ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Seminar for Applied Mathematics SAM

Parallel ALS Algorithm for the Hierarchical Tucker Representation

S. Etter

Research Report No. 2015-25
August 2015

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

PARALLEL ALS ALGORITHM FOR 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 [9, 5] or its simpler special case, the *tensor train* (TT) format from [19, 18], 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. [6, 11, 15]. We further refer to [8] for a theoretical introduction and to [12, 7] 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 [10, 16] and its various extensions like the *density matrix renormalisation group* (DMRG) algorithm from [10, 16] or the *ALS + steepest descent* (ALS(SD)) and *alternating minimal energy* (AMEn) algorithms from [3] are amongst 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 [18, 5], 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 dimension partition 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 for the DMRG case in [21]. In the mental picture given above, the parallelisation scheme proposed there exploits that information needs to be exchanged repeatedly in the DMRG algorithm, 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-point of the TT chain.

The remainder of this paper is organised as follows. Section 2 introduces the terms and notation necessary for our presentation, and in Section 3 we discuss the parallel implementation and analyse the parallel scaling for a fairly general class of HTR algorithms. Section 4 presents the above-mentioned parallelisation scheme in the particular setting of the ALS algorithm, but the (straightforward) extension to the ALS(SD) algorithm will also be discussed and numerically investigated.

2. Terminology and Notation.

2.1. Tensors. The definition of tensors used in this paper is based on a generalised concept of tuples obtained as follows. Let D be a finite but otherwise 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 $t^{(1)} \in \bigtimes_{k\in D^{(1)}} A_k$, $t^{(1)} \in \bigtimes_{k\in D^{(2)}} A_k$ 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 $t := t^{(1)} \times t^{(2)}$. If $D := \{k\}$ is a singleton, we set $\bigtimes_{\ell\in D} A_\ell = A_k$, i.e. we do not distinguish between tuples of length one and their single elements.

A *tensor* x 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 with elements from $[n_k] := \{0, \dots, n_k - 1\}$ for some $n_k \in \mathbb{N}$ which are themselves indexed by $k \in D$. 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. For notational convenience, we further write $x(i_D)$ instead of x_{i_D} to denote the evaluation of x at i_D. We define the addition and scalar multiplication of tensors to be element-wise,

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

for all $x, y \in \mathbb{K}^{\bigtimes_{k\in D}[n_k]}$ and $\alpha \in \mathbb{K}$. The inner product and norm on $\mathbb{K}^{\bigtimes_{k\in D}[n_k]}$ 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 $z \in \mathbb{R}$.

Tensors can be reshaped to matrices just like matrices can be reshaped to vectors. We use the symbol $\mathcal{M}_{R,C}(x)$ to refer to such a *matricisation* [5] (also called *matrix unfolding* [19] or *flattening*) of a tensor $x \in \mathbb{K}^{\times_{k\in R\cup C}[n_k]}$ where the modes in the set R go into the rows and the modes in C into the columns of the resulting matrix . See also [8, §5.2] for a more rigorous definition of this operation.

The *mode product* defined next is a generalisation of the matrix product for tensors. Let $x \in \mathbb{K}^{\times_{k\in M\cup K}[n_k]}$, $y \in \mathbb{K}^{\times_{k\in K\cup N}[n_k]}$ be two tensors such that M, K and N are pairwise disjoint. The expression xy defines a new tensor $z \in \mathbb{K}^{\times_{k\in M\cup N}[n_k]}$ 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 is equivalent to the tensor product [8, §1.1.1]. The same operation has already been introduced in [1] under the name "tensor-times-tensor" (`ttt`) product, but because there the tensor modes are enumerated instead of labelled the notation is more complicated than what we propose here.

If we interpret a square matrix as a linear operator on $\mathbb{K}^{[n_k]}$, 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 $R(k)$ and $C(k)$ with $n_{R(k)} := n_{C(k)} := n_k$ called *row* and *column mode* of k, respectively. Further, we define $[n_k]^2 := [n_{R(k)}] \times [n_{C(k)}]$ and $\mathcal{M}(x) := \mathcal{M}_{R(D),C(D)}(x)$ for a tensor $x \in \mathbb{K}^{\times_{k\in D}[n_k]^2}$ containing only squared modes. Multiplication with row/column modes follows special rules which generalise the rules of the matrix product: a column mode $C(k)$ is only multiplied with a k- or $R(k)$-mode appearing to the right of the tensor carrying the $C(k)$-mode, and similarly, a row mode $R(k)$ is only multiplied with a k- or $C(k)$-mode appearing to the left. If in the resulting tensor there is only either a row mode $R(k)$ or a column mode $C(k)$ present, we rename it to k. The simple matrix-vector product Ax with $A \in \mathbb{K}^{[n_k]^2}$ and $x \in \mathbb{K}^{[n_k]}$ is thus to be read as follows. Because x stands to the right of A, the k-mode of x is multiplied with the $C(k)$-mode of A, yielding an intermediate result in $\mathbb{K}^{[n_{R(k)}]}$ whose $R(k)$-mode is then renamed to simply k since it appears without an accompanying $C(k)$-mode. We thus get a final result in $\mathbb{K}^{[n_k]}$ as expected.

Given a tensor $x \in \mathbb{K}^{\times_{k\in D}[n_k]}$ and some set $M \subseteq D$, we define the expression $x\langle M$ to upgrade the M-modes of x to $R(M)$-modes, and similarly $M\rangle x$ upgrades the M-modes to $C(M)$-modes. If a $\langle M$ appears next to a $M\rangle$, we merge the two symbols into $\langle M\rangle$. The main application of this notation is to exclude certain modes from being multiplied. For example, we must write $x \,\langle\{k\}\rangle\, y \in \mathbb{K}^{[n_k]^2}$ to denote the outer product of two vectors $x, y \in \mathbb{K}^{[n_k]}$, whereas dropping the $\langle\{k\}\rangle$ as in $xy \in \mathbb{K}$ yields their inner product up to conjugation of x in the complex case.

We conclude this subsection with a number of technical definitions.

Definition 2.1 (Identity Tensor). *Let D be some mode set. The* identity tensor $\mathbb{I}_D$ *is the tensor in $\mathbb{K}^{\times_{k\in D}[n_k]^2}$ such that $\mathbb{I}_D\, x = x$ for all $x \in \mathbb{K}^{\times_{k\in D}[n_k]}$.*

Definition 2.2 (Transposed Tensor). *Let $A \in \mathbb{K}^{(\times_{k\in D}[n_k])\times(\times_{k\in S}[n_k]^2)}$ be a tensor with some squared modes S. The symbol A^T denotes the tensor in the same space $\mathbb{K}^{(\times_{k\in D}[n_k])\times(\times_{k\in S}[n_k]^2)}$ where the $R(k)$- and $C(k)$-modes are interchanged for each $k \in S$. The non-squared modes D are not modified.*

Definition 2.3 (Inverse Tensor). *Let $A \in \mathbb{K}^{\times_{k\in D}[n_k]^2}$ be a tensor operator. The*

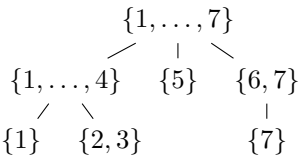


Figure 2.1: Example dimension partition tree for $D = \{1, \ldots, 7\}$.

inverse tensor A^{-1} *is the unique tensor in* $\mathbb{K}^{\times_{k\in D}[n_k]^2}$ *such that* $AA^{-1} = \mathbb{I}_D$.

DEFINITION 2.4 (Orthogonal Tensor). *Let* $x \in \mathbb{K}^{\times_{k\in D}[n_k]}$ *be a tensor and* $M \subset D$ *a mode set.* x *is called* M-orthogonal *if* $x \langle M \rangle x = \mathbb{I}_M$.

