\|x\|} \leq \varepsilon$ with $\varepsilon \in \mathbb{R}_{>0}$ a user-specified tolerance parameter. We call the ALS(SD) algorithm *serial* if the ALS algorithm on line 4 is the serial Algorithm 3, and *parallel* if the algorithm in question is the parallel Algorithm 4.

4.5. Numerical Experiments. We investigate the numerical properties of the above algorithms by means of the d-dimensional Poisson equation $-\Delta u = 1$ on $[0, 1]^d$ with homogeneous Dirichlet boundary conditions, discretised according to the standard finite difference scheme on a uniform mesh with $n = 2^6$ interior grid points in each dimension. This linear system of equations is *quantised* [27, 20, 21, 15] into 6 virtual modes of length 2 each, and the resulting $6d$ modes are organised into a mode tree by first constructing a balanced standard tree for the d physical modes and then replacing each leaf in this tree with a balanced standard tree for the 6 virtual modes of the respective dimension. Further details about the numerical experiments are given at the end of this subsection.

In a first test, we employ both the serial and parallel ALS and ALS(SD) algorithms on a single core to check whether the modifications required for parallelisation have any impact on the convergence of the algorithms. As shown in Figure 6, this is not the case.

Next, we investigate the strong scaling of the parallel ALS(SD) algorithm. From Figure 7 we conclude:

- Theorem 23 allows to predict the parallel scaling with reasonable accuracy if the number of leaves d is replaced with an *effective* d taking into account that the vertex tensors near the leaves are smaller than the ones near the root and the mode tree is not perfectly balanced because d is not a power of 2.
- The scalability decreases with increasing iteration count of the ALS(SD) algorithm. This is because in the first iteration, the ranks are almost uniform ($2 \leq r \leq 4$) while in later iterations the ranks become increasing towards the root ($r \leq 13$).

Technical Details. All benchmarks were run on two twelve-core AMD Opteron 6174 processors (2.2 GHz). The implementation is based on MPI and uses asynchronous point-to-point communication for all messages except the exchange of some norms which is implemented in terms of a broadcast. The algorithm generating the vertex distributions is described in section S1. The runtimes per vertex were assumed uniform such that $\mathrm{depth}_t(v \mid r^\star) \propto \mathrm{depth}(v \mid r^\star)$ and $\mathrm{height}_t(v \mid r^\star) \propto \mathrm{height}(v \mid r^\star)$. The local LSEs are solved using the conjugate gradient algorithm, terminating the iterations once the relative local residual drops below 10^{-10} or the iteration count reaches the dimension of the LSE. We use an HTR network of the indicated ranks and with random vertex tensors as initial guess for the ALS methods, and the right-hand side, i.e. the all-ones tensor, for the ALS(SD) algorithms. "Relative residual" refers to $\frac{\|b-Ax\|}{\|b\|}$ where A, b denote the (global) coefficient matrix and right hand side and x the current iterate. The markers in Figure 6 refer to the value of this quantity after an ALS/ALS(SD) iteration has been completed.

5. Conclusion. We have seen that the TT ALS algorithm for solving linear systems of equations from [13, 19] is not suitable for parallelisation because algorithms

