Efficient, Optimal MPI Datatype Reconstruction for Vector and Index Types*

Martin Kalany
kalany@par.tuwien.ac.at

Jesper Larsson Träff
traff@par.tuwien.ac.at

Vienna University of Technology (TU Wien)
Faculty of Informatics, Institute of Information Systems
Research Group Parallel Computing
Favoritenstrasse 16/184-5, 1040 Vienna, Austria

ABSTRACT

Type reconstruction is the process of finding an efficient representation in terms of space and processing time of a data layout as an MPI derived datatype. Practically efficient type reconstruction and normalization is important for high-quality MPI implementations that strive to provide good performance for communication operations involving non-contiguous data. Although it has recently been shown that the general problem of computing optimal tree representations of derived datatypes allowing any of the MPI derived datatype constructors can be solved in polynomial time, the algorithm for this may unfortunately be impractical for datatypes with large counts. By restricting the allowed constructors to vector and index-block type constructors, but excluding the most general `MPI_Type_create_struct` constructor, the problem can be solved much more efficiently. More precisely, we give a new $O(n \log n / \log \log n)$ time algorithm for finding cost-optimal representations of MPI type maps of length n using only vector and index-block constructors for a simple but flexible, additive cost model. This improves significantly over a previous $O(n\sqrt{n})$ time algorithm for the same problem, and the algorithm is simple enough to be considered for practical MPI libraries.

1. INTRODUCTION

The derived datatype functionality of MPI [6] makes it possible to describe the structure (layout in memory) of data to be communicated in a way that is orthogonal to the communication operations used. This is a unique and powerful mechanism of considerable generality, and for its effective use in applications it is important that the MPI library implementation can handle derived datatypes in an efficient manner. One aspect of this is to be able to find space and processing time efficient, internal representations for derived datatypes. We have addressed various aspects of this problem in a number of papers [1, 3, 11], with the aim of settling the complexity of the problem of computing cost-optimal representations in a cost model that reasonably reflects space and related aspects of the processing costs.

MPI defines a small set of constructors of increasing generality for constructing derived datatypes that can describe any given application data layout. A data layout, also called a *type map*, is a sequence of basic datatypes (`MPI_CHAR`, `MPI_INT`, `MPI_DOUBLE`, ...) and their relative displacements. Such layouts can be more concisely represented by grouping subsequences of types that repeat in certain patterns. The MPI derived datatype mechanism provides various constructors to describe different kinds of repeated patterns. A derived datatype is a tree or DAG structure with inner nodes describing repetitions of leaves corresponding to the basic datatypes of the type map. If a cost is associated with inner constructor nodes that reflects space consumption and processing time, it is a natural problem to ask for the least cost tree or DAG description of a given type map. We term this the *type reconstruction problem* [3]. The related, important problem of changing a given type tree or DAG into a cost-optimal one is called *type normalization* [3].

As recently shown [1], the type reconstruction problem into trees for type maps of length n can be solved in $O(n^4)$ operations. Although polynomial, this is too time consuming for applications where n is large. For instance, n could be on the order of p, the number of MPI processes in the application, as could be the case when distributed arrays are used. Commonly used type normalization in MPI libraries seems limited to heuristic approaches with no cost-optimality guarantees [4, 7, 9, 10].

In this paper we study the situation when the set of allowed constructors is restricted. The problematic constructor is `MPI_Type_create_struct`, which allows subtrees of different derived datatypes. Indeed, this is the only constructor which turns derived datatype descriptions into trees or DAGs. All other MPI type constructors take only a single subtype, and thus the corresponding descriptions are simple paths. In a previous paper [11], we showed that the problem can be solved much more efficiently if only the constructors for contiguous substructures (`MPI_Type_contiguous`), strided vectors (`MPI_Type_vector`) and indexed layouts (`MPI_Type_create_indexed_block`) are allowed, namely in $O(n\sqrt{n})$

*This work was co-funded by the European Commission through the EPiGRAM project (grant agreement no. 610598).

EuroMPI '15, September 21-23, 2015, Bordeaux , France
© 2015 ACM. ISBN 978-1-4503-3795-3/15/09...$15.00
DOI: http://dx.doi.org/10.1145/2802658.2802671

operations. In this paper we improve considerably on this result, and show that the restricted problem can be solved almost linearly in $O(n \log n / \log \log n)$ operations. Our new, different algorithm is also more straightforward, and could possibly be implemented in MPI libraries.

We extend our algorithm to also cover the `MPI_Type_-indexed` constructor, which allows blocks of different numbers of substructures at the given displacements. Unfortunately, this incurs an increase in complexity.

Layouts that can be described optimally with the vector and indexed constructors are common in applications, e.g., sub-matrices or -tensors, or triangular or banded matrices, etc. It is worth pointing out that an optimal derived datatype using only the restricted set of constructors can possibly be further normalized when also `MPI_Type_create_struct` is allowed. A simple example of this is shown in Figure 2, where the path constructors allow only a trivial, linear sized description in contrast to the constant sized description possible when the struct constructor is also permitted. Thus, although normalization of derived datatypes that use only the restricted set of constructors can lead to worthwhile improvements, it is well possible that better representations can be found by introducing structure constructors in the internal representation.

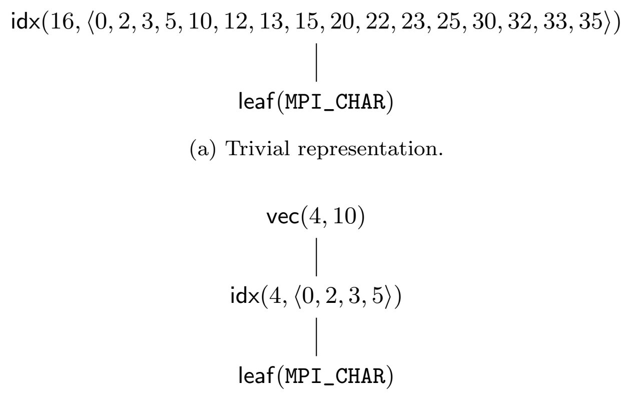
In the remainder of the paper, we concentrate on explaining and proving optimality of type reconstruction for the restricted set of derived datatype constructors. We also briefly discuss additional derived datatype constructors that can be handled by our algorithm. Such constructors would make it possible to describe larger classes of data layouts in a concise way without having to revert to the full generality of `MPI_Type_create_struct`, and still permitting low-complexity reconstruction. Such constructors are therefore of interest to future MPI developments.

2. THE PROBLEM

Type reconstruction is the problem of constructing cost-optimal type trees from given displacement sequences representing MPI type maps. Before we can address the problem, we need to formally define our representation of type paths and the cost model used. We also explain how the MPI constructors map efficiently to our internal representation.

As stated, an MPI type map [6, Chapter 4] is a(n ordered) sequence of pairs of basic datatypes corresponding to data types of C or FORTRAN together with their relative displacements. With a given start address, a type map is therefore an explicit, linear sized description of a data layout that might appear in an application.

