2016 IEEE International Parallel and Distributed Processing Symposium

Polynomial-time Construction of Optimal MPI Derived Datatype Trees

Robert Ganian*, Martin Kalany†, Stefan Szeider*, Jesper Larsson Träff†
*Algorithms and Complexity Group
Vienna University of Technology (TU Wien), Vienna, Austria
{rganian,sz}@ac.tuwien.ac.at
†Parallel Computing Group
Vienna University of Technology (TU Wien), Vienna, Austria
{kalany,traff}@par.tuwien.ac.at

***Abstract*—The derived datatype mechanism is a powerful, integral feature of the Message-Passing Interface (MPI) for communicating arbitrarily structured, possibly non-consecutive and non-homogeneous application data. MPI defines a set of derived datatype constructors of increasing generality, which allows to describe arbitrary data layouts in a reasonably compact fashion. The constructors may be applied recursively, leading to tree-like representations of the application data layouts. Efficient derived datatype representations are required for MPI implementations to efficiently access and process structured application data. We study the problem of finding tree-like representations of MPI derived datatypes that are *optimal in terms of space and processing cost*.**

More precisely, we consider the so-called MPI TYPE TREE RECONSTRUCTION PROBLEM of determining a least-cost tree-like representation of a given data layout for a given set of constructors. In an additive cost model that accounts for the space consumption of the constructors and lower-bounds the processing costs, we show that the problem can be solved in *polynomial time* for the *full* set of MPI datatype constructors. Our algorithm uses dynamic programming and requires the solution of a series of shortest path problems on an incrementally built, directed, acyclic graph.

***Keywords*-MPI, derived datatypes, type reconstruction, dynamic programming**

I. INTRODUCTION

The Message-Passing Interface (MPI) [1] is a paradigm example of a parallel communication interface with a generic mechanism for describing non-consecutive and non-homogeneous application data to allow the implementation to communicate such data efficiently. These so-called *derived datatypes* provide application programmers with means for specifying application data layouts (e.g., submatrices, diagonals and columns) and delegating the actual data accesses to the MPI library implementation. Provided the library handles derived datatypes well, this is potentially more portable than hand-written, application and system specific code for copying, packing and unpacking non-consecutive, structured data before and after communication operations. Derived datatypes can be used with any MPI communication model (point-to-point, one-sided, collective) and give the library implementation possibilities that are difficult to exploit at the application level, for instance pipelining large data volumes and directly exploiting hardware features for, e.g., strided communication. In the best case, explicit and intermediate copies can be avoided altogether, leading to so-called *zero-copy* implementations [2], [3]. Run-time processing of derived datatypes has been studied intensively in the MPI community (see [4]–[6] for recent examples and references), and MPI library implementers have invested considerable effort in making derived datatypes effcient enough to be used in real applications [7].

In this paper we study an orthogonal aspect of derived datatypes, and will be concerned with the space required to represent derived datatypes. This is important for minimizing storage used by derived datatypes (important also when transmitting derived datatype descriptions as may be required in MPI one-sided communication), as well as for their efficient processing. Derived datatypes can be represented by trees (or more generally, directed acyclic graphs) of constructor nodes that capture the regularities in the data layout described by the derived datatype. The natural question is to ask for a most *concise* such tree representation of the given layout for some set of allowed constructor nodes, e.g., those found in MPI [1, Chapter 4]. Conciseness measures the total space required for the tree in the form of offsets, repetition counts, strides, and so on, and lower bounds the processing time since all this information has to be looked up when processing the derived datatype and accessing the data. In the MPI community, this problem has been referred to as the *type reconstruction problem* [8]. A closely related problem, called *type normalization*, asks for a most concise tree describing the same layout as some given derived datatype tree. The performance impact of optimal derived datatype descriptions can be significant (see Section II), and both problems are eventually important for very high-quality MPI libraries, and would be as well in other parallel interfaces or languages supporting communication of structured, non-consecutive data. Ideally, a compiler would be able to perform both reconstruction and normalization of data layout

This work was co-funded by the European Commission through the EPiGRAM project (grant agreement no. 610598), and by the Austrian Science Fund (FWF) project P26696. Robert Ganian is also affiliated with FI MU, Brno, Czech Republic.

1530-2075/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPS.2016.13

descriptions given more or less explicitly by the application programmer with the constructs available in the parallel language [4], [5].