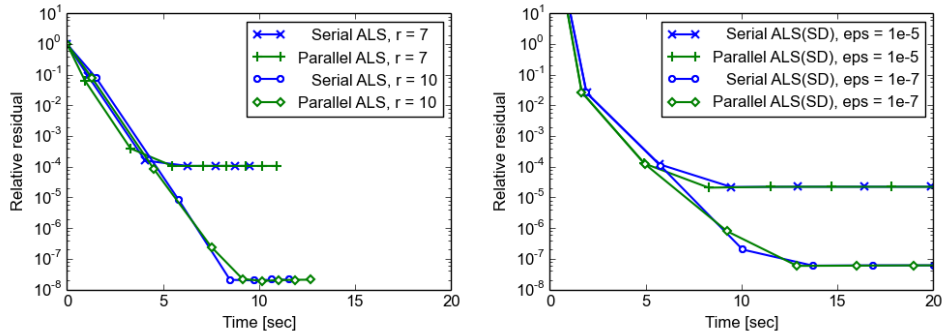
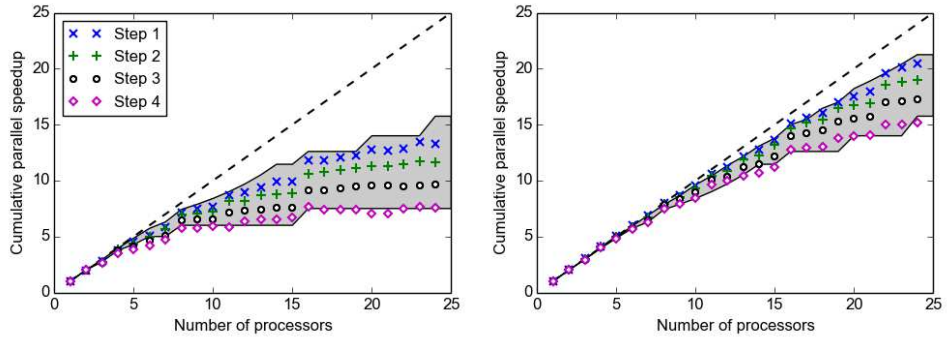


Fig. 6: Convergence of the serial and parallel ALS and ALS(SD) algorithms applied to the 128-dimensional Poisson equation. r denotes the rank of the iterate, `eps` the relative truncation tolerance. Both algorithms were run on a single core.



(a) $d_p = 32$ (b) $d_p = 128$

Fig. 7: Strong scaling of the ALS(SD) solver applied to the Poisson equation with $d_p = 32$ (left) and $d_p = 128$ (right) physical dimensions and relative truncation tolerance $\varepsilon = 10^{-7}$. "Cumulative parallel speedup" is computed as $\frac{T(1)}{T(p)}$ where $T(p)$ is the wall clock time up to and including the indicated ALS(SD) iteration on p processors. The dashed lines denote perfect speedup, the upper end of the grey area the optimal speedup from Theorem 23 for $d = 4d_p$ and the lower end the optimal speedup for $d = d_p$. The method converges in four steps for both $d_p = 32$ as well as $d_p = 128$.

based on the TT format are generally only poorly parallelisable, and the ALS algorithm in particular is not parallelisable even when based on different formats. We therefore presented a novel ALS algorithm overcoming both of these constraints and found it to allow for $\mathcal{O}\big(\frac{d}{\log(d)}\big)$ parallel speedup (d denoting the number of dimensions). This on the one hand means the parallel algorithm enjoys asymptotically almost optimal (up to the logarithmic factor) weak scaling in the limit $d \to \infty$. On the other hand, it implies we need very high-dimensional problems in order to scale

the algorithm to large processor counts: we have seen it already takes $6 \times 128 = 768$ dimensions or $2^{768} \approx 10^{231}$ unknowns to get decent scaling up to only 24 cores.

Another approach to parallelisation is to distribute the workload associated to a single vertex, as has been done in [2, 12, 18]. In the generic setting considered here, this boils down to parallelising the dense linear algebra operations at the vertex level, most importantly the `gemm` underlying the mode product. Such a parallelisation scheme avoids the dimensionality problem discussed above but often amounts to parallelising a small inner loop nested in a large outer one in which case it will also not scale well.

The two approaches are complementary, and we expect most applications will require combining their respective merits. In such a setup, the algorithm developed here allows for important speedup *in addition* to the parallel potential found at other levels in the software stack, in particular in the ideal case of low ranks and high dimensions where the algorithms are hardly parallelisable otherwise.