In this work, we restrict consideration to the `MPI_Type_-contiguous`, `MPI_Type_vector`, and `MPI_Type_create_indexed_-block` and `MPI_Type_indexed` constructors, which can describe only homogeneous data layouts that consist of exactly one distinct basic datatype, called the *base type*. Therefore, we model our problem as that of finding a concise description of a *displacement sequence* using a given set of *restricted constructors* (to distinguish it from the problem where the full set of MPI-like constructors is allowed). An n-element displacement sequence D is a sequence $D = \langle d_0, d_1, \ldots, d_{n-1} \rangle$ of arbitrary integer displacements $D[i] = d_i$ for $0 \leq i < n$. We can describe any displacement sequence possibly more concisely as a *basic type path* built from the restricted set of constructors introduced in Definition 1. A *basic type path* T is either a leaf node, or a vector or an index node over a


idx(16, ⟨0, 2, 3, 5, 10, 12, 13, 15, 20, 22, 23, 25, 30, 32, 33, 35⟩)
leaf(MPI_CHAR)
vec(4, 10)
idx(4, ⟨0, 2, 3, 5⟩)
leaf(MPI_CHAR)

(a) Trivial representation.

(b) More concise type path representation.

Figure 1: Two type path representations for the displacement sequence $D = \langle 0, 2, 3, 5, 10, 12, 13, 15, 20, 22, 23, 25, 30, 32, 33, 35 \rangle$ for elements of type `MPI_CHAR`.

sequence T' described by a type path.

Definition 1 (Basic type path constructors). *A basic type path can be constructed from the following three constructors.*

1. *A* leaf, $\mathsf{leaf}(B)$, *represents the base type B with displacement 0.*
2. *A* vector $\mathsf{vec}(c, d, C)$ *with* count c *and* stride d *describes the catenation of c sequences C at relative displacements $0, d, 2d, \ldots, (c-1)d$.*
3. *An* index, $\mathsf{idx}(c, \langle i_0, i_1, \ldots, i_{c-1} \rangle, C)$, *with* count c *and* indices $\langle i_0, i_1, \ldots, i_{c-1} \rangle$ *describes the catenation of c sequences C at relative displacements $i_0, i_1, \ldots, i_{c-1}$.*

Note that any n element displacement sequence D can be represented as $\mathsf{idx}(n, D, \mathsf{leaf}(B))$, where B is the base type of the type map. This so-called trivial representation lists all displacements explicitly. An example is given in Figure 1a. A more concise representation is possible by exploiting regularity in the structure of the displacement sequence, as shown in Figure 1b.

Listing 1: A possible **Typenode** structure for representing nodes in type paths.

```
struct {
    enum nodetype = {leaf, vec, idx, idxbuc};
    int c ;                                  /* count */
    int d ;                                 /* stride */
    int D[] ;          /* displacement of subtypes */
    int b[] ;                         /* bucket sizes */
    Typenode subtype ;      /* sub- or base type */
} Typenode;
```

A type path represents a displacement sequence that can be obtained by an ordered traversal of the type path's nodes. This process is called *flattening* and Algorithm 2 gives a straightforward implementation for the set of constructors considered in this paper (we make no claim that this approach is a good way of implementing flattening). A type

Algorithm 2: Flattening procedure defining the displacement sequence represented by a given type path T (nodes are represented by the C-style structure shown in Listing 1). The procedure is called with a base offset, which will normally be 0. The procedure can trivially be extended to also cover further constructors.

```
Function Flatten($T$, $base$)
    switch $T.nodetype$ do
        case leaf                                  /* leaf */
            print $base$;
        case vec                                   /* vector */
            for $i \leftarrow 0$; $i < T.c$; $i$++ do
                Flatten($T.subtype$, $base + i \cdot T.d$);
        case idx                                   /* index */
            for $i \leftarrow 0$; $i < T.c$; $i$++ do
                Flatten( $T.subtype$, $base + T.D[i]$);
        case idxbuc                                /* index bucket */
            for $i \leftarrow 0$; $i < T.c$; $i$++ do
                for $j \leftarrow 0$; $j < T.b[i]$; $j$++ do
                    Flatten($T.subtype$,
                        $base + T.D[i] + j \cdot T.d$);
```

path T is said to *represent a displacement sequence* D if $D = \texttt{Flatten}(T, 0)$. It is clear that different type path representations for a given displacement sequence exist.

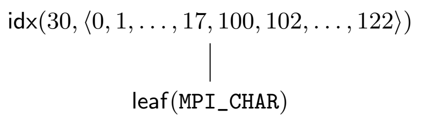
We will use these type constructors as our internal representation of the MPI derived datatypes. The representation is obvious. Let T be an MPI derived datatype with subtype (oldtype) S, and let e be the extent of S. Assuming that we can represent S using our restricted constructors by S, then we can represent T by T as follows.

- If T is `MPI_Type_contiguous` (c, S) then T is $\mathsf{vec}(c, e, S)$.
- If T is `MPI_Type_vector` (c, b, s,S) then T is $\mathsf{vec}(c, se, \mathsf{vec}(b, e, S))$.
- If T is `MPI_Type_create_indexed_block` (c, $\langle i_0, i_1, \ldots, i_{c-1}\rangle$, S) then T is $\mathsf{idx}(c, \langle i_0e, i_1e, \ldots, i_{c-1}e\rangle, S)$.

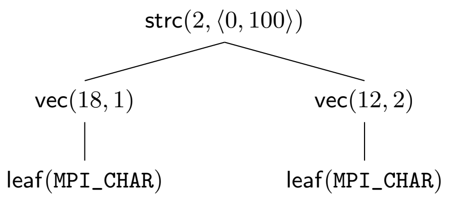
An MPI basic datatype T of c bytes is represented by a leaf node $\mathsf{leaf}(\mathsf{T})$. For the examples given in this paper, we will usually assume the base type to be `MPI_CHAR` with a size of 1 byte. All examples can easily be adapted to different base types by multiplying displacements and strides with the size (in bytes) of the new base type. Note that the MPI extent of a derived datatype is not stored as part of our internal representation.

We associate costs with the constructor nodes such that the *cost* of a type node is proportional to the number of words that must be stored in order to process the node. This includes the node type ($\mathsf{leaf}, \mathsf{vec}, \mathsf{idx}$), count, displacement or pointer to index or type array, pointer to child node, and a per element lookup cost for the lists of indices:

$$\begin{aligned} \text{cost}(\mathsf{leaf}(B)) &= K_{\mathsf{leaf}} \\ \text{cost}(\mathsf{vec}(c, d, C)) &= K_{\mathsf{vec}} \\ \text{cost}(\mathsf{idx}(c, \langle\ldots\rangle, C)) &= K_{\mathsf{idx}} + cK_{\text{lookup}} \end{aligned}$$



(a) Optimal type path representation.



(b) Optimal and more concise type tree representation.

Figure 2: Optimal (a) basic type path and (b) type tree representations for a displacement sequence $D = \langle 0, 1, \ldots, 17, 100, 102, \ldots, 122\rangle$.

The constants can be adjusted to reflect other overheads related to representing and processing a node. The per node cost accounts for the space required to store the constructor node. It is also a lower bound on the effort required to process a type node since all stored fields have to be accessed, including the list of indices in index nodes. The *cost* of a type path T is defined as the *additive cost* of its nodes T_i: $\text{cost}(T) = \sum_i \text{cost}(T_i)$. For the examples in this paper we have taken $K_{\mathsf{leaf}} = K_{\mathsf{vec}} = K_{\mathsf{idx}} = 6$ and $K_{\text{lookup}} = 1$, which matches with the C-style structure (Listing 1) used to represent the type constructors considered in this paper. Using these cost values, the trivial representation in Figure 1a has a cost of $K_{\mathsf{idx}} + 16K_{\text{lookup}} + K_{\mathsf{leaf}} = 28$. The alternate representation in Figure 1b has a total cost $K_{\mathsf{vec}} + K_{\mathsf{idx}} + 4K_{\text{lookup}} + K_{\mathsf{leaf}} = 22$ and is thus more concise.