DEFINITION 2.5 (Tensor Orthogonalisation). *Let* $x \in \mathbb{K}^{\times_{k\in D}[n_k]}$ *be a tensor and* $k \in D$ *some mode such that* $\prod_{\ell\in D\setminus\{k\}} n_\ell \geq n_k$. *The symbol* $\mathcal{Q}_k(x)$ *denotes a pair* $(q \in \mathbb{K}^{\times_{\ell\in D}[n_\ell]}, r \in \mathbb{K}^{[n_k]^2})$ *of tensors such that* $x = qr$ *and* q *is* $\{k\}$*-orthogonal.*

REMARK 2.6. *Tensor orthogonalisation can be implemented by computing the QR decomposition* $QR := \mathcal{M}_{D\setminus\{k\},\{k\}}(x)$ *and setting* $\mathcal{M}_{D\setminus\{k\},\{k\}}(q) := Q$ *and* $\mathcal{M}(r) := R$. *This procedure costs* $\mathcal{O}\left(n_k^2 \prod_{\ell\in D\setminus\{k\}} n_\ell\right)$ *floating-point operations [4, §5.2.1].*

DEFINITION 2.7 (Tensor SVD [2]). *Let* $x \in \mathbb{K}^{\times_{k\in D}[n_k]}$ *be a tensor and* $k \in D$ *some mode such that* $\prod_{\ell\in D\setminus\{k\}} n_\ell \geq n_k$. *The* tensor SVD $\mathcal{S}_k(x)$ *denotes a triplet*

$$\left(u \in \mathbb{K}^{\times_{\ell\in D}[n_\ell]},\ s \in \mathbb{K}^{[n_k]^2},\ v \in \mathbb{K}^{[n_k]^2}\right)$$

such that $x = usv$, $\mathcal{M}(s)$ *is diagonal and* u *and* v *are* $\{k\}$*- and* $\{R(k)\}$*-orthogonal, respectively. The diagonal entries of* s*, denoted by* s_{i_k}*, are real and non-negative. They are called* singular values.

REMARK 2.8. *The tensor SVD can be implemented by computing the matrix SVD* $U\Sigma V^* := \mathcal{M}_{D\setminus\{k\},\{k\}}(x)$ *and setting* $\mathcal{M}_{D\setminus\{k\},\{k\}}(u) := U$, $\mathcal{M}(s) := \Sigma$ *and* $\mathcal{M}(v) := V^*$ *(note the adjoint). This procedure costs* $\mathcal{O}\left(n_k^2 \prod_{\ell\in D\setminus\{k\}} n_\ell\right)$ *floating-point operations [4, §5.4.5].*

2.2. Hierarchical Tucker Representation. Our presentation in later sections is based on the following definitions of *dimension partition trees* and the *hierarchical Tucker representation* (HTR) [5, 8].

DEFINITION 2.9 (Dimension Partition Tree). *Let* D *be some mode set and* $\mathcal{P}(D)$ *its power set, i.e. the set of all subsets of* D. *A set* $T_D \subset \mathcal{P}(D)$ *is called a* dimension partition tree *if it satisfies* $D \in T_D$, $\{\} \notin T_D$ *and* $\alpha \cap \beta \neq \{\} \implies \alpha \subseteq \beta \vee \beta \subseteq \alpha$ *for all* $\alpha, \beta \in T_D$. *The last condition establishes the hierarchical structure of* T_D *such that* T_D *can be visualised as a tree with the set* D *being its root, see Figure 2.1.*

We use the terms child(α), parent(α'), neighbour(α) and sibling(α') for $\alpha \in T_D$ and $\alpha' \in T_D \setminus \{D\}$ to refer to vertices $\alpha \in T_D$ or sets of vertices $S \subseteq T_D$ according to the usual definitions of these terms. In addition, we define

$$\begin{aligned}
\text{descendant}(\alpha) &:= \{\beta \in T_D \mid \beta \subseteq \alpha\},\\
\text{ancestor}(\alpha) &:= \{\beta \in T_D \mid \alpha \subset \beta\},\\
\text{level}(\alpha) &:= \#\,\text{ancestor}(\alpha),\\
\text{colevel}(\alpha) &:= \max\{\text{level}(\beta) \mid \beta \in \text{descendant}(\alpha)\} - \text{level}(\alpha).\ ^{1}
\end{aligned}$$

The dimension partition tree T_D on which these terms depend is implicitly given as the tree from which α or α' were taken. Furthermore, we define the sets

$$\begin{aligned}\operatorname{leaf}(T_D) &:= \{\alpha \in T_D \mid \operatorname{child}(\alpha) = \{\}\},\\ \operatorname{interior}(T_D) &:= T_D \setminus (\{D\} \cup \operatorname{leaf}(T_D)).\end{aligned}$$

DEFINITION 2.10 (HTR Network). *Let T_D be a dimension partition tree and $r \in \mathbb{N}^{T_D \setminus \{D\}}$ a tuple of integers called* ranks. *For each vertex $\alpha \in T_D$, we define the* edge set $E_\alpha \subseteq T_D \setminus \{D\}$ *and the set of* free modes $D_\alpha \subseteq D$,

$$E_\alpha := \begin{cases}\operatorname{child}(\alpha) & \text{if } \alpha = D,\\ \{\alpha\} \cup \operatorname{child}(\alpha) & \text{otherwise}\end{cases}, \qquad D_\alpha := \alpha \setminus \left(\bigcup_{\beta \in \operatorname{child}(\alpha)} \beta\right).$$

A tuple of tensors $x \in \operatorname{HTR}(T_D, r, n) := \bigtimes_{\alpha \in T_D} \mathbb{K}^{\left(\times_{\epsilon \in E_\alpha}[r_\epsilon]\right) \times \left(\times_{k \in D_\alpha}[n_k]\right)}$ is called an HTR network. *We implicitly map such a network x to the tensor $\prod_{\alpha \in T_D} x_\alpha$ such that we have $\operatorname{HTR}(T_D, r, n) \subseteq \mathbb{K}^{\times_{k \in D}[n_k]}$. Furthermore, we introduce the set of HTR-formatted linear operators*

$$\operatorname{HTR}^2(T_D, r, n) := \bigtimes_{\alpha \in T_D} \mathbb{K}^{\left(\times_{\epsilon \in E_\alpha}[r_\epsilon]\right) \times \left(\times_{k \in D_\alpha}[n_k]^2\right)} \subseteq \mathbb{K}^{\times_{k \in D}[n_k]^2}.$$

Edge modes of two networks $x \in \operatorname{HTR}(T_D, r^{(x)}, n)$, $y \in \operatorname{HTR}(T_D, r^{(y)}, n)$ are considered distinct. In particular, in expressions of the form $x_\alpha\, y_\alpha$ with $\alpha \in T_D$, the modes $\epsilon \in E_\alpha$ are not multiplied. If it is not clear that a mode ϵ belongs to the network x, we clarify by *tagging* ϵ as in $x(\epsilon)$.

In the existing literature (e.g. [8, Definition 11.2]), dimension partition trees are required to satisfy two additional conditions:

- T_D has to be a proper binary tree, i.e. $\#\operatorname{child}(\alpha) \in \{0, 2\}$ for all $\alpha \in T_D$.
- Only the leaf vertices $\alpha \in \operatorname{leaf}(T_D)$ have free modes, and each such leaf vertex has exactly one free mode.

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

$$\max_{\alpha \in \operatorname{leaf}(T_D)} \operatorname{level}(\alpha) - \min_{\alpha \in \operatorname{leaf}(T_D)} \operatorname{level}(\alpha) \leq 1.$$

It will sometimes be useful to use tree terms like child or sibling relative to some temporary root $\alpha \in T_D$ not necessarily equal to D. We will write $\tau(\beta \mid \alpha)$ to denote such *relative* tree relationships, where τ is a template for any element of the set $\{\text{descendant}, \text{ancestor}, \text{child}, \text{parent}, \text{sibling}\}$. For example, in the tree from Figure 2.1 we have $\operatorname{child}(\{1, \ldots, 4\} \mid \{2, 3\}) = \{\{1\}, \{1, \ldots, 7\}\}$ and $\operatorname{sibling}(\{1, \ldots, 7\} \mid \{2, 3\}) = \{\{1\}\}$. Similarly, we write $\uparrow(\beta \mid \alpha)$ to refer to the edge which connects the vertex $\beta \in T_D \setminus \{\alpha\}$ to $\operatorname{parent}(\beta \mid \alpha)$. In case $\alpha = D$, we have $\uparrow(\beta \mid D) = \beta$, and in the tree from Figure 2.1 we e.g. have $\uparrow(\{1, \ldots, 7\} \mid \{2, 3\}) = \{1, \ldots, 4\}$.