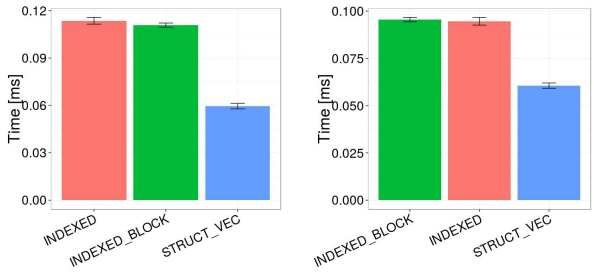
Our main, new result is to show that the type reconstruction problem of finding a most concise datatype tree describing a given data layout can be solved in polynomial time for the full set of MPI derived datatype constructors. Specifically, we give an algorithm that finds an optimally concise type tree for any given sequence of displacement/basetype pairs (a so-called MPI type map) of length n in $O(n^4)$ operations. Prior to this paper there has been little systematic work on this problem. MPI libraries have typically employed straight-forward heuristics for derived datatype normalization (see [4], [5], [9]–[11] for explicit descriptions), but with no guarantees of optimality. The problems were first formalized in [8], and results for restricted sets of constructors (essentially leaving out the `MPI_Type_create_struct` constructor) for both reconstruction and normalization were given in [6], [12].

The paper is structured as follows. We define our set of constructors and precisely formulate the type reconstruction problem in Section III. Our main result is given in Section IV, which describes our dynamic programming algorithm for type reconstruction, proves correctness and establishes the complexity bound. In Section V we discuss how the problem changes when type trees can be folded into DAGs. Section VI deals briefly with the type normalization problem. Concluding remarks are given in Section VII.

II. Type normalization performance benefits

We give a simple example that illustrates both the potential performance benefits of good derived datatype representations as well as shortcomings of current MPI library implementations. Assume an $n \times n$ integer matrix (in row-major order), of which we want to communicate the elements of the first row followed by the elements of the first column. This data layout can be described with different MPI derived datatype constructors. Using `MPI_Type_create_indexed_block`, all of the $2n-1$ base type/displacement pairs must be listed explicitly. With `MPI_Type_indexed` the row elements can be described more compactly as a single, contiguous block, but the column elements must still be listed one-by-one. Using `MPI_Type_create_struct`, the row data is described in the same way, but `MPI_Type_vector` can be used as a sub-type to describe the column data as a strided layout.

We measure the performance of these three different derived datatypes, called `INDEXED_BLOCK_TYPE`, `INDEXED_TYPE` and `STRUCT_VEC_TYPE` respectively, in a simple ping-pong benchmark with a matrix of order $n = 1000$. Fig. 1 shows the mean execution time with the 95% confidence intervals with two MPI processes at



(a) OpenMPI (b) NEC MPI

Figure 1: Ping-pong performance of different derived datatypes describing the same, given data layout.

different nodes[1]. With both MPI implementations, execution times are roughly equal for `INDEXED_BLOCK_TYPE` and `INDEXED_TYPE`, indicating that the first datatype is normalized by both libraries. A simple heuristic suffices to detect that the first 1000 elements listed explicitly by `INDEXED_BLOCK_TYPE` are in fact contiguous. With `STRUCT_VEC_TYPE`, execution times drop to around one half and to two thirds, respectively. The normalization heuristics performed by these MPI libraries therefore do not suffice to obtain the much more efficient description of the data layout. The `STRUCT_VEC_TYPE` datatype is in fact an optimal description of the given data layout in our model.

III. The type reconstruction problem

An MPI *type map* is an ordered sequence of relative (integer) displacements, each indexing a certain basic, predefined datatype (integer, char, floating point number, etc.) relative to some base address [1, Section 4.1]. Formally, a type map $M = (B, D)$ consists of a sequence of base types (the *type signature*) $B = \langle b_0, b_1, \ldots, b_{n-1} \rangle$ plus a sequence of displacements $D = \langle d_0, d_1, \ldots, d_{n-1} \rangle$. Both the base types $B[i] = b_i$ and displacements $D[i] = d_i$ are indexed from 0 to $n-1$, where n is the *length* of the type map. The base types correspond to elementary, built-in data types of the host language and are represented with MPI's predefined datatypes [1, p. 25]. The displacements are arbitrary integers, and the same displacement can appear more than once (although this will normally not be the case, and is disallowed for some uses of derived data types). Thinking of displacements as (byte) addresses, clearly any application data layout can be described by a type map. The ordering constraint (displacement sequence, *not* displacement set) implies that data are accessed in a specific order, which is important for data layouts used in communication operations. Type maps typically have structure in the form of regularities, since they arise from specific applications. This

[1]Benchmarking was done on a small, 36-node cluster at TU Wien, each node consisting of two AMD Opteron 6134 processors (2.3 GHz, 8 cores) and 32GB of memory. The code was compiled with `gcc 4.4.7` and tested with NEC MPI 1.3.1 and OpenMPI 1.8.4 libraries. The experiment was repeated 100 times for each derived datatype and MPI library.