We can now formally define the type reconstruction problem with the basic set of constructors from Definition 1.

Basic Type Path Reconstruction Problem
Instance: A displacement sequence D of length n.
Task: Find a least-cost (that is, optimal) type path T representing D using the leaf, idx and vec constructors. A type path T is optimal if $\text{cost}(T) \leq \text{cost}(T')$ for any type path T' representing D.

The constructor that we are not considering here, and which changes the difficulty of the type reconstruction problem considerably is `MPI_Type_create_struct`. In [1] we represent this as a *struct* node, $\mathsf{strc}(c, \langle i_0, i_1, \ldots, i_{c-1}\rangle, \langle C_0, C_1, \ldots, C_{c-1}\rangle)$, with *count* c and *indices* $\langle i_0, i_1, \ldots, i_{c-1}\rangle$ that describe the catenation of c sequences $C_0, C_1, \ldots, C_{c-1}$ at relative displacements $i_0, i_1, \ldots, i_{c-1}$. The cost associated with this constructor is

$$\text{cost}(\mathsf{strc}(c, \langle\ldots\rangle, \langle\ldots\rangle, \langle\ldots\rangle)) = K_{\mathsf{strc}} + 2cK_{\text{lookup}}$$

and again we take $K_{\mathsf{strc}} = 6$. With this constructor, derived datatypes are now no longer limited to be simple paths only, but can be arbitrary trees or DAGs. As shown in Figure 2, *type trees* make it possible to describe displacement sequences arbitrarily more concisely than with the restricted constructors only. If only the restricted set of constructors is

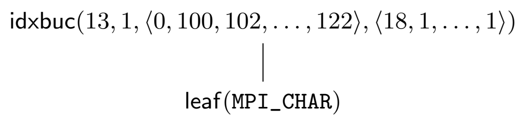

idxbuc(13, 1, ⟨0, 100, 102, . . . , 122⟩, ⟨18, 1, . . . , 1⟩)
leaf(MPI_CHAR)

Figure 3: An optimal *extended type path* representation of the displacement sequence $D = \langle 0, 1, \ldots, 17, 100, 102, \ldots, 122\rangle$ with a cost of 38.

permitted, D must be represented as a flat index block type with a cost of 42, whereas the strc constructor allows for a more concise representation with cost 34. The cost savings can be made arbitrarily large by increasing the length of the two subsequences.

MPI has another constructor for indexed layouts where the indexed blocks may have varying element counts but are otherwise all of the same subtype. The `MPI_Type_indexed` constructor therefore also gives rise to only type paths, and can sometimes be more less costly than the `MPI_Type_create_indexed_block` constructor. We will also be able to reconstruct type paths optimally when this constructor is permitted, and term the corresponding problem the Extended Type Path Reconstruction Problem. Formally, an *index bucket* node, $\mathsf{idxbuc}(c, d, \langle i_0, i_1, \ldots i_{c-1}\rangle, \langle b_0, b_1, \ldots b_{c-1}\rangle, C)$, with *count* c and *substride* d describes the catenation of c sequences at relative indices $i_0, i_1, \ldots i_{c-1}$. The i-th sequence is the catenation of b_i sequences C at relative displacements $0, d, 2d, \ldots (b_i - 1)d$. The idxbuc constructor node models the `MPI_Type_indexed` constructor as follows.

- If T is `MPI_Type_indexed` $(c, \langle i_0, i_1, \ldots, i_{c-1}\rangle, \langle b_0, b_1, \ldots b_{c-1}\rangle, \mathsf{S})$ then T is $\mathsf{idxbuc}(c, e, \langle i_0 e, i_1 e, \ldots, i_{c-1} e\rangle, \langle b_0, b_1, \ldots b_{c-1}\rangle, S)$, where e is the extent of S.

The cost associated with the idxbuc constructor is

$$\text{cost}(\mathsf{idxbuc}(c, d, \langle\ldots\rangle, \langle\ldots\rangle, C)) \quad = \quad K_{\mathsf{idxbuc}} + 2cK_{\text{lookup}}$$

where we can take $K_{\mathsf{idxbuc}} = 6$ as for the other constructors. The type path reconstruction problem extends naturally as follows.

Extended Type Path Reconstruction Problem
Instance: A displacement sequence D of length n.
Task: Find a least-cost (optimal) type path T representing D using the leaf, vec, idx and idxbuc constructors.

Figure 3 shows an optimal extended type path representation for the same displacement sequence used in Figure 2. Note that its cost is slightly smaller than the cost of the optimal basic type path in Figure 2a, but also slightly larger than the cost of the optimal type tree representation in Figure 2b.

3. TYPE PATH RECONSTRUCTION

We now give a detailed exposition of our algorithms for the type reconstruction problems with type path constructors. We first define precisely some important properties of displacement sequences and make some general, but crucial observations that will apply for both the Basic Type Path Reconstruction Problem and the Extended Type Path Reconstruction Problem. An $O(n \log n / \log \log n)$ algorithm solving the Basic Type Path Reconstruction Problem is then presented in detail in Section 3.2, and Section 3.3 extends the algorithm to solve the Extended Type Path Reconstruction Problem, though the latter increases the runtime to $O(n^2 \log^2 n)$.

A *segment* of an n-element displacement sequence D from index i to index j is denoted by $D[i, j] = \langle d_i, d_{i+1}, \ldots, d_j\rangle$, $0 \le i \le j < n$. A segment $D[0, q-1]$, $q \le n$ is also called a *prefix of length* q. Analogously, a *postfix starting at index* q denotes the segment $D[q, n-1]$. For convenience, we will refer to a prefix of length q as P_q. Note that a displacement sequence is always a prefix of itself.

Definition 2 (Repeated Prefix, Strided Prefix). *A* repeated prefix *(or* repetition*) of* length q *in an n-element displacement sequence D is a prefix $P_q = D[0, q-1]$ s.t. q is a divisor of n and for all i,j, $1 \le i < n/q$, $1 \le j < q$ we have that $D[j] - D[0] = D[iq + j] - D[iq]$. A* strided prefix *additionally fulfills $D[(i+1)q] - D[iq] = D[q] - D[0]$ for all i, $0 \le i < n/q - 1$, where $d = D[q] - D[0]$ is the* stride *of the repeated prefix P_q.*

The definition of a repeated prefix can equivalently be stated as $q \mid n$ and $D[j] - D[j-1] = D[iq+j] - D[iq+j-1]$ for all i, j, $1 \le i < n/q$, $1 \le j < q$. Taking $S[i] = D[i] - D[i-1]$ for all i, $1 \le i < n$, this is equivalent to

$$\forall i, 1 \le i < n : \ i \bmod q = 0 \vee S[i \bmod q] = S[i] \quad .$$

By definition, for any displacement sequence D the prefix of length 1 is a repeated prefix which corresponds to the trivial representation $D = \mathsf{idx}(n, D, \mathsf{leaf}(B))$, implying that a type path representation exists for any displacement sequence. Repeated prefixes and strided prefixes of length greater than 1 directly lead to possibly more efficient representations of a displacement sequence with the idx, idxbuc and vec constructors, respectively. If a repeated prefix P_q of length q exists in a displacement sequence D of length n, D can be represented as $\mathsf{idx}(c, \langle D[0], D[q], \ldots, D[(c-1)q]\rangle, P_q)$, where $c = n/q$. If the prefix is additionally strided, D can be represented as $\mathsf{vec}(n/q, D[q] - D[0], P_q)$. Furthermore, a repeated prefix P_q always allows for a representation via the idxbuc constructor, although such a representation will only be more cost efficient than the analogous representation via an idx constructor if D contains "buckets" where the prefix P_q is a strided repetition, repeated b_i times with stride d in the i-th bucket. Additionally, D can also be represented as $D = \mathsf{idx}(1, \langle 0\rangle, D)$ or $D = \mathsf{vec}(1, 0, D)$. However, due to the additive cost model, such representations are in general more expensive. Observe that by the definition of the vec, idx and idxbuc constructors, the sequences C must be repeated prefixes, only the exact nature of the repetition (strided, irregular, bucket layout) differs among those constructors. Note that the leaf constructor can only appear as a leaf node and is also the only constructor that may form the leaf node of a type path representation. Obviously, any type path contains exactly one such leaf node.