3. Parallelisation of HTR Algorithms. Most HTR algorithms exhibit a common algorithmic structure to be pointed out in §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.

[1] Put differently, $\operatorname{colevel}(\alpha)$ is the longest distance from α to any leaf in $\operatorname{descendant}(\alpha)$.

3.1. Common Structure. DEFINITION 3.1 (Tree Traversing Algorithm). *An HTR algorithm running on a dimension partition tree T_D is called a* tree traversing algorithm *if there exists a partially ordered set S of ordered pairs (α, β) involving neighbouring vertices $\alpha, \beta \in T_D$ such that the algorithm can be formulated as follows.*

1: **for** each $(\alpha, \beta) \in S$ **do**
2: On α: Prepare a message m
3: Transfer m from α to β
4: On β: 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 3.2 (Root-to-Leaves Algorithm). *A tree traversing algorithm is called* root-to-leaves *if the set S and the partial order defined thereon are given by*

$$S := \{(\text{parent}(\alpha), \alpha) \mid \alpha \in T_D \setminus \{D\}\},$$
$$(\text{parent}(\alpha), \alpha) \leq (\text{parent}(\beta), \beta) \quad :\Longleftrightarrow \quad \alpha \in \text{ancestor}(\beta) \cup \{\beta\}.$$

DEFINITION 3.3 (Leaves-to-Root Algorithm). *A tree traversing algorithm is called* leaves-to-root *if the set S and the partial order define thereon are given by*

$$S := \{(\alpha, \text{parent}(\alpha)) \mid \alpha \in T_D \setminus \{D\}\},$$
$$(\alpha, \text{parent}(\alpha)) \leq (\beta, \text{parent}(\beta)) \quad :\Longleftrightarrow \quad \alpha \in \text{descendant}(\beta).$$

DEFINITION 3.4 (Parallel Tree Traversing Algorithm). *A tree traversing algorithm is called* parallel *if the preparation and consumption of messages not ordered by the partial order on S can be executed concurrently.*

DEFINITION 3.5 (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 [5] as well as the computation of the inner product are all tree parallel algorithms, and it will be the topic of §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 3.6. *Let $\mathtt{A}$ be a tree parallel algorithm consisting of a single root-to-leaves/leaves-to-root part running on a balanced standard dimension partition tree T_D. We assume:*

- *The operations on a vertex $\alpha \in T_D$ 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 α have finished.*
- *It takes $\mathtt{A}$ one time unit to prepare/consume all messages at an interior vertex $\alpha \in \text{interior}(T_D)$, and no time at the root D or a leaf $\alpha \in \text{leaf}(T_D)$.*
- *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 α" to denote the consumption of all incoming and the preparation of all outgoing messages on vertex $\alpha \in T_D$. The second assumption is derived from the fact that for a standard dimension partition 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 time steps of length one time unit 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 reasonable accurate reference against which the empirically observed scaling can be compared.

LEMMA 3.7 (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 3.6, and let* $T_{\mathtt{A}}(p)$ *denote the optimal runtime of algorithm* `A` *running on* p *processors. Then,* $T_{\mathtt{RtL}}(p) = T_{\mathtt{LtR}}(p)$ *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 $\alpha(t, q)$ which is to be processed, if any. We call such a function $\alpha(t, q)$ a *vertex schedule*. Let $\alpha_{\mathtt{RtL}}(t, q)$ be an optimal vertex schedule for `RtL`, i.e. $\alpha_{\mathtt{RtL}}(t, q)$ is compatible with the constraints from Definition 3.2 and leads to the optimal parallel execution time $T_{\mathtt{RtL}}(p)$. One easily verifies that $\alpha_{\mathtt{LtR}}(t, q) := \alpha_{\mathtt{RtL}}(T_{\mathtt{RtL}}(p) - t, q)$ is a valid vertex schedule for `LtR` and has the same execution time, which proves $T_{\mathtt{RtL}}(p) \geq T_{\mathtt{LtR}}(p)$. Applying the same argument in the opposite direction proves $T_{\mathtt{RtL}}(p) \leq T_{\mathtt{LtR}}(p)$ which yields the claim. ∎

THEOREM 3.8 (Scaling of Tree Parallel Algorithms). *Let* `A` *be an algorithm satisfying Assumption 3.6 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

$$S(p) := \frac{T(1)}{T(p)} = O\left(\frac{d}{\log_2 p + \frac{d}{p}} \right). \tag{3.1}$$

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 $\#\{\alpha \in T_D \mid \operatorname{colevel}(\alpha) = 1\} = \lfloor \frac{d}{2} \rfloor$ which proves the last statement. We therefore assume $p \leq \lfloor \frac{d}{2} \rfloor$ in the remainder of this proof. By Lemma 3.7, it is further sufficient to consider only a root-to-leaves algorithm. We propose the following vertex schedule for this case (see also Figure 3.1): at time $t < t(p) := \lceil \log_2 p \rceil - 1$, pick any $2^{t+1} < p$ processors and let these process the vertices on level $t + 1$ (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 level $t(p) + 1$ 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

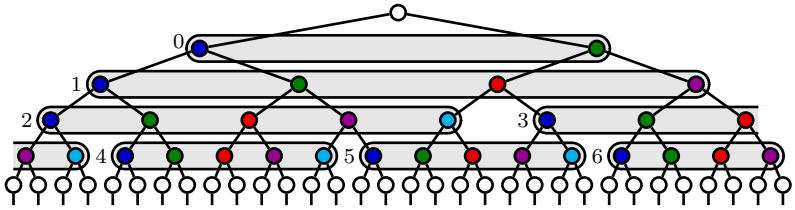


Figure 3.1: Example vertex schedule for $d = 32$, $p = 5$. The colors distinguish between processors and the grey beams group the vertices according to the time step during which they are processed.

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. □

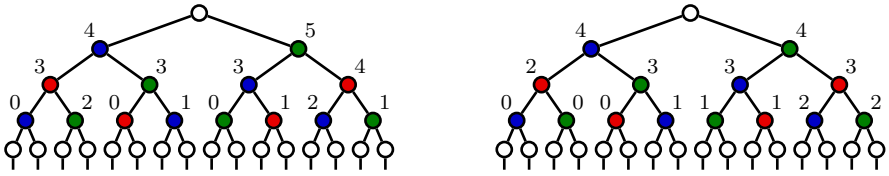
If a tree parallel algorithm consists of more than one root-to-leaves/leaves-to-root part, we assume that the parts must be run one after the other and cannot overlap. Again, this assumption is not necessarily satisfied in practice, but it simplifies the argument and provides a reasonable approximation 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 (3.1) is still valid even in this more general setting.

3.3. Parallel Implementation. The proof of Theorem 3.8 presented a parallelisation scheme for HTR algorithms in the idealised setting of Assumption 3.6. In practice, however, several complications arise such that this scheme may not apply or the optimality guarantee given by Theorem 3.8 may no longer be valid:

- The dimension partition 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 3.1, 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 dimension partition 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 $q(\alpha)$ mapping each vertex $\alpha \in T_D$ to a processor $q(\alpha) \in \{0, \dots, p-1\}$ by whom the jobs associated with vertex α 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



(a) Suboptimal schedule.

(b) Optimal schedule.

Figure 3.2: Vertex distribution for which LPF scheduling may not deliver optimal performance. Let the algorithm in question be leaves-to-root and Assumption 3.6 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.

done on a case-by-case basis. Assigning all jobs of vertex α to the same processor $q(\alpha)$ allows to store all the data associated with α exclusively on processor $q(\alpha)$ and therefore reduces the need for communication.

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

$$\text{level}_t(\alpha) := \sum_{\beta \in \text{ancestor}(\alpha)} t(\alpha)$$

and the *weighted colevel*

$$\text{colevel}_t(\alpha) := \max\{\text{level}_t(\beta) + t(\beta) \mid \beta \in \text{descendant}(\alpha)\} - \text{level}_t(\alpha) - t(\alpha).$$

We further define the *local connected component* $C(\alpha) \subseteq T_D$ of a vertex $\alpha \in T_D$ to be the largest connected component such that $\alpha \in C(\alpha)$ and $q(\beta) = q(\alpha)$ for all $\beta \in C(\alpha)$, and finally set local root$(\alpha) := \arg\min_{\beta \in C(\alpha)} \text{level}(\beta)$.