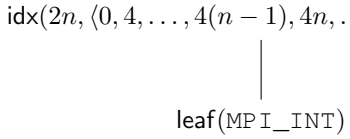
can be exploited to obtain more concise descriptions. We do this by type trees, where interior *constructor nodes* describe some ordered catenation of the layout(s) described by the child node(s). We consider the following set of constructors onto which MPI's derived datatype constructors can easily be mapped.

Definition 1 (Type constructors, type trees)**.** A *type tree* is constructed from the following five *constructor nodes*:

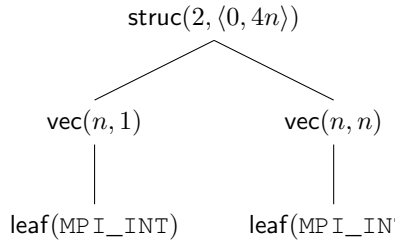
1) A *leaf* $\mathsf{leaf}(b)$ represents the base type b with displacement 0.
2) A *(homogeneous) vector* $\mathsf{vec}(c, s, L)$ with *count* c and *stride* s describes the catenation of c type maps L at relative displacements $0, s, 2s, \ldots, (c-1)s$.
3) A *(homogeneous) index* $\mathsf{idx}(c, \langle i_0, i_1, \ldots, i_{c-1}\rangle, L)$ with *count* c and *indices* $i_0, i_1, \ldots, i_{c-1}$ describes the catenation of c type maps L at relative displacements $i_0, i_1, \ldots, i_{c-1}$.
4) An *indexed bucket*, $\mathsf{idxbuc}(c, s, \langle i_0, i_1, \ldots i_{c-1}\rangle, \langle a_0, a_1, \ldots a_{c-1}\rangle, L)$, with *bucket count* c, *substride* s and bucket sizes $a_0, a_1 \ldots, a_{c-1}$, describes the catenation of c type maps at relative indices $i_0, i_1, \ldots i_{c-1}$. The i-th type map is the catenation of a_i type maps L at relative displacements $0, s, 2s, \ldots (a_i - 1)s$.
5) A *heterogeneous index*, or *struct*, $\mathsf{struc}(c, \langle i_0, i_1, \ldots, i_{c-1}\rangle, \langle L_0, L_1, \ldots, L_{c-1}\rangle)$, with *count* c describes the catenation of c type maps $L_0, L_1, \ldots, L_{c-1}$ at relative displacements $i_0, i_1, \ldots, i_{c-1}$.

Assume for example a type map consisting of 3 consecutively stored characters, i.e., a type map $L = (\langle char, char, char\rangle, \langle 0, 1, 2\rangle)$. Using L as a sub-type for an idx constructor together with count $c = 3$ and displacements $\langle 4, 10, -10\rangle$ creates the type map $(\langle char, \ldots\rangle, \langle 4, 5, 6, 10, 11, 12, -10, -9, -8\rangle)$, i.e., the idx constructor concatenates c replications of L at displacements 4, 10 and -10. Given a type map M with displacement sequence $\langle 3, 5, 7, 9, 11\rangle$ and all base types being integers, M can be described by $\mathsf{idx}(1, \langle 3\rangle, \mathsf{vec}(5, 2, \mathsf{leaf}(\texttt{MPI\_INT})))$. A more involved example is shown in Fig. 2. Note that any type map (B, D) of length n can trivially be represented as $\mathsf{struc}(n, D, B)$.