We formalize the above observations to give a precise characterization of type paths. A type path consists of type nodes that directly correspond to the type constructors and a subtype (or subtree) is a type path representing the repeated sequence C. An optimal type path T representing a displacement sequence D of length n is either

- $T = \mathsf{leaf}(B)$, a single leaf node with base type B if n equals 1; or
- $T = \mathsf{vec}(c, d, S)$, where the prefix $P_q = \langle D[0], \ldots, D[q-1]\rangle$, with $q = n/c$, is a strided prefix in D with stride d, and S is an optimal type path representation for the prefix P_q; or
- $T = \mathsf{idx}(c, \langle i_0, \ldots, i_{c-1}\rangle, S)$, where the prefix $P_q = \langle D[0], \ldots, D[q-1]\rangle$, with $q = n/c$, is a repeated prefix in D, S is an optimal type path representation for the prefix P_q and the indices $i_0, \ldots, i_{c-1}$ are such that `Flatten` $(T,0) = D$; or
- $T = \mathsf{idxbuc}(c, d, \langle i_0, \ldots, i_{c-1}\rangle, \langle b_0, b_1, \ldots, b_{c-1}\rangle, S)$, the prefix P_q is a repeated prefix in D, S is an optimal type path representation for P_q and the prefix P_q is repeated b_i times with stride d in the i-th subsequence starting at index i_i.

For convenience, we will write $D = T$ to state that the type path T represents the displacement sequence D.

From the above observations it is clear that repeated prefixes play a key role when constructing optimal type path representations. A straightforward approach to find all repeated prefixes is to simply check for each divisor q of n whether the prefix of length q is repeated in D. While the check for a given divisor can trivially be done in linear time, the number of divisors $d(n)$ can be somewhat large. Let $n = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$ be the prime factor decomposition of n. By the fundamental theorem of arithmetic, $d(n) = (a_1 + 1) \ldots (a_k + 1)$. As shown in [5, pages 219–222]

$$d(n) = e^{O\left(\frac{\log n}{\log \log n}\right)} \quad .$$

Although this straightforward approach directly leads to a sub-quadratic (more precisely: $O\left(n n^{\frac{1}{\log \log n}}\right)$) time procedure to compute all repeated prefixes, it does not seem to allow for an $O(n \log n)$ time algorithm, since the upper-bound on the number of divisors is sharp. The next section presents a procedure for computing the repeated prefixes of a displacement sequence more efficiently.

3.1 Repeated Prefixes

We now present our algorithm for efficiently computing all repeated prefixes of a displacement sequence, which proves the following Lemma. The procedure is a key ingredient for solving the Basic Type Path Reconstruction Problem efficiently.

Lemma 1. *For any displacement sequence D of length n, all repeated prefixes of D can be found in $O(n \log n / \log \log n)$ time and $O(n)$ space.*

For a given q, we define the set R_q as

$$R_q = \{p \mid p \text{ divides } q \;\wedge\; P_p \text{ is a repeated prefix in } D\}$$

and the set R as $R = \bigcup_{q|n} R_q$. Given a displacement sequence D of length n and a non-trivial divisor q of n, Algorithm 5 computes in linear time the set R_q. We give a simple example to illustrate this rather technical notion and to highlight the difference between a set R_q and the set R: Assume a displacement sequence D of length $n = 24$ where, in addition to the trivially repeated prefixes of length 1 and 24, the prefixes of length 6, 8 and 12 are repeated. For $q = 12$, the algorithm finds $R_{12} = \{1, 6, 12\}$, that is, all repeated prefixes of length p where p is a divisor of q. However, the repeated prefix of length 8 is not found, since 8 is not a divisor of 12. To find the repeated prefix of length 8, Algorithm 5 needs to be run with parameter $q = 8$. This intuition is formalized in the following lemma.

Lemma 2. *Let D be an arbitrary displacement sequence of length n and q be an arbitrary divisor of n. All prefixes repeated in D that are of length p, where p is a divisor of q (i.e., the set R_q), can be found in linear time.*

To find the whole set R of repeated prefixes, we need to ensure that all possible divisors of n are covered. This is done in Algorithm 3 by applying the above observation for those divisors of n that contain all but one of its distinct prime factors. More formally, given the prime factor decomposition $n = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$ of n, let $q_i = n/p_i$ for $1 \le i \le k$ and apply Lemma 2 for each q_i. The correctness of this approach and the claimed time and space bounds are shown in the proofs of Lemma 2 and Lemma 1 in the remainder of this section. While the algorithms presented in this section are quite short and straightforward to implement, the proofs are somewhat technical and require careful attention to detail.

Algorithm 3: Finding all repeated prefixes.

```
Input: Displacement sequence $D$ of length $n$.
Output: $r_q = true$ if the prefix of length $i$ is repeated
        in $D$.
Function AllRP($D$, $n$)
    // prime factorization of $n$
    $f \leftarrow 2$; $n' \leftarrow n$; $d \leftarrow 0$;
    $all \leftarrow 0$; $distinct \leftarrow 0$;
    while $f \le n'$ do
        if $n' \bmod f = 0$ then
            if $f \ne d$ then
                $d \leftarrow f$;
                $distinct$.add($d$);
            $all$.add($f$);
        else
            $f$++;
    // initialization
    for $i \leftarrow 1$; $i \le n$; $i$++ do
        $r_i \leftarrow false$;
    // Find all repeated prefixes
    for $i \leftarrow 1$; $i < |distinct|$; $i$++ do
        $factors \leftarrow all$;
        Remove one occurrence of $distinct[i]$ from
        $factors$;
        AllRPDivisor($D$, $n$, $n/distinct[i]$, $factors$);
    // trivially repeated prefix
    $r_n \leftarrow true$;
```

We start with a presentation of Algorithm 3, which uses Algorithm 5 as a black box to solve the problem given in Lemma 2. Given a displacement sequence D of length n, it sets $r_q = true$ if and only if the prefix of length q is repeated in D. The algorithm first factorizes n, the length of the input sequence. The variable *all* is used to store all prime factors