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 $\alpha \in T_D$ which maximise $\text{level}_t(\text{local root}(\alpha))$.

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

$$\max\{\text{colevel}_t(\beta) + t(\beta) \mid \beta \notin C(\alpha) \wedge \text{parent}(\beta) \in C(\alpha)\}.$$

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 $C \subseteq T_D$ does not matter as the runtime will in any case be $\sum_{\alpha \in C} t(\alpha)$. What does matter, however, is the time t at which q sends the message from the local root β of C to parent(β) since this provides a lower bound $t + \text{level}_t(\beta)$ on the overall runtime. The above algorithm greedily minimises this bound. While LPF scheduling does not necessarily achieve the optimal runtime, see Figure 3.2, 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 [10, 16] tackles the linear system of equations (LSE) $Ax = b$ where the operator $A \in \mathrm{HTR}^2(T_D, r^{(A)}, n)$, the right-hand side $b \in \mathrm{HTR}(T_D, r^{(b)}, n)$ and an initial guess $x \in \mathrm{HTR}(T_D, r, n)$ for the solution $A^{-1}b$ are all represented in HTR. Given such an x and a vertex $\alpha \in T_D$, let us define the *environment tensor*

$$U_\alpha(x) := \prod_{\beta \in T_D \setminus \{\alpha\}} x_\beta \in \mathbb{K}^{\left(\times_{\epsilon \in E_\alpha}[r_\epsilon]\right) \times \left(\times_{k \in D \setminus D_\alpha}[n_k]\right)}.$$

This quantity corresponds to the matrix Q_k from [16] and to the retraction operator $P_{i,1,x}$ from [10]. In its simplest form, the ALS algorithm then reads as follows.

Algorithm 1 ALS Algorithm

1: **repeat**
2: **for** vertex $\alpha \in T_D$ **do**
3: Update x_α to the solution of the *local LSE*

$$\overline{U_\alpha(x)} \, \langle E_\alpha \, A \, E_\alpha \rangle \, U_\alpha(x) \, x_\alpha = \overline{U_\alpha(x)} \, b. \tag{4.1}$$

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 (4.1) can only be solved numerically if the condition number

$$\kappa\left(\mathcal{M}\left(\overline{U_\alpha(x)} \, \langle E_\alpha \, A \, E_\alpha \rangle \, U_\alpha(x)\right)\right)$$

  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 [13], but we need to give further details in order to prepare for the discussion in §4.2.

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

DEFINITION 4.1 (α-Orthogonality). *An HTR network $x \in \mathrm{HTR}(T_D, r, n)$ is called* α-orthogonal *with $\alpha \in T_D$ if $U_\alpha(x)$ is E_α-orthogonal.*

DEFINITION 4.2 (Subtree Tensor). *Let $x \in \mathrm{HTR}(T_D, r, n)$ be an HTR network and $\alpha, \beta \in T_D$ two vertices. We define the* subtree tensor $S_{\beta|\alpha}(x)$ *through*

$$S_{\beta|\alpha}(x) \in \begin{cases} \mathbb{K}^{[r_{\uparrow(\beta|\alpha)}] \times \left(\times_{k \in D(\beta|\alpha)}[n_k]\right)} & \text{if } \beta \neq \alpha \\ \mathbb{K}^{\times_{k \in D}[n_k]} & \text{if } \beta = \alpha \end{cases}, \quad S_{\beta|\alpha}(x) := \prod_{\gamma \in \mathrm{descendant}(\beta|\alpha)} x_\gamma.$$

Here, $D(\beta \mid \alpha)$ denotes the free modes in the subtree of β relative to the root α, i.e.

$$D(\beta \mid \alpha) = \begin{cases} D \setminus \mathrm{parent}(\beta \mid \alpha) & \text{if } \beta \in \mathrm{ancestor}(\alpha) \\ \beta & \text{otherwise} \end{cases}.$$

DEFINITION 4.3 (Strong α-Orthogonality). *An HTR network $x \in \mathrm{HTR}(T_D, r, n)$ is called* strongly α-orthogonal *if all subtree tensors $S_{\beta|\alpha}(x)$ with $\beta \in T_D \setminus \{\alpha\}$ are $\{\uparrow(\beta \mid \alpha)\}$-orthogonal.*

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

$$U_\alpha(x) = \prod_{\beta \in \mathrm{neighbour}(\alpha)} S_{\beta|\alpha}(x),$$

and by the $\{\uparrow(\beta \mid \alpha)\}$-orthogonality of the $S_{\beta|\alpha}(x)$ we have

$$\begin{aligned}
\overline{U_\alpha(x)} \,\langle E_\alpha \rangle\, U_\alpha(x) &= \overline{\left(\prod_{\beta \in \mathrm{neighbour}(\alpha)} S_{\beta|\alpha}(x) \right)} \,\langle E_\alpha \rangle\, \left(\prod_{\beta \in \mathrm{neighbour}(\alpha)} S_{\beta|\alpha}(x) \right) \\
&= \prod_{\beta \in \mathrm{neighbour}(\alpha)} \overline{S_{\beta|\alpha}(x)} \,\langle \{\uparrow(\beta|\alpha)\} \rangle\, S_{\beta|\alpha}(x) \\
&= \prod_{\beta \in \mathrm{neighbour}(\alpha)} \mathbb{I}_{\{\uparrow(\beta|\alpha)\}} = \mathbb{I}_{E_\alpha}.
\end{aligned}$$

If x is α-orthogonal and A Hermitian, Theorem 4.1b) in [10] allows us to bound the condition number of the local LSEs by

$$\kappa\left(\mathcal{M}\left(\overline{U_\alpha(x)} \,\langle E_\alpha \, A \, E_\alpha \rangle\, U_\alpha(x) \right) \right) \leq \kappa\left(\mathcal{M}\left(A \right) \right).$$

Our aim is therefore to *α-orthogonalise* x before solving the local LSE at α, i.e. to transform the vertex tensors $(x_\beta)_{\beta \in T_D}$ such that x becomes α-orthogonal but the represented tensor $\prod_{\beta \in T_D} x_\beta$ remains unchanged. We next reformulate and extend the strong D-orthogonalisation algorithm from [5, Alg. 3] to obtain an efficient scheme for iteratively orthogonalising an HTR network $x \in \mathrm{HTR}(T_D, r, n)$ with respect to all its vertices $\alpha \in T_D$ as required by the ALS Algorithm 1. This scheme will make use of the following vertex-wise operation.

DEFINITION 4.4 (Vertex Orthogonalisation). *Let $x \in \mathrm{HTR}(T_D, r, n)$ be an HTR network, $\epsilon \in E$ an edge and $\alpha, \beta \in T_D$ the two vertices such that $\epsilon \in E_\alpha \wedge \epsilon \in E_\beta$.* Orthogonalisation of x_α with respect to ϵ *is defined as the following operation:*

1: $(q, r) := \mathcal{Q}_\epsilon(x_\alpha)$
2: $x_\alpha := q$
3: $x_\beta := r x_\beta$

Note that vertex orthogonalisation does not modify the represented tensor since we have $\tilde{x}_\alpha \tilde{x}_\beta = q r x_\beta = x_\alpha x_\beta$ where x_α, x_β and $\tilde{x}_\alpha, \tilde{x}_\beta$ denote the vertex tensors before and after the orthogonalisation step, respectively. The trick to strongly D-orthogonalise an HTR network is to orthogonalise its vertices in the right order, which is leaves-to-root:

Algorithm 2 Strong D-Orthogonalisation

1: RECURSE(D)
2: **function** RECURSE(α)
3: **for** $\beta \in \mathrm{child}(\alpha)$ **do** RECURSE(β) **end for**
4: Orthogonalise x_α with respect to α *// Definition 4.4*
5: **end function**

Correctness of this algorithm is proven in [5]. Once an HTR network is strongly orthogonal with respect to any vertex $\alpha \in T_D$, we can move this orthogonal centre around using the following theorem.