Acknowledgements. This paper originated from the master thesis [5]. I would like to thank Vladimir Kazeev for his advice on tensor network formats and the related linear solvers, and Robert Gantner for his support regarding the implementation.

REFERENCES

[1] B. W. Bader and T. G. Kolda, *Algorithm 862: Matlab tensor classes for fast algorithm prototyping*, ACM Trans. Math. Softw., 32 (2006), pp. 635–653, doi:10.1145/1186785.1186794.

[2] G. K.-L. Chan, *An algorithm for large scale density matrix renormalization group calculations*, J. Chemical Physics, 120 (2004), pp. 3172–3178, doi:10.1063/1.1638734.

[3] L. De Lathauwer, B. De Moor, and J. Vandewalle, *A multilinear singular value decomposition*, SIAM J. Matrix Anal. Appl., 21 (2000), pp. 1253–1278, doi:10.1137/S0895479896305696.

[4] S. Dolgov and D. Savostyanov, *Alternating minimal energy methods for linear systems in higher dimensions*, SIAM J. Sci. Comput., 36 (2014), pp. A2248–A2271, doi:10.1137/140953289.

[5] S. Etter, *Parallel tensor-formatted numerics for the chemical master equation*, master's thesis, Seminar of Applied Mathematics, ETH Zurich, 2015, https://www.math.ethz.ch/sam/studies/student-projects-and-theses.html.

[6] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, USA, third ed., 1996.

[7] L. Grasedyck, *Hierarchical singular value decomposition of tensors*, SIAM J. Matrix Anal. Appl., 31 (2009/10), pp. 2029–2054, doi:10.1137/090764189.

[8] L. Grasedyck, *Polynomial approximation in hierarchical Tucker format by vector-tensorization*, Tech. Report 308, IGMP, RWTH Aachen, April 2010, http://www.igpm.rwth-aachen.de/forschung/preprints/308.

[9] L. Grasedyck, D. Kressner, and C. Tobler, *A literature survey of low-rank tensor approximation techniques*, GAMM-Mitt., 36 (2013), pp. 53–78, doi:10.1002/gamm.201310004.

[10] W. Hackbusch, *Tensor spaces and numerical tensor calculus*, Springer Verlag, Heidelberg, 2012, doi:10.1007/978-3-642-28027-6.

[11] W. Hackbusch and S. Kühn, *A new scheme for the tensor representation*, J. Fourier Anal. and Appl., 15 (2009), pp. 706–722, doi:10.1007/s00041-009-9094-9.

[12] G. Hager, E. Jeckelmann, H. Fehske, and G. Wellein, *Parallelization strategies for density matrix renormalization group algorithms on shared-memory systems*, J. Computational Physics, 194 (2004), pp. 795 – 808, doi:10.1016/j.jcp.2003.09.018.

[13] S. Holtz, T. Rohwedder, and R. Schneider, *The alternating linear scheme for tensor optimization in the tensor train format*, SIAM J. Sci. Comput., 34 (2012), pp. A683–A713, doi:10.1137/100818893.

[14] T. C. Hu, *Parallel sequencing and assembly line problems*, Operations Research, 9 (1961), pp. 841–848.

[15] B. N. Khoromskij, *$O(d \log N)$-quantics approximation of N-d tensors in high-dimensional numerical modeling*, Constr. Approx., 34 (2011), pp. 257–280, doi:10.1007/s00365-011-9131-1.

[16] B. N. Khoromskij, *Tensors-structured numerical methods in scientific computing: survey on recent advances*, Chemometr. Intell. Lab. Syst., 110 (2012), pp. 1 – 19, doi:10.1016/j.

chemolab.2011.09.001.

[17] D. Kressner and C. Tobler, *Preconditioned low-rank methods for high-dimensional elliptic PDE eigenvalue problems*, Comput. Methods Appl. Math., 11 (2011), pp. 363–381, doi:10.2478/cmam-2011-0020.