Algorithm 4: Finding the longest potentially repeated prefix in the postfix starting at q of a displacement sequence.

```
Input: Displacement sequence $D$ of length $n$ and an
       integer $q$ that is a divisor of $n$.
Output: Largest $p$ s.t. $p \mid q$ and the prefix of length $p$
        is potentially repeated in $D$ w.r.t. $q$.
1 Function LongestInPostfix($D$, $n$, $q$)
2     for $i \leftarrow 1$; $i < n/q$; $i$++ do
3         for $j \leftarrow 1$; $j < q$; $j$++ do
4             if $D[iq + j] - D[iq] \neq D[j] - D[0]$ then
5                 $D' \leftarrow D[iq + j, n - 1]$;
6                 $n' \leftarrow n - iq - j$;
7                 $q' \leftarrow \gcd(q, j)$;
8                 return LongestInPostfix($D'$, $n'$, $q'$);
9     return $q$;
```

of n in ascending order, while the variable *distinct* keeps track of its unique prime factors (in arbitrary order). Given the prime factor decomposition of n, $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$, *all* contains a_1 times the prime factor p_1, followed by a_2 occurrences of p_2 and so on, whereas *distinct* contains each of $p_1, p_2, \dots, p_k$ exactly once. We make no claim that this approach is particularly efficient, but, as the analysis will show, its asymptotic runtime behavior suffices for the purposes of this paper. The algorithm then proceeds to find all repeated prefixes by applying Lemma 2 once for each distinct prime factor p_i of n, that is, by running Algorithm 5 once with parameter $q = n/p_i$ to compute all repeated prefixes that are of length p, where p is a divisor of q. The prime factor decomposition of q (which can easily be obtained by removing one occurrence of p_i from the prime factor decomposition of n) is passed to Algorithm 5, which requires that the first element may be accessed and removed in constant time. Therefore, the sets *all* and $factors$, which contain all prime factors of n and q respectively, are assumed to be implemented as linked lists. Correctness and the claimed time and space bounds are shown in the proof of Lemma 1 at the end of this section, after a detailed presentation of Algorithm 5.

The following intuitive corollary makes it clear that once some repeated prefix has been found, other (but not all) shorter repeated prefixes can be found easily:

Corollary 1. *Assume an n-element displacement sequence D s.t. the prefix P_q, $1 < q \leq n$, is a repeated prefix. If the prefix P_p, for $1 \leq p < q$ is repeated in the prefix P_q, P_p is also repeated in D.*

Proof. By the definition of repeated prefixes, we have that

1. $\forall i,\ 1 \leq i < n$: $i \bmod q = 0 \vee S[i \bmod q] = S[i]$ and
2. $\forall j,\ 1 \leq j < q$: $j \bmod p = 0 \vee S[j \bmod p] = S[j]$.

Since $q \mid n$ and $p \mid q$, we get that $\forall i, 1 \leq i < n:\ i \bmod p = 0 \vee S[i \bmod p] = S[i]$, i.e., the prefix of length p is repeated in D. □

Given a displacement sequence D of length n and a divisor q of n, `LongestInPostfix` (Algorithm 4) finds the longest *potentially* repeated prefix P_p, where p is a divisor of q. We say that a prefix P_p is potentially repeated in D w.r.t. q if the prefix P_p being repeated in the prefix P_q implies that P_p is also a repeated prefix in D. The usefulness of this rather technical notion will become clear once we discuss Algorithm 5.

The function `LongestInPostfix` checks for every segment $D[iq, (i+1)q - 1]$, $0 \leq i < n/q - 1$, whether the prefix of length q is repeated. If this is the case, the prefix P_q is a repeated prefix in the postfix $D[q, n-1]$ and therefore also the longest potentially repeated prefix w.r.t. q. Since it is also repeated in the prefix $P_q = D[0, q-1]$, the potentially repeated prefix P_q is trivially a repeated prefix in D.

If the condition for repeated prefixes is not met at index j of the i-th segment $D[iq, (i+1)q - 1]$, the procedure proceeds recursively to find the longest prefix of length p repeated in the postfix $D[iq + j, n-1]$, where p is a divisor of $\gcd(q, j)$. The condition being met up to index $iq + j - 1$ implies that the prefix P_q is repeated in the segment $D[q, iq + j - 1]$ (the requirement that q must be a divisor of the length of this segment can be ignored here). Due to Corollary 1, it suffices to check whether the prefix P_p is repeated in the prefix P_q to determine whether or not P_p is repeated in the segment $D[q, iq + j - 1]$ (this check will be done by Algorithm 5), i.e., P_P is potentially repeated in the segment $D[q, iq + j - 1]$. Since furthermore P_p is repeated in the postfix starting at $iq + j$, P_p is potentially repeated in the postfix starting at index q. This extends to divisors r of p: if p is potentially repeated in the postfix $D[q, n-1]$ and the prefix P_p is repeated in $D[0, q-1]$ (and therefore also in $D[0, p-1]$), P_p is repeated in D.

The condition not being met at index $iq + j$ means that $D[iq + j] - D[iq] \neq D[j] - D[0]$, implying that the length P of the longest repeated prefix in D, where $p \mid q$, is at most j. Since we are only interested in those prefixes of length p where $p \mid q$, we find that $p \leq \gcd(q, j)$. Since the prefix of length 1 is a repeated prefix in any displacement sequence and 1 is always a divisor of q, the procedure is guaranteed to find a longest repeated prefix.

Algorithm 5 uses `LongestInPostfix` to efficiently compute all repeated prefixes of length p, where p is a divisor of q (i.e., the set R_q). If the prefix of length q is potentially repeated in the postfix $D[q, n-1]$, it is trivially a repeated prefix in D (since the prefix $D[0, q-1]$ is repeated exactly once in that sequence). The algorithm proceeds by recursively computing all repeated prefixes in the repeated prefix of length q. Note that the longest possible, non-trivial repeated prefix is of length less than or equal to q over the smallest prime factor of q. Due to Corollary 1, any prefix repeated in the prefix P_q is also repeated in the displacement sequence D. If the longest potentially repeated prefix is of length $q_l < q$, we need to check whether the prefix of length q_l is repeated in the prefix of length q. This is done by recursively computing all repeated prefixes of the prefix $D[0, q-1]$. The prime factors of q (with multiplicity) are passed to Algorithm 5 in the initial call and the algorithm keeps track of the prime factorization of the parameter q.

Proof of Lemma 2. The procedure `LongestInPostfix` requires $O(n)$ time, due to the recursive call proceeding with the postfix $D[iq + j, n-1]$ in case the condition for repeated prefixes is not met for an element at index $iq + j$. Note that when called with parameter q for a displacement sequence

Algorithm 5: Finding all prefixes of length p, where p is a divisor of q, that are repeated in the displacement sequence D of length n.

```
Input: Displacement sequence $D$ of length $n$; an integer $q$ that is a non-trivial divisor of $n$ and the distinct prime factors of $q$ in ascending order
Output: $r_q$ is set to $true$ if the prefix of length $q$ is repeated in $D$.
Function AllRPDivisor($D$, $n$, $q$, $factors$)
    if $q == 1$ then
        $r_1 \leftarrow true$;
        return;
    $q_l \leftarrow$LongestInPostfix($D$, $n$, $q$);
    if $q_l == q$ then
        $r_q \leftarrow true$;
        $d \leftarrow$ smallest prime factor of $q$;
        $q_l \leftarrow q/d$;
        Remove one occurrence of $d$ from $factors$;
    AllRPDivisor($D$, $q$, $q_l$, $factors$);
```

of length n, the procedure does not do the check for any element in the prefix $D[0, q-1]$. We use this fact to account for any subsequent calls of `LongestInPostfix` that happen in Algorithm 5: The algorithm recurses on the prefix of length (at most) q, for which it calls `LongestInPostfix` with a displacement sequence of length q. Therefore, during an execution of Algorithm 5, the procedure `LongestInPostfix` performs the check on line 4 of Algorithm 4 for each element at most once. The smallest prime factor of q is stored as the first element of $factors$. Since $factors$ is implemented via a linked list, the smallest prime factor of q can be retrieved and removed in constant time. □

Proof of Lemma 1. The prime factorization as outlined in Algorithm 3 is feasible in linear time. Although this approach is clearly not the most efficient, its runtime is linear in the size of the whole input. (Although not in the size of the binary representation of the number we want factored, so the previous statement does not contradict the fact that prime factorization is not known to be solvable in polynomial time.) Note that any number is a composite of at most $\log_2(n)$ prime factors (the worst case is n being a power of 2) and thus the size of both all and $distinct$ is $O(\log n)$.