THEOREM 4.5. *Let $x \in \mathrm{HTR}(T_D, r, n)$ be an HTR network and $\alpha \in T_D$, $\beta \in \mathrm{neighbour}(\alpha)$ two vertices such that x is strongly α-orthogonal. After orthogonalising x_α with respect to $\uparrow(\alpha \mid \beta)$, x is strongly β-orthogonal.*

Proof. Because x was initially strongly α-orthogonal and the subtree tensors $S_{\gamma|\alpha}(x) = S_{\gamma|\beta}(x)$ with $\gamma \in T_D \setminus \{\alpha, \beta\}$ are not affected by the vertex orthogonalisation, we only need to prove that $S_{\alpha|\beta}(x)$ is $\{\uparrow(\alpha \mid \beta)\}$-orthogonal. We can write $S_{\alpha|\beta}(x) = x_\alpha \prod_{\gamma \in \mathrm{child}(\alpha|\beta)} S_{\gamma|\alpha}(x)$, therefore we have

$$\begin{aligned}
&\overline{S_{\alpha|\beta}(x)}\ {}_{\langle\{\uparrow(\alpha|\beta)\}\rangle}\ S_{\alpha|\beta}(x) = \dots \\
&\qquad = \overline{x_\alpha}\ {}_{\langle\{\uparrow(\alpha|\beta)\}}\left(\prod_{\gamma\in\mathrm{child}(\alpha|\beta)} \overline{S_{\gamma|\alpha}(x)}\ {}_{\langle\{\uparrow(\gamma|\alpha)\}\rangle}\ S_{\gamma|\alpha}(x)\right){}_{\{\uparrow(\alpha|\beta)\}\rangle}\ x_\alpha \\
&\qquad = \overline{x_\alpha}\ {}_{\langle\{\uparrow(\alpha|\beta)\}\rangle}\ x_\alpha = \mathbb{I}_{\{\uparrow(\alpha|\beta)\}}. \qquad \square
\end{aligned}$$

In conclusion, stabilisation of the ALS Algorithm 1 requires us to make an initial call to the strong D-orthogonalisation Algorithm 2 and then let the orthogonality centre follow the vertex on which we solve the local LSE by using Theorem 4.5. 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 shown in [13] 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 order shown in Figure 4.2a, which we call *mole-like* order because it runs through the network like a mole runs through its burrow. Note that mole-like 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.

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 [16] and G_i (without the last vertex tensor A_i), H_i in [10].

DEFINITION 4.6 (Contracted Subtrees). *Let*

$$x \in \mathrm{HTR}(T_D, r^{(x)}, n), \quad A \in \mathrm{HTR}^2(T_D, r^{(A)}, n), \quad y \in \mathrm{HTR}(T_D, r^{(y)}, n)$$

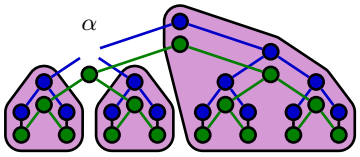
be HTR networks and $\alpha, \beta \in T_D$, $\alpha \neq \beta$ two vertices. We define the contracted subtrees

$$\begin{aligned}
(x|y)_{\beta|\alpha} &:= \prod_{\gamma\in\mathrm{descendant}(\beta|\alpha)} \overline{x_\gamma}\ y_\gamma \quad \in \mathbb{K}^{[r^{(x)}_{x(\uparrow(\beta|\alpha))}]\times[r^{(y)}_{y(\uparrow(\beta|\alpha))}]}, \\
(x|A|y)_{\beta|\alpha} &:= \prod_{\gamma\in\mathrm{descendant}(\beta|\alpha)} \overline{x_\gamma}\, A_\gamma\, y_\gamma \in \mathbb{K}^{[r^{(x)}_{x(\uparrow(\beta|\alpha))}]\times[r^{(A)}_{A(\uparrow(\beta|\alpha))}]\times[r^{(y)}_{y(\uparrow(\beta|\alpha))}]}.
\end{aligned}$$

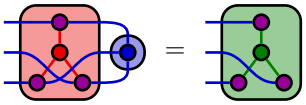
Furthermore, we set $(\star)_\beta := (\star)_{\beta|D}$ for $\star = x|y$ and $\star = x|A|y$.

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

$$\begin{aligned}
\overline{U_\alpha(x)}\ {}_{\langle E_\alpha}\ A\ {}_{E_\alpha\rangle}\ U_\alpha(x) &:= A_\alpha \prod_{\beta\in\mathrm{neighbour}(\alpha)} (x|A|x)_{\beta|\alpha}, \\
\overline{U_\alpha(x)}\, b &:= b_\alpha \prod_{\beta\in\mathrm{neighbour}(\alpha)} (x|b)_{\beta|\alpha},
\end{aligned} \tag{4.2}$$

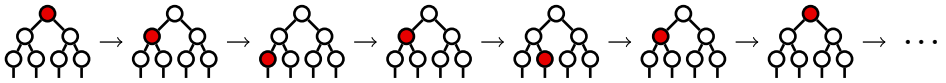


(a) Contracted subtrees.

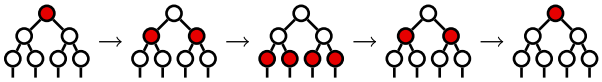


(b) Local LSE.

Figure 4.1: (a) Contracted subtrees $(x|b)_{\beta|\alpha}$ for $\beta \in \text{neighbour}(\alpha)$. (b) Local LSE in terms of the contracted subtrees $(x|A|x)_{\beta|\alpha}$, $(x|b)_{\beta|\alpha}$, $\beta \in \text{neighbour}(\alpha)$. Vertices and edges of A are shown in red, of x in blue and of b in green.



(a) Serial ALS Algorithm 3: mole-like.



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

Figure 4.2: Tree traversal orders of the serial and parallel HTR ALS algorithm. The red vertices denote the ones on which we currently solve local LSEs.

see also Figure 4.1, 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 formula

$$(x|A|y)_{\beta|\alpha} := \overline{x_\beta}\, A_\beta\, y_\beta \left(\prod_{\gamma \in \text{child}(\beta|\alpha)} (x|A|y)_{\gamma|\alpha} \right) \tag{4.3}$$

and the analogous expression for $(x|y)_{\beta|\alpha}$. 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 compute the contracted subtrees $(\star)_{\beta|D}$ for all $\beta \in \text{neighbour}(D)$ ($\star$ stands for both $x|A|x$ and $x|b$), which if evaluated according to (4.3) requires to compute all contracted subtrees $(\star)_{\beta|D}$ for $\beta \in T_D \setminus \{D\}$. If we then move from D to one of its neighbours $\alpha \in \text{neighbour}(D)$, we can reuse the $(\star)_{\beta|D}$ with $\beta \in \text{child}(\alpha)$ such that the only contracted subtrees to compute anew are the two tensors $(\star)_{D|\alpha}$. Because we already have the $(\star)_{\beta|\alpha} = (\star)_{\beta|D}$ for $\beta \in \text{sibling}(\alpha)$, this step involves only computations on D and has therefore a constant cost with respect to the network size. Proceeding further according to the mole-like tree traversal order from Figure 4.2a, we can continue in this manner for all vertices in the network.

4.2. Parallelisation. An important feature of the local LSE (4.1) 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

Algorithm 3 HTR ALS Algorithm

```
1: Strongly D-orthogonalise x                                  // Algorithm 2
2: Compute (x|A|x)_α and (x|b)_α for all α ∈ T_D \ {D}         // Equation (4.3)
3: repeat RECURSE(D) until convergence
4: function RECURSE(α)
5:     for β ∈ child(α) do
6:         Solve the local LSE                                 // Equations (4.1), (4.2)
7:         Orthogonalise x_α with respect to β                 // Definition 4.4
8:         Compute (x|A|x)_{α|β} and (x|b)_{α|β}               // Equation (4.3)
9:         RECURSE(β)
10:    end for
11:    Solve the local LSE                                     // Equations (4.1), (4.2)
12:    if α ≠ D then
13:        Orthogonalise x_α with respect to α                 // Definition 4.4
14:        Compute (x|A|x)_α and (x|b)_α                       // Equation (4.3)
15:    end if
16: end function
```