[18] Y. Kurashige and T. Yanai, *High-performance ab initio density matrix renormalization group method: Applicability to large-scale multireference problems for metal compounds*, J. Chemical Physics, 130 (2009), 234114, doi:10.1063/1.3152576.

[19] I. Oseledets and S. Dolgov, *Solution of linear systems and matrix inversion in the TT-format*, SIAM J. Sci. Comput., 34 (2012), pp. A2718–A2739, doi:10.1137/110833142.

[20] I. V. Oseledets, *Approximation of matrices with logarithmic number of parameters*, Doklady Mathematics, 80 (2009), pp. 653–654, doi:10.1134/S1064562409050056.

[21] I. V. Oseledets, *Approximation of $2^d \times 2^d$ matrices using tensor decomposition*, SIAM J. Matrix Anal. Appl., 31 (2010), pp. 2130–2145, doi:10.1137/090757861.

[22] I. V. Oseledets, *Tensor-train decomposition*, SIAM J. Sci. Comput., 33 (2011), p. 2295–2317, doi:10.1137/090752286.

[23] I. V. Oseledets, *Constructive representation of functions in low-rank tensor formats*, Constructive Approximation, 37 (2013), pp. 1–18, doi:10.1007/s00365-012-9175-x.

[24] I. V. Oseledets and E. E. Tyrtyshnikov, *Breaking the curse of dimensionality, or how to use SVD in many dimensions*, SIAM J. Sci. Comput., 31 (2009), pp. 3744–3759, doi:10.1137/090748330.

[25] R. N. C. Pfeifer, J. Haegeman, and F. Verstraete, *Faster identification of optimal contraction sequences for tensor networks*, Phys. Rev. E, 90 (2014), p. 033315, doi:10.1103/PhysRevE.90.033315.

[26] E. M. Stoudenmire and S. R. White, *Real-space parallel density matrix renormalization group*, Phys. Rev. B, 87 (2013), p. 155137, doi:10.1103/PhysRevB.87.155137.

[27] E. E. Tyrtyshnikov, *Tensor approximations of matrices generated by asymptotically smooth functions*, Sbornik: Mathematics, 194 (2003), pp. 941–954, doi:10.1070/SM2003v194n06ABEH000747.

SUPPLEMENTARY MATERIALS: PARALLEL ALS ALGORITHM FOR SOLVING LINEAR SYSTEMS IN THE HIERARCHICAL TUCKER REPRESENTATION

SIMON ETTER

S1. Vertex Distribution Algorithm. This section describes the vertex distribution algorithm used in the scaling studies in Figure 7.

We again assume all operations are associated with vertices rather than endpoints of edges, the mode tree is standard and balanced with $d := \#D$, and the computational costs per interior vertex are uniform, cf. Assumption 21. The proof of Theorem 23 revealed the following conditions for a vertex distribution to allow for optimal parallel scaling under these assumptions.

- Down to depth $t(p) := \lceil \log_2 p \rceil - 1$, all vertices at the same depth must be assigned to different processors.
- The vertices from depth $t(p)+1$ onwards must be evenly distributed over the processors.

The new points addressed by our algorithm are that the communication costs and the processing times for the root and leaves may be non-negligible. We therefore minimise the former by assigning as many neighbouring vertices to the same processor as possible without violating the above constraints, and distribute the latter by making sure we deal each processor an equal share of leaf vertices.

These considerations are reflected in the following features of our algorithm.

- Rather than assigning individual vertices, we deal entire *branches* at the time. Given a vertex $v \in V$, the *branch starting at vertex* v, branch(v), is the set assembled by the following algorithm.

```
1: branch(v) := {}
2: u := v
3: while child(u | r*) ≠ {} do
4:     branch(v) := branch(v) ∪ {u}
5:     u := right child(u | r*)
6: end while
7: branch(v) := branch(v) ∪ {u}
```

  We assume here the two children of a non-leaf vertex v are arbitrarily marked as left and right child, respectively.
- We introduce two counters n_interior(q) and n_branch(q) to keep track of the number of interior vertices at depths larger than $t(p)$ and the number of branches (or equivalently, the number of leaves) assigned to processor q.