Algorithm 3 applies Lemma 2 (implemented as Algorithm 5) for each $q_i = n/p_i$, where the p_i are the distinct prime factors. Note that any non-trivial divisor q of n is equal to n/p, where p a divisor of n that contains at least 1 of n's prime factors. Since Algorithm 5 with parameter q finds all repeated prefixes of length q' where $q' \mid q$, calling it for all $q_i = n/p_i$ ensures that all divisors of n are covered.

As discussed above, Algorithm 5 requires $O(n)$ steps on displacement sequences of length n. The number of executions is bounded by the number of distinct prime factors of n, which, due to Robin [8], is $O(\log n/\log\log n)$, implying the claimed runtime bound. Apart from the two linked lists of size $O(\log n)$, the algorithm requires $O(n)$ variables r_i, and the total required space is therefore $O(n)$. □

3.2 Basic Type Path Reconstruction

We now present our algorithm for solving the type reconstruction problem for vector and index-block constructors.

Theorem 1. *The* Basic Type Path Reconstruction Problem *can be solved in $O(n \log n/\log\log n)$ time and $O(n)$ additional space.*

The following definitions and observations are derived from analogous observations for the Basic Type Reconstruction Problem as discussed in [1], which includes the strc (`MPI_Type_create_struct`) constructor. We therefore present them in an intuitive, if rather informal way. Removing strc from the set of considered constructors simplifies these observations considerably and formal proofs can easily be obtained by following the rigorous proofs given in [1] and restricting them to the idx, vec and leaf constructors.

We first solve the problem for the special case of displacement sequences in *normal form*, which for a displacement sequence D of length n is defined as $\hat{D}[i] = D[i] - D[0]$, for all i, $0 \le i < n$. Observe that the structure of a displacement sequence does not change by normalizing it. In particular, any prefix that is a repeated (strided) prefix in D is also a repeated (strided) prefix in $\hat{D}$ and vice versa. Lemma 5 at the end of this section shows how to efficiently construct an optimal type path representation for an arbitrary displacement sequence D out of an optimal solution for its normal form $\hat{D}$.

Any displacement sequence can be represented by *shifting* its normal form by the value $D[0]$, i.e., $D = \mathsf{idx}(1, \langle D[0]\rangle, \hat{D})$. We call an index node $\mathsf{idx}(c, \langle i_0, \dots\rangle, C)$ with $i_0 \neq 0$ a *shifted node*, where $s = i_0$ is called the node's *shift value*. Given a type path T that contains an idx node N and represents a displacement sequence D, adding some value s to all indices of N shifts D by s. A *nice type path* is a type path that contains at most one shifted index node, which, if it exists, is the first idx node on the path from the root to the leaf node.

Lemma 3. *For any type path T, a nice type path representation $\tilde{T}$ of less or equal cost exists.*

Proof. Given a type path T representing a displacement sequence D, a nice type path $\tilde{T}$ can easily be constructed by shifting all idx nodes by their respective shift values. More formally, let the index nodes occurring in T be numbered as $N^1, N^2, \dots$. Replace each index node $N^j = \mathsf{idx}(c^j, \langle i_0^j, \dots, i_{c^j-1}^j\rangle, S^j)$ in T with $N^k = \mathsf{idx}(c^j, \langle 0, i_1^j - i_0^j, \dots, i_{c-1}^j - i_0^j\rangle, S^j)$ and let $s = \sum_{N^j} i_0^j$ be the (positive) sum of all shift values. In total, the represented displacement sequence is shifted by $-s$ and the constructed type path represents the normalized displacement sequence $\hat{D}$, from which a nice type path representation of D can be obtained by shifting the top-most idx node by s. Since none of the above operations increases the cost of any of the nodes in T and no nodes are added or removed to obtain $\tilde{T}$, $\tilde{T}$ is of the same cost as T. Note that a type path that does not contain any idx nodes by definition is a nice type path. □

We now characterize the structure of optimal type paths. As discussed above, representations of the form $\mathsf{idx}(1, \langle 0\rangle, D)$ and $\mathsf{vec}(1, 0, D)$ are redundant and thus cannot be part of an optimal type path representation. Corollary 3 directly implies that an optimal type path contains at most one index node with count 1, i.e., a node of the form $\mathsf{idx}(1, \langle i_0\rangle, \dots)$, which is necessarily a shifted index node. Note that any index node corresponding to a constructor $\mathsf{idx}(c, \langle \ldots \rangle, C)$, with C of length m and $c \geq 2$, represents a displacement sequence of length at least $2m$. This holds analogously for vector nodes and therefore for all except at most two nodes in an optimal type path (the two possible exceptions being a shifted index node with count 1 and the leaf node). The height of an optimal type path therefore is $O(\log n)$.

Algorithm 6: Construction of optimal type path representations for displacement sequences in normal form.

```
    Input: Displacement sequence $\hat{D}$ of length $n$ in normal
           form with base type $B$
    Output: Optimal type path representation of $\hat{D}$.
1   Function Typepath($\hat{D}$, $n$, $B$)
2       // Step 1: Trivial optimal representation
           for $P_1$
3       $T_1 = \mathsf{leaf}(B)$;
4       // Step 2: Find all repeated prefixes
5       AllRP($\hat{D}$, $n$);
6       // Step 3: Find strided prefixes
7       for $q \leftarrow 1$; $q < n$; $q$++ do
8           if $r_q \neq true$ then continue;
9           $s_q \leftarrow q$;
10          $s \leftarrow D[q] - D[0]$;
11          while $s_q + q < n$ and $D[s_q + q] - D[s_q] = s$ do
12              $s_q \leftarrow s_q + q$;
13      // Step 4: Construct optimal representation
           for each repeated prefix
14      foreach $q \in \{i \mid r_i = true,\ 2 \leq i \leq n\}$ do
15          $c_{best} \leftarrow \infty$;
16          foreach $p \in \{i \mid r_i = true,\ 1 \leq i < q\}$ do
17              $c_{\mathsf{vec}} \leftarrow K_{\mathsf{vec}} + cost(T_p)$;
18              if $q \leq s_p$ and $c_{\mathsf{vec}} < c_{best}$ then
19                  $stride \leftarrow \hat{D}[p] - \hat{D}[0]$;
20                  $T_q \leftarrow \mathsf{vec}(q/p, stride, T_p)$;
21                  $c_{best} \leftarrow c_{\mathsf{vec}}$;
22              $c_{\mathsf{idx}} \leftarrow K_{\mathsf{idx}} + (q/p)K_{\mathrm{lookup}} + cost(T_p)$;
23              if $c_{\mathsf{idx}} < c_{best}$ then
24                  // $indices \leftarrow \langle \hat{D}[0], \hat{D}[p], \ldots,$
                       $\hat{D}[(q/p - 1)p] \rangle$
25                  $T_q \leftarrow \mathsf{idx}(q/p, indices, T_p)$;
26                  $c_{best} \leftarrow c_{\mathsf{idx}}$;
27      return $T_n$;
```

The dynamic programming principle applies, since any subtree of an optimal type path must itself be an optimal representation of the respective prefix. These observations allow for an efficient dynamic programming algorithm to solve the Basic Type Path Reconstruction Problem, which is given in Algorithm 6 and proceeds in four steps. In steps 1 to 3, the algorithm computes all repeated prefixes of D and gathers information about possible representations (e.g., if a prefix can be represented by a strided repetition of another, shorter repeated prefix). Starting with the prefix of length 2, the algorithm in the fourth step iteratively constructs an optimal type path for each repeated prefix by using this information as well as already computed optimal representations for shorter repeated prefixes. Note that for a normalized displacement sequence, the prefix of length 1 is the sequence $\langle 0 \rangle$ and therefore the optimal representation is via a leaf node. The algorithm thus directly sets $T_1 = \mathsf{leaf}(B)$ in step 1.