local LSEs concurrently. In Algorithm 3, this is expressed by the fact that we must run the loop over the children (line 5) sequentially, because the computations for the local LSE and the contracted subtrees on lines 6, 8 depend on the contracted subtrees computed on line 14 in the subordinate calls to RECURSE. The ALS algorithm as presented in §4.1 is therefore not parallelisable without algorithmic modifications eliminating the dependency between local LSE solves. Since already the 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 5 to 10 in Algorithm 3 as follows.

```
1: Solve the local LSE                                         // Equations (4.1), (4.2)
2: for β ∈ child(α) do
3:     Compute (x|A|x)_{α|β} and (x|b)_{α|β}                   // Equation (4.3)
4: end for
5: parallel for β ∈ child(α)
6:     RECURSE(β)
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 4.2b.

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_α but does not generate new information which would influence the local LSE at α, while the second loop does generate such information but does not read x_α. 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 4.7 (Simultaneous Vertex Orthogonalisation). *Let $x \in \mathrm{HTR}(T_D, r, I)$ be an HTR network and $\alpha \in T_D$ a vertex.* Orthogonalisation of x_α with respect to its children *is defined as the following operation:*

1: **for** $\beta \in \mathrm{child}(\alpha)$ **do**
2: $\quad (u_\beta, s_\beta, v_\beta) := \mathcal{S}_\beta(x_\alpha)$
3: $\quad x_\alpha := u_\beta\, s_\beta$
4: $\quad x_\beta := v_\beta\, x_\beta$
5: **end for**
6: **for** $\beta \in \mathrm{child}(\alpha)$ **do**
7: $\quad x_\alpha := x_\alpha\, s_\beta^{-1}$
8: $\quad x_\beta := s_\beta\, x_\beta$
9: **end for**

THEOREM 4.8. *Simultaneous vertex orthogonalisation leaves the represented tensor $x = \prod_{\beta \in T_D} x_\beta$ invariant.*

Proof. Lines 3, 4 do not modify x since $u_\beta s_\beta v_\beta\, x_\beta = x_\alpha x_\beta$, and the same holds for lines 7, 8 since $x_\alpha s_\beta^{-1} s_\beta x_\beta = x_\alpha x_\beta$. ∎

THEOREM 4.9. *Assume $S_{\beta|\alpha}(x)$ is $\{\uparrow(\beta|\alpha)\}$-orthogonal for all $\beta \in \mathrm{neighbour}(\alpha)$.*[2] *After orthogonalising x_α with respect to its children, $S_{\alpha|\beta}(x)$ is $\{\beta\}$-orthogonal for all $\beta \in \mathrm{child}(\alpha)$.*

Proof. We denote by x the original network, by x' the network that is obtained after the first loop (lines 1 to 5) has been executed and by x'' the final network. We define $\beta_1, \ldots, \beta_c \in \mathrm{child}(\alpha)$, $c := \#\,\mathrm{child}(\alpha)$, to be the children of α 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 $\beta = \beta_i$ of the first loop has been executed.

The proof splits into two parts.

1. We have

$$S_{\beta_i|\alpha}(x^{(j)}) = \begin{cases} S_{\beta_i|\alpha}(x) & \text{if } j < i \\ v_{\beta_i} S_{\beta_i|\alpha}(x) & \text{otherwise} \end{cases}$$

   for $i, j = 1, \ldots, c$. Since v_{β_i} and $S_{\beta_i|\alpha}(x)$ are $R(\beta_i)$- and $\{\beta_i\}$-orthogonal, respectively, $S_{\beta_i|\alpha}(x^{(j)})$ is $\{\beta_i\}$-orthogonal for any $i, j = 1, \ldots, c$.
2. Let $\gamma \in \mathrm{neighbour}(\alpha)$. From the proof of Theorem 4.8 it follows that $S_{\alpha|\gamma}(x)$ is not modified by lines 3, 4 unless γ is equal to the loop variable β. This proves

$$S_{\alpha|\beta_i}(x') = u_{\beta_i} s_{\beta_i} \prod_{\gamma \in \mathrm{child}(\alpha|\beta_i)} S_{\gamma|\beta_i}(x^{(i)}), \quad \forall i = 1, \ldots, c. \tag{4.4}$$

   Arguing similarly for the second loop, we obtain

$$S_{\alpha|\beta_i}(x'') = u_{\beta_i} \prod_{\gamma \in \mathrm{child}(\alpha|\beta_i)} S_{\gamma|\beta_i}(x^{(i)}), \qquad \forall i = 1, \ldots, c. \tag{4.5}$$

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

u_{β_i} is $\{\beta_i\}$-orthogonal and the $S_{\gamma|\beta_i}(x^{(i)})$ are $\{\uparrow(\alpha \mid \beta_i\}$-orthogonal by assumption for $\gamma = \text{parent}(\alpha)$ and by part 1 for $\gamma \in \text{sibling}(\beta_i)$. Therefore, $S_{\alpha|\beta_i}(x'')$ is $\{\beta_i\}$-orthogonal. □

Definition 4.7 silently assumed that s_β is invertible, i.e. that no singular value is exactly 0. In our code, we ensure this condition by transforming the singular values $(s_\beta)_{i_\beta}$, $i_\beta \in [r_\beta]$, with

$$(s_\beta)_{i_\beta} := \max\{(s_\beta)_{i_\beta}, \texttt{eps}\,(s_\beta)_0\} \tag{4.6}$$

where `eps` denotes the machine precision and $(s_\beta)_0$ 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 4.8 relies on the identities $u_\beta s_\beta v_\beta = x_\alpha$ and $s_\beta\, s_\beta^{-1} = \mathbb{I}_{\{\beta\}}$, both of which are satisfied up to machine precision when using the singular values from (4.6). Thus, the statement in Theorem 4.8 is valid up to machine precision as well.

In Theorem 4.9, the rounding in the singular values implies that (4.4) is satisfied up to a relative error of $\mathcal{O}(\texttt{eps})$. Multiplication with s_β^{-1} may then blow this error up such that the relative error in (4.5) is $\mathcal{O}(1)$, i.e. the $S_{\alpha|\beta}(x)$ may not be $\{\beta\}$-orthogonal at all. Luckily, we can limit the impact of rounding errors by interleaving orthogonalisation and subtree computation as follows.

Definition 4.10 (Combined Orthogonalisation and Contracted Subtree Computation). *Let $x \in \text{HTR}(T_D, r, I)$ be an HTR network and $\alpha \in T_D$ a vertex.* Combined orthogonalisation and contracted subtree computation at x_α *is defined as the following operation:*

for $\beta \in \text{child}(\alpha)$ **do**
  // Orthogonalise x_α with respect to β
  $(u_\beta, s_\beta, v_\beta) := \mathcal{S}_\beta(x_\alpha)$
  $x_\alpha := u_\beta$
  $x_\beta := s_\beta\, v_\beta\, x_\beta$
  // Compute subtrees
  Compute $(x|A|x)_{\alpha|\beta}$ and $(x|b)_{\alpha|\beta}$ *// Equation (4.3)*
  // Temporarily move the non-orthogonal factor to x_α
  $x_\alpha := x_\alpha\, s_\beta$
  $x_\beta := s_\beta^{-1}\, x_\beta$ *// (*)*
  // Update $(x|A|x)_\beta$ and $(x|b)_\beta$
  $(x|A|x)_\beta := \overline{v_\beta}\,(x|A|x)_\beta\, v_\beta^T$
  $(x|b)_\beta := \overline{v_\beta}\,(x|b)_\beta$
end for
for $\beta \in \text{child}(\alpha)$ **do**
  // Move the non-orthogonal factor to x_β again
  $x_\alpha := x_\alpha\, s_\beta^{-1}$
  $x_\beta := s_\beta\, x_\beta$ *// (*)*
  // Update $(x|A|x)_\beta$ and $(x|b)_\beta$
  $(x|A|x)_\beta := s_\beta\,(x|A|x)_\beta\, s_\beta$ *// (+)*
  $(x|b)_\beta := s_\beta\,(x|b)_\beta$ *// (+)*
end for

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 $(x|A|x)_\beta$ *and* $(x|b)_\beta$ *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 4.10 are equivalent to the ones in Definition 4.7, therefore Theorems 4.8 and 4.9 are also valid for combined orthogonalisation and subtree computation. The main idea of Definition 4.10 is to compute the contracted subtrees at a point (line 7) where $S_{\alpha|\beta}(x)$ is $\{\beta\}$-orthogonal up to errors of order $\mathcal{O}(\varepsilon)$. Since the local LSE are assembled based on these cached $(x|A|x)_{\alpha|\beta}$, $(x|b)_{\alpha|\beta}$ capturing accurately $\{\beta\}$-orthogonal $S_{\alpha|\beta}(x)$, it no longer matters that the final $S_{\alpha|\beta}(x)$ are not accurately $\{\beta\}$-orthogonal. We further remark that the updates to the contracted subtrees on lines 12, 13 and 20, 21 in Definition 4.10 are a consequence of the modifications done to x_β on lines 5, 10 and 18. In the serial Algorithm 3, such updates are not necessary because $(x|A|x)_\beta$, $(x|b)_\beta$ are updated anyway on line 14 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 [21] 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.

Algorithm 4 Parallel HTR ALS Algorithm

We assume an implicit cache of contracted subtrees. This cache is initialised on line 2 and only updated when we explicitly say so, namely on lines 7 and 15. 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: Strongly D-orthogonalise x                                  // Algorithm 2
2: Compute (x|A|x)_α and (x|b)_α for all α ∈ T_D \ {D}         // Equation (4.3)
3: repeat RECURSE(D) until convergence
4: function RECURSE(α)
5:     if child(α) ≠ {} then
6:         Solve the local LSE                                 // Equations (4.1), (4.2)
7:         Orthogonalise and compute contracted subtrees       // Definition 4.10
8:         parallel for β ∈ child(α)
9:             RECURSE(β)
10:        end parallel for
11:    end if
12:    Solve the local LSE                                     // Equations (4.1), (4.2)
13:    if α ≠ D then
14:        Orthogonalise x_α with respect to α                 // Definition 4.4
15:        Compute (x|A|x)_α and (x|b)_α                       // Equation (4.3)
16:    end if
17: end function
```