The technical details incurred by dealing a branch and updating the counters are taken care of by the DealBranch function in Algorithm S1.

To describe the final algorithm, we index the vertices $v(x, y)$ with two coordinates x indicating the depth and y enumerating the vertices at depth x from left to right, starting at 0. We write $x_{\max}$ to denote the largest value x can assume and denote by $y_{\max}(x)$ the number of vertices at depth x. It will turn out that when we reach level x, all vertices with odd y-coordinate are already assigned to a processor such that only the even vertices remain to be considered. We employ the Matlab notation $y = 0 : 2 : y_{\max}(x) - 1$ to restrict the y-coordinate to this range, and similarly write $x = a : b$ to indicate x ranges from a to b in unit increments.

Our vertex distribution algorithm, shown in Algorithm S1, proceeds in two stages. First (lines 13 to 20), we assign all branches starting at depths $\leq t(p)$ to the first $2^{t(p)} < p$ processors. Then (lines 22 to 38), we assign the branches $\mathrm{branch}(v)$ with $\mathrm{depth}(v \mid r^\star) > t(p)$ to the processor owning the parent vertex $\mathrm{parent}(v \mid r^\star)$ if it *fits*, i.e. if neither of the following conditions are satisfied.

- Denote by n the total number of interior vertices at depths below $t(p)$. To equidistribute the workload each processor should receive either $\lfloor \frac{n}{p} \rfloor$ or $\lceil \frac{n}{p} \rceil$ such vertices, with a fixed number $n \bmod p$ of processors in the latter category. Thus, a branch $\mathrm{branch}(v)$ does not fit on a processor q if $\mathrm{n\_interior}(q) + \#\,\mathrm{branch}(v) - 1$ (-1 because we ignore the leaf) is either strictly larger than $\lceil \frac{n}{p} \rceil$ or it is equal and we have already assigned that many interior vertices to some other $n \bmod p$ processors.
- Assigning too many vertices at a given depth to the same processor q may unnecessarily force other processors to wait for messages from q and thus increase the overall execution time. To avoid this scenario, we also equidistribute the vertices at a fixed depth, that is we do not assign $\mathrm{branch}(v)$ at depth $x = \mathrm{depth}(v \mid r^\star)$ to processor q if $\mathrm{n\_branch}(q) + 1$ is either strictly larger than $\lceil \frac{y_{\max}(x)}{p} \rceil$ or it is equal and we have already assigned that many branches to $y_{\max}(x) \bmod p$ other processors.

If $\mathrm{branch}(v)$ does not fit on $q(\mathrm{parent}(v \mid r^\star))$, we assign the branch to the processor owning the least number of either interior vertices or branches, depending on whether $\mathrm{branch}(v)$ includes interior vertices or not. See Algorithm S1 for details.

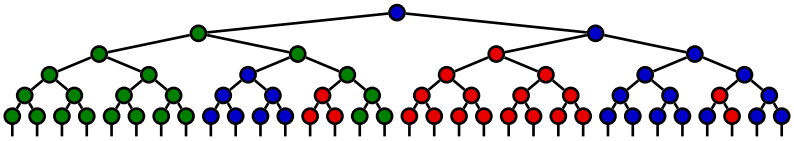
Some example vertex distributions generated by this algorithm are shown in Figure S1.