In the second step, all repeated prefixes of $\hat{D}$ are computed as shown in the previous section. Recall that a repeated prefix of length q allows for representations via an idx node. In the third step, it computes for each repeated prefix P_q the longest prefix P_p, $q \leq p \leq n$ s.t. P_q is a strided prefix in P_p.

In the final step, an optimal type path representation for the displacement sequence $\hat{D}$ is found by iteratively computing an optimal type path representation T_p for each repeated prefix P_p. Each node stores the cost of the type path rooted at the node. When a new type path T_q for the the prefix P_q is constructed out of a prefix P_p, the cost of T_p can be computed in constant time by adding the cost of the constructed root node to the cost $cost(T_p)$ of the subtype. The information gathered by the first three steps allows for an efficient construction of an optimal representation for a prefix of length q out of already computed optimal representations for repeated prefixes of length less than q. Since only repeated prefixes can ever be part of a type path representation, prefixes that are not repeated in $\hat{D}$ need not be considered. As discussed at the begin of Section 3, the optimal type path representation for a repeated prefix of length q, $1 < q \leq n$, is either a single leaf node, or an idx or vec node with an optimal representation of a shorter repeated prefix as the subtype and appropriate parameters. The most cost-efficient representation is found by enumerating all possible representations of the prefix of length q, as done in the body of the nested for-loop. If two prefixes P_p and P_q, $p \leq q$ are both repeated in $\hat{D}$, it is clear that P_p is also repeated in P_q and that therefore a representation of P_q via an idx node of the form $P_q = \mathsf{idx}(q/p, \langle \ldots \rangle, P_p)$ is possible. The prefix of length 1 is a repeated prefix in any displacement sequence and thus the algorithm is guaranteed to find a valid representation for all repeated prefixes. If the prefix P_p is a strided repetition the prefix P_q, P_q can alternatively be represented via a vec node as $P_q = \mathsf{vec}(q/p, \hat{D}[p] - \hat{D}[0], P_p)$. Note that r_n was set to $true$ by Algorithm 3 (the prefix of length n is by definition a repeated prefix), and thus an optimal type path representation for the displacement sequence $\hat{D}$ is constructed in the last iteration.

The first step implicitly assumes constant update times for the variables r_i, which can be guaranteed by storing them in an n-element array. In the fourth step however we cannot afford to scan an array of size n for those r_i that were set to $true$, since that would lead to an $O(n^2)$ time bound for the two nested loops. This can be avoided by copying only the required information to a new, smaller array that contains only the values i for which $r_i = true$. We have omitted this detail in Algorithm 6 to simplify the presentation.

Note that the array of indices for an idx node can easily be constructed when the count of the index node is known and we employ a standard trick for dynamic programming algorithms to meet the time and space bounds. We do not need to store the index array with each constructed idx node, but defer this step until an optimal type path for $\hat{D}$ was found. The missing data for all idx nodes in this type path can easily be filled in by traversing the constructed type path.

LEMMA 4. *Theorem 1 holds for displacement sequences in normal form, i.e., for any normal-form displacement sequence $\hat{D}$ of length n, Algorithm 6 constructs an optimal type path representation in $O(n \log n / \log\log n)$ time and $O(n)$ space.*

PROOF. As shown in Section 3.1, all the repeated prefixes of $\hat{D}$ can be computed using $O(n \log n / \log\log n)$ time and $O(n)$ space. The second step requires $O(n/q)$ time for each repeated prefix of length q. Using the upper bound for the divisor summatory function due to Gronwall [2], this is

$$\sum_{q|r_q=true} n/q \leq \sum_{q|n} q = O(n \log n / \log\log n) \quad .$$

The body of the inner for-loop of the fourth step requires only constant time due to the information gathered in the previous steps and the trick of not computing the array of indices for idx nodes as outlined above. For each q, it is executed at most $O(q)$ times (a rough upper bound for the number of repeated prefixes of q, which is at most equal to the number of divisors of q). The remainder of the outer for-loop's body trivially requires constant time and we can again apply Gronwall's upper bound for the divisor summatory function to see that the fourth step requires

$$\sum_{q|n} O(q) = O(n \log n / \log\log n)$$

time.

To store the optimal type path representation T_q for the prefix P_q, it suffices to store the kind of and required parameters for the root node (minus the array of indices in case it is an idx node) plus a pointer to the representation of the subtree and thus $O(n)$ space is requires to store the representations of all repeated prefixes. The algorithm additionally requires $O(n)$ variables r_i and s_i and the required space is thus $O(n)$ in total. □

LEMMA 5. *An optimal type path representation for any displacement sequence D of length n can be constructed out of optimal type path representations $\hat{T}_q$ for all repeated prefixes of the normalized displacement sequence $\hat{D}$ in $O(n)$ time.*

PROOF. By Lemma 3, a cost-equivalent nice type path representation $\tilde{T}$ of D exists for any optimal type path representation T of D. By assumption, an optimal nice type path $\hat{T} = \hat{T}_n$ representing $\hat{D}$ exists. To construct an optimal representation of D, find the top most index node $N = \mathsf{idx}(c, \langle i_0, \ldots, i_{c-1}\rangle, S)$ in $\hat{T}$ and add the displacement sequence's shift $s = D[0]$ to all its indices, i..e, $N = \mathsf{idx}(c, \langle i_0 + s, \ldots, i_{c-1} + s\rangle, S)$. Note that $\tilde{T}$ has the same cost as $\hat{T}$ and is therefore optimal. Traversing the whole type path requires only $O(\log n)$ time and the list of indices in an index node is at most of length n. This step is therefore feasible in $O(n)$ time.

If such an index node does not exist, it follows directly from Lemma 3 that the optimal representation of D is of the form $\tilde{T} = \mathsf{idx}(n/q, \langle \ldots \rangle, \tilde{T}_q)$ for some divisor q of n (including the trivial divisors 1 and n). As in Algorithm 6, the representation of least-cost among this set of possible representations can be found in linear time, proving the claimed time bound. □

PROOF OF THEOREM 1. The BASIC TYPE PATH RECONSTRUCTION PROBLEM for a displacement sequence D of length n can be solved by computing an optimal type path representation for the normalized sequence $\hat{D}$ using Algorithm 6 and applying the post-processing step outlined in Lemma 5. The claimed time and space bounds follow directly from Lemma 4 and Lemma 5. □

Algorithm 7: Constructing an optimal representation of P_q with idxbuc as root and the prefix P_p as subtype.

```
1  $E \leftarrow \{D[(i+1)q] - D[iq] \mid 0 \leq i < p/q - 1\}$;
2  sort $E$;
3  $d \leftarrow$ most frequently occurring element in $E$;
4  $c \leftarrow$ number of occurrences of $d$ in $E$;
5  $c_{\mathsf{idxbuc}} \leftarrow K_{\mathsf{idxbuc}} + 2(p/q - c)K_{lookup}$;
6  if $c_{\mathsf{idxbuc}} < c_{best}$ then
7      $indices[0] \leftarrow 0$;
8      $j \leftarrow 0$;
9      for $i \leftarrow 0$; $i < q/p - 1$; $i$++ do
10         if $\hat{D}[(i+1)p] - \hat{D}[ip] = d$ then
11             $b[j] \leftarrow b[j] + 1$;
12         else
13             $j \leftarrow j + 1$;
14             $indices[j] \leftarrow \hat{D}[(i+1)p]$;
15     $T_q \leftarrow \mathsf{idxbuc}(p/q - c, d, indices, b, T_p)$;
16     $c_{best} \leftarrow c_{\mathsf{idxbuc}}$;
```

3.3 Extended Type Path Reconstruction

In this section, we show how to incorporate also `MPI_Type_indexed` into the type reconstruction algorithm.