4.3. Computational Costs. We complement 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 [10, 16]. For this purpose, we assume T_D to be a standard dimension partition tree and $x \in \mathrm{HTR}(T_D, r^{(x)}, n)$,

$A \in \mathrm{HTR}^2(T_D, r^{(A)}, n)$, $b \in \mathrm{HTR}(T_D, r^{(b)}, n)$. We define

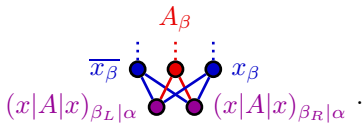
$$d := \#D, \quad n := \max_{k \in D} n_k,$$
$$r := \max_{\alpha \in T_D \setminus \{D\}} r_\alpha^{(x)}, \quad R := \max_{\alpha \in T_D \setminus \{D\}} r_\alpha^{(A)}, \quad R_b := \max_{\alpha \in T_D \setminus \{D\}} r_\alpha^{(b)}.$$

The ranks of x are usually larger than the ranks of A or b, thus we will assume $R^a r^b \leq R^{a'} r^{b'}$ if $a+b = a'+b'$ and $b \leq b'$, 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 $r \leq n$. This allows us to order such terms according to $n^a r^b \leq n^{a'} r^{b'}$ if $a + b = a' + b' \wedge a \leq a'$.

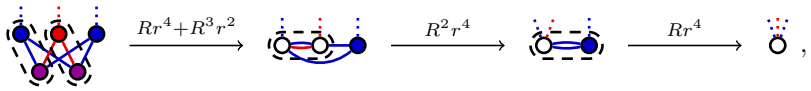
In the following, we count for each part of the ALS algorithm — orthogonalisation, contracted subtree computations and local LSE solves — the number of floating-point operations (FLOP) arising during a single iteration, i.e. a single call to RECURSE(D). We state already here that the costs at the root 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 [5, Lemma 4.8] for a more detailed result regarding the cost of the strong D-orthogonalisation Algorithm 2.

The recursive computation of the **contracted subtrees** $(x|A|x)_{\beta|\alpha}$ with $\beta \in \mathrm{interior}(T_D)$, $\alpha \in T_D$ and $\mathrm{child}(\beta \mid \alpha) = \{\beta_L, \beta_R\}$ requires evaluating

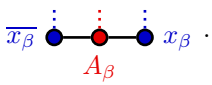


We propose to contract this network according to the sequence[3]



which costs $\mathcal{O}(R^2 r^4)$ FLOP.

At the leaf vertices $\beta \in \mathrm{leaf}(T_D)$, the network to contract is given by

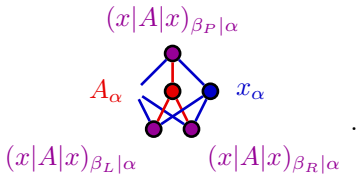


and can be evaluated in $\mathcal{O}(n^2 Rr)$ FLOP. In total, computing the contracted subtrees $(x|A|x)_{\beta|\alpha}$ thus costs $\mathcal{O}\left(dR^2 r^4 + dn^2 Rr\right)$. The costs for computing $(x|b)_{\beta|\alpha}$ are obtained similarly. We only state the final result, which is $\mathcal{O}\left(dR_b r^3 + dnR_b r\right)$.

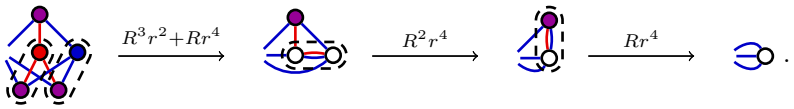
The **local LSEs** are best solved using an iterative method like conjugate gradient or GMRES because on the one hand, the old, to-be-replaced vertex tensor often provides a good initial guess for the new, replacing one, and on the other hand, the special structure of the local matrix allows for an efficient matrix-vector product. To

[3] The Matlab script from [20] was of great use to determine such contraction sequences.

make the second point more concrete, we consider the matrix-vector product at an interior vertex $\alpha \in \text{interior}(T_D)$ with $\text{neighbour}(\alpha) = \{\beta_P, \beta_L, \beta_R\}$ which amounts to contracting



see also Figure 4.1b. 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 iterand and therefore fail to produce a reasonably accurate solution if the initial iterand ranks are chosen too small, but become unnecessarily costly if these ranks are overestimated. The ALS(SD) algorithm from [3] 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 $r_z = 3$ applied to the system $\mathbb{I}_D\, z = b - Ax$, see [3] for details. For the truncation step, we use the algorithm from [5] choosing the ranks adaptively such that the original and truncated tensors $x \in \text{HTR}(T_D, r, n)$, $\tilde{x} \in \text{HTR}(T_D, \tilde{r}, n)$ 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.

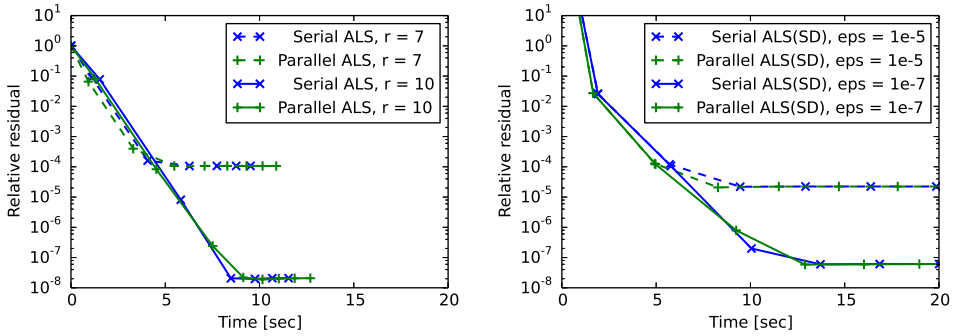


Figure 4.3: Convergence of the serial and parallel ALS and ALS(SD) algorithms applied to the 128-dimensional Poisson equation. r denotes the rank of the iterand, `eps` the relative truncation tolerance.