Algorithm S1 Vertex Distribution Algorithm

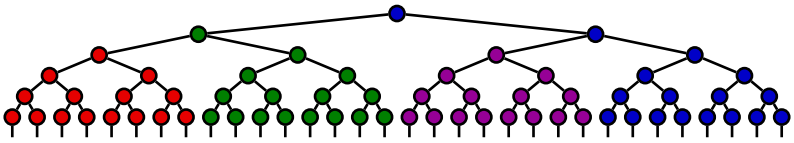
n_interior(q) := 0 and n_branch(q) := 0 for all $q = 0 : p-1$
function DealBranch(v, q)
  while child($v \mid r^\star$) $\neq \{\}$ **do**
    $q(v) := q$
    if depth($v \mid r^\star$) $> t(p)$ **then**
      n_interior(q) := n_interior(q) + 1
    end if
    $v :=$ right child($v \mid r^\star$)
  end while
  $q(v) := q$
  n_branch(q) := n_branch(q) + 1
end function

// Deal branches starting at depths $\leq t(p)$
$q := 0$
for $x = 0 : t(p)$ **do**
  for $y = 0 : 2 : y_{\max}(x) - 1$ **do**
    DealBranch($v(x, y), q$)
    $q := q + 1$
  end for
end for

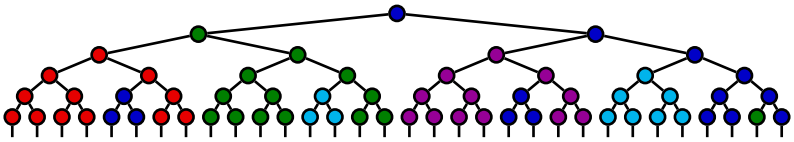
// Deal branches starting at depths $> t(p)$
for $x = t(p) + 1 : x_{\max}$ **do**
  unassigned := $\{\}$
  for $y = 0 : 2 : y_{\max}(x) - 1$ **do**
    if $v(x, y)$ fits on q(parent($v(x, y) \mid r^\star$)) (see text) **then**
      DealBranch($v(x, y), q$(parent($v(x, y) \mid r^\star$))
    else
      unassigned := unassigned $\cup \{v(x, y)\}$
    end if
  end for
  for $v \in$ unassigned **do**
    if child(v) $\neq \{\}$ **then**
      DealBranch($v, \arg\min_{q=0:p-1}$ n_interior(q))
    else
      DealBranch($v, \arg\min_{q=0:p-1}$ n_branch(q))
    end if
  end for
end for



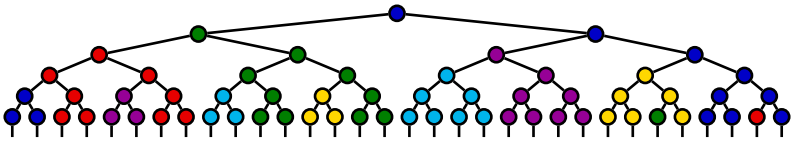
(a) $p = 3$



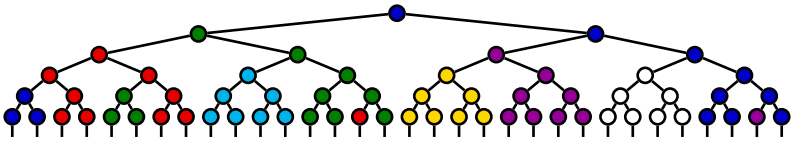
(b) $p = 4$



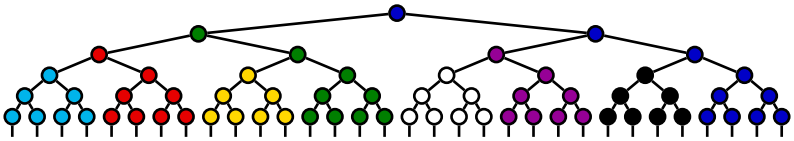
(c) $p = 5$



(d) $p = 6$



(e) $p = 7$



(f) $p = 8$

Fig. S1: Vertex distributions generated by Algorithm S1 for $d = 32$ and p as indicated. The colors indicate the processors to which the vertices have been assigned to.