THEOREM 2. *The* EXTENDED TYPE PATH RECONSTRUCTION PROBLEM*can be solved in $O(n^2 \log^2 n)$ time.*

We argue informally that the properties and observations made in Section 3 regarding (optimal) type paths still hold for type paths including the idxbuc constructor. See also [1], where analogous properties for type trees (including the strc constructor) were proved.

The idxbuc constructor too relies heavily on repeated prefixes and therefore the EXTENDED TYPE PATH RECONSTRUCTION PROBLEM can be solved by extending Algorithm 6 to additionally check all possible representations of a repeated prefix q via an idxbuc constructor. Note that if a prefix P_q is repeated in a displacement sequence D, there always exists a representation of D via an idxbuc node using P_q as a subtype (though we might have all $b_i = 1$, in which case this representation is a more redundant way of using an idx node). Observe that using a bucket stride $d = D[q] - D[0]$ joins the two d-strided segments $D[0, q-1]$, $D[q, 2q-1]$ into one bucket. The most cost-efficient representation using an idxbuc node and a prefix P_q as subtype is therefore one that joins as many segments as possible into one bucket, since the cost of an idxbuc node depends on the number but not on the size of the buckets.

Algorithm 7 gives a procedure for constructing such an optimal representation. It can directly be used in Algorithm 6 by adding it to the body of the inner for-loop (e.g., after line 26). We however make no claim here that this approach is particularly efficient and expect that the asymptotic runtime can be reduced significantly by the use of a clever preprocessing step analogous to vec constructors (step 3 in Algorithm 6).

Proof of Theorem 2. The cost of Algorithm 7 is dominated by the sorting of $q/p = O(q)$ elements (the remaining constant number of steps can all be implemented in a single scan of P_q). Applying the upper bound for the divisor summatory function for the inner for-loop, we have that for a prefix P_q $O(q \log q / \log \log q)$ elements are sorted to find the most-cost efficient representation via an `idxbuc` node. Thus in total

$$\sum_{q|n} O(q \log q / \log \log q) \leq \sum_{q|n} O(q \log q) \leq \sum_{q=1}^{n} O(q \log q) = O(n^2 \log n)$$

elements need to be sorted, which can be done in $O(n^2 \log^2 n)$ steps. $\square$

Further constructors that rely on repeated prefixes, e.g., the *bucket type* as discussed in [13] or the *triangular type* that occurred in [12], can be incorporated into our algorithm in a similar manner to the `idxbuc` constructor. The properties of (optimal) type paths as discussed in Section 3 can easily be extended to type paths including the two additional constructors and therefore optimality of the constructed type paths can also be shown. However, as for the `idxbuc` constructor, further work is required to improve the asymptotic runtime of the resulting algorithm.

4. CONCLUSION

This paper explored further the problem of finding good representations of MPI derived datatypes that are efficient in terms of space and processing costs. We showed that the restriction to the MPI type vector and index-block type constructors makes it possible to solve the type reconstruction problem in close to linear time with a practical and easily implementable algorithm. We provided a theoretical framework for the construction of provably optimal type paths for type constructors relying on repeated prefixes (such as `MPI_Type_vector` and `MPI_Type_indexed`). The paper improves over the result in [11], also in the sense that there is no need for a separate, tedious to implement dynamic programming step over type paths. We further showed how to extend the algorithm to handle also the `MPI_Type_indexed` constructor, as well as additional constructors not found in MPI for describing bucket and triangular data layouts. It makes sense to continue to investigate both the processing and the reconstruction for such additional constructors, perhaps to consider them for future extensions of the MPI standard. We point out that the algorithm given in [11] does not seem to allow for the inclusion of further constructors, at least not in the straightforward way outlined here.

We focused exclusively on the type reconstruction problem of finding the best possible derived datatype for a given, explicitly described, linear-sized application data layout. The related *type normalization problem* of transforming a given derived datatype into a cost-optimal representation can trivially be solved by first flattening the derived type as shown in Algorithm 2 and then reconstructing for the resulting displacement sequence. This trivial solution is clearly undesirable since a derived datatype can be significantly more concise than the displacement sequence it represents, making the complexity of this approach hard to control. We do not know whether our algorithms of this paper can be used to give non-trivial results for type normalization. The currently best known approach for type normalization is that of [11].

5. REFERENCES

[1] R. Ganian, M. Kalany, S. Szeider, and J. L. Träff. Polynomial-time construction of optimal tree-structured communication data layout descriptions. CoRR abs/1506.09100, 2015.

[2] T. Gronwall. Some asymptotic expressions in the theory of numbers. *Transactions of the American Mathematical Society*, 14(1):113–122, 1913.

[3] W. D. Gropp, T. Hoefler, R. Thakur, and J. L. Träff. Performance expectations and guidelines for MPI derived datatypes: a first analysis. In *Recent Advances in Message Passing Interface. 18th European MPI Users' Group Meeting*, volume 6960 of *Lecture Notes in Computer Science*, pages 150–159. Springer, 2011.

[4] F. Kjolstad, T. Hoefler, and M. Snir. Automatic datatype generation and optimization. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 327–328, 2012.

[5] E. Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*, volume 1. Teubner, 1909.

[6] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.1*, June 4th 2015. www.mpi-forum.org.

[7] T. Prabhu and W. Gropp. DAME: A runtime-compiled engine for derived datatypes. In *Recent Advances in the Message Passing Interface (EuroMPI)*, 2015. To appear.

[8] G. Robin. Estimation de la fonction de Tchebychef θ sur le k-ième nombre premier et grandes valeurs de la fonction $\omega(n)$ nombre de diviseurs premiers de n. *Acta Arithmetica*, 42(4):367–389, 1983.

[9] R. Ross, N. Miller, and W. D. Gropp. Implementing fast and reusable datatype processing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 404–413. Springer, 2003.

[10] T. Schneider, F. Kjolstad, and T. Hoefler. MPI datatype processing using runtime compilation. In *Recent Advances in the Message Passing Interface, 20th European MPI Users's Group Meeting (EuroMPI)*, pages 19–24, 2013.

[11] J. L. Träff. Optimal MPI datatype normalization for vector and index-block types. In *Recent Advances in Message Passing Interface. (EuroMPI/ASIA)*, pages 33–38, 2014.

[12] J. L. Träff, F. Lübbe, A. Rougier, and S. Hunold. Isomorphic, sparse MPI-like collective communication operations for parallel stencil computations. In *Recent Advances in the Message Passing Interface (EuroMPI)*, 2015. To appear.

[13] J. L. Träff and A. Rougier. Zero-copy, hierarchical gather is not possible with MPI datatypes and collectives. In *Recent Advances in Message Passing Interface. (EuroMPI/ASIA)*, pages 39–44, 2014.