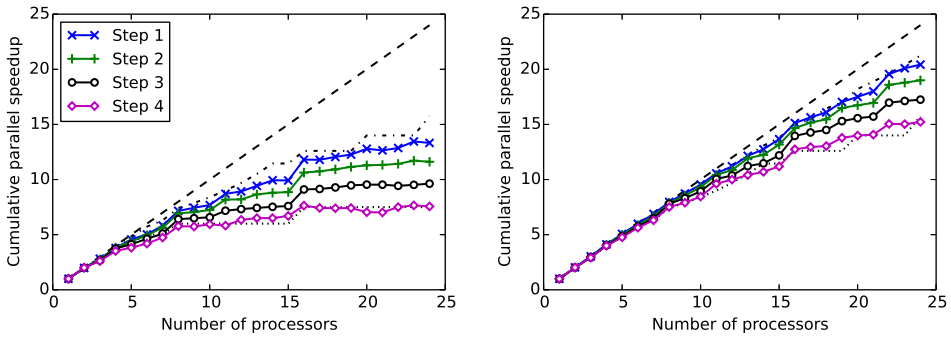
4.5. Numerical Experiments. We investigate the numerical properties of the above algorithms by means of the d-dimensional Poisson equation $-\Delta u = 1$ on $[0,1]^d$ with homogeneous Dirichlet boundary conditions, discretised according to the standard finite difference scheme on a uniform mesh with $n = 2^6$ interior grid points in each dimension. This linear system of equations is then *quantised* [22, 14, 17, 11] into 6 virtual modes of length 2 each, and the resulting $6d$ modes are organised into a dimension partition 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 4.3, this is not the case.

Next, we investigate the strong scaling of the parallel ALS(SD) algorithm. From Figure 4.4 we conclude:

- Theorem 3.8 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 dimension partition tree is not perfectly balanced.
- The scalability decreases with increasing iteration count of the ALS(SD) algorithm. This is because in the first iteration, the ranks are almost uniform (between 2 and 4) while in later iterations the ranks become increasing towards the root.

Technical Details. All benchmarks were run on two twelve-core AMD Opteron 6174 processors (2.2 GHz). 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 iterand. The crosses in Figure 4.3 refer to the value



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

Figure 4.4: 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 $\varepsilon = 10^{-7}$. The dashed lines denote perfect speedup, the dash-dotted lines the optimal speedup from Theorem 3.8 for $d = 4d_p$ and the dotted lines the optimal speedup for $d = d_p$.

of this quantity after an ALS/ALS(SD) iteration has been completed. "Cumulative parallel speedup" is computed as $\frac{T(1)}{T(p)}$ where $T(p)$ is the wall clock time up to the indicated ALS(SD) iteration on p processors. The vertices were distributed using a heuristic algorithm mapping the vertices such that as many neighbours as possible are assigned to the same process under the constraint that the optimal speedup from Theorem 3.8 must still be achievable. The runtimes per vertex were assumed uniform such that $\mathrm{level}_t(\alpha) \propto \mathrm{level}(\alpha)$ and $\mathrm{colevel}_t(\alpha) \propto \mathrm{colevel}(\alpha)$.

5. Conclusion. We pioneered the parallel implementation of tensor-network structured algorithms. In particular, we motivated why any truly parallel implementation must be based on the HTR rather than the simpler and more well-known TT format, presented a generic parallelisation scheme and derived explicit bounds on the parallel potential of HTR algorithms. Based on ideas from [21], we then proposed a modified, parallelisable ALS algorithm and highlighted the implementation details required to make the modified algorithm efficient as well as stable with respect to finite-precision arithmetic. Our numerical experiments demonstrate that it is possible to get an order of magnitude speedup already on moderately sized networks, yet they also hint at possible obstacles preventing effective parallelisation, namely low dimensionality and non-uniform ranks.

Acknowledgements. I would like to thank Vladimir Kazeev for his advice on tensor network formats and tensor-network-structured linear solvers, and Robert Gantner for his support regarding the implementation.

REFERENCES

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

[2] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle, *A multilinear singular value decomposition*, SIAM J. Matrix Anal. Appl., 21 (2000), pp. 1253–1278 (electronic).

[3] S. Dolgov and D. Savostyanov, *Alternating minimal energy methods for linear systems in higher dimensions*, SIAM Journal on Scientific Computing, 36 (2014), pp. A2248–A2271.
[4] Gene H. Golub and Charles F. Van Loan, *Matrix Computations (3rd Ed.)*, Johns Hopkins University Press, Baltimore, MD, USA, 1996.
[5] Lars Grasedyck, *Hierarchical singular value decomposition of tensors*, SIAM J. Matrix Anal. Appl., 31 (2009/10), pp. 2029–2054.
[6] L. Grasedyck, *Polynomial approximation in hierarchical Tucker format by vector–tensorization*, Tech. Report 308, RWTH Aachen, Institut für Geometrie und Praktische Mathematik, April 2010.
[7] Lars Grasedyck, Daniel Kressner, and Christine Tobler, *A literature survey of low-rank tensor approximation techniques*, GAMM-Mitt., 36 (2013), pp. 53–78.
[8] Wolfgang Hackbusch, *Tensor spaces and numerical tensor calculus*, vol. 42 of Springer Series in Computational Mathematics, Springer, Heidelberg, 2012.
[9] W. Hackbusch and S. Kühn, *A new scheme for the tensor representation*, Journal of Fourier Analysis and Applications, 15 (2009), pp. 706–722. 10.1007/s00041-009-9094-9.
[10] S. Holtz, T. Rohwedder, and R. Schneider, *The alternating linear scheme for tensor optimization in the tensor train format*, SIAM Journal on Scientific Computing, 34 (2012), pp. A683–A713.
[11] Boris N. Khoromskij, *$O(d \log N)$-quantics approximation of N-d tensors in high-dimensional numerical modeling*, Constr. Approx., 34 (2011), pp. 257–280.
[12] Boris N. Khoromskij, *Tensors-structured numerical methods in scientific computing: Survey on recent advances*, Chemometrics and Intelligent Laboratory Systems, 110 (2012), pp. 1 – 19.
[13] Daniel Kressner and Christine Tobler, *Preconditioned low-rank methods for high-dimensional elliptic PDE eigenvalue problems*, Comput. Methods Appl. Math., 11 (2011), pp. 363–381.
[14] I. Oseledets, *Approximation of matrices with logarithmic number of parameters*, Doklady Mathematics, 80 (2009), pp. 653–654.
[15] I.V. Oseledets, *Constructive representation of functions in low-rank tensor formats*, Constructive Approximation, 37 (2013), pp. 1–18.
[16] I. Oseledets and S. Dolgov, *Solution of linear systems and matrix inversion in the TT-format*, SIAM Journal on Scientific Computing, 34 (2012), pp. A2718–A2739.
[17] I. V. Oseledets, *Approximation of $2^d \times 2^d$ matrices using tensor decomposition*, SIAM Journal on Matrix Analysis and Applications, 31 (2010), pp. 2130–2145.
[18] I. V. Oseledets, *Tensor-train decomposition*, SIAM Journal on Scientific Computing, 33 (2011), p. 2295–2317.
[19] 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 (2009), pp. 3744–3759.
[20] Robert N. C. Pfeifer, Jutho Haegeman, and Frank Verstraete, *Faster identification of optimal contraction sequences for tensor networks*, Phys. Rev. E, 90 (2014), p. 033315.
[21] E. M. Stoudenmire and Steven R. White, *Real-space parallel density matrix renormalization group*, Phys. Rev. B, 87 (2013), p. 155137.
[22] E. E. Tyrtyshnikov, *Tensor approximations of matrices generated by asymptotically smooth functions*, Sbornik: Mathematics, 194 (2003), pp. 941–954.

Recent Research Reports

Nr.	Authors/Title

2015-15 A. Paganini and S. Sargheini and R. Hiptmair and C. Hafner
Shape Optimization of microlenses

2015-16 V. Kazeev and Ch. Schwab
Approximation of Singularities by Quantized-Tensor FEM

2015-17 P. Grohs and Z. Kereta
Continuous Parabolic Molecules

2015-18 R. Hiptmair
Maxwell's Equations: Continuous and Discrete

2015-19 X. Claeys and R. Hiptmair and E. Spindler
Second-Kind Boundary Integral Equations for Scattering at Composite Partly Impenetrable Objects

2015-20 R. Hiptmair and A. Moiola and I. Perugia
A Survey of Trefftz Methods for the Helmholtz Equation

2015-21 P. Chen and Ch. Schwab
Sparse-Grid, Reduced-Basis Bayesian Inversion: Nonaffine-Parametric Nonlinear Equations

2015-22 F. Kuo and R. Scheichl and Ch. Schwab and I. Sloan and E. Ullmann
Multilevel Quasi-Monte Carlo Methods for Lognormal Diffusion Problems

2015-23 C. Jerez-Hanckes and Ch. Schwab
Electromagnetic Wave Scattering by Random Surfaces:

Uncertainty Quantification via Sparse Tensor Boundary Elements