It can easily be seen that each of the MPI derived datatype constructors (for contiguous, vector, index, indexed bucket and structured data layouts) [1, Chapter 4] is expressible by the constructors of Definition 1, and that the mapping is almost one-to-one. For instance, the `MPI_Type_vector` constructor denotes a layout consisting of a strided sequence of blocks, each being a strided sequence of some base type b. This is expressed as $\mathsf{vec}(c, s, \mathsf{vec}(a, e, b))$ where c is the number of blocks, s their stride, a the number of elements in each block, and e the stride used within each block. The idx constructor corresponds to `MPI_Type_create_indexed_block` and makes it possible to express the repetition of the same layout at some given displacements; for this only the sequence of start indices



(a) Trivial representation (corresponds to `INDEXED_BLOCK_TYPE`) lists all base type/displacement pairs and requires $O(n)$ memory.



(b) Optimal representation (corresponds to `STRUCT_VEC_TYPE`) exploits regular, strided structure of column data. Requires constant memory.

Figure 2: The data layout assumed in Section II (elements of the first row and first column of an $n \times n$ matrix of integer values) is formally described by a type map of length $2n-1$ with all base types being int and displacements $D = \langle 0, 4, 8, \ldots, 4(n-1), 4n, 8n, \ldots, 4n(n-1)\rangle$ (assuming 4 Byte integers). Two possible type tree representations are shown.

and the length of this sequence need to be stored. The `MPI_Type_indexed` constructor, which corresponds to our idxbuc constructor, stores also a repetition count for each index. It allows to represent data layouts of irregularly strided blocks more concisely. The most expressive, arbitrary *branching constructor* struc can express the catenation of a sequence of possibly different, smaller layouts each starting at an arbitrary displacement. This is the only constructor with more than one sub-type. In contrast to the similar MPI constructor `MPI_Type_create_struct`, which has also a repetition count for each sub-type, our struc constructor saves this extra sequence. If a sub-type is indeed a repetition of some even smaller sub-type, this information is part of the sub-type and not of the struc node itself. The constructors increase in generality and storage cost: an idx node is a struc node where all sub-types are equal, and therefore does not need to store a sequence of sub-types; a vec node is an idx node with regularly strided displacements, which can be computed from a single scalar instead of an explicit index sequence.

A type tree represents a type map that can be obtained by an ordered traversal of the type tree. This process is called *flattening* and is captured by the algorithm in Listing 1, which suffices for our purposes here (there are better ways of doing flattening, e.g., [13]). The converse is not true: any type map has infinitely many possible type tree representations. Note that the length of the type map

Listing 1: Flattening procedure defining the type map represented by a given type tree T.

```
1  Function Flatten(T, base)
2      switch T.kind do
3          case leaf        /* leaf, base type */
4              output T.b, base;
5          case vec                    /* vector */
6              for i ← 0; i < T.c; i++ do
7                  Flatten (T.S, base + i · T.s);
8          case idx                    /* index */
9              for i ← 0; i < T.c; i++ do
10                 Flatten (T.S, base + T.ix[i]);
11         case idxbuc       /* indexed bucket */
12             for i ← 0; i < T.c; i++ do
13                 ix ← base + T.ix[i];
14                 for j ← 0; j < T.a[i]; j++ do
15                     Flatten (T.S, ix + j · T.s);
16         case struc /* heterogeneous index */
17             for i ← 0; i < T.c; i++ do
18                 Flatten (T.S[i], base + T.ix[i]);
```

described by a type tree T can be arbitrarily larger than the number of nodes in T.

By the *conciseness* of a type tree we mean the space taken by the representation. This is constant for vec and leaf nodes and proportional to the length of the index, bucket size and type sequences for the other constructors. Processing costs are related to conciseness: The concise vec constructor that describes a strided repetition of a sub-type can often be handled by strided memory-copy or communication operations. Constructors with sequences of displacements, bucket sizes or sub-types need at least a traversal of the corresponding sequences and typically entail more irregular and expensive memory accesses. We therefore focus on a simple cost model for optimizing conciseness. The *cost* of a type node shall be proportional to the number of words that must be stored to process the node. This includes the node type (leaf, vec, idx, idxbuc, struc), count, stride and (pointer to) sub-type as well as a per-element cost for index, bucket size and (pointer to) sub-type arrays.

$$\begin{aligned}
\text{cost}(\mathsf{leaf}(b)) &= K_{\mathsf{leaf}} \\
\text{cost}(\mathsf{vec}(c,s,L)) &= K_{\mathsf{vec}} \\
\text{cost}(\mathsf{idx}(c,\ldots)) &= K_{\mathsf{idx}} + cK_{\text{ix}} \\
\text{cost}(\mathsf{idxbuc}(c,\ldots)) &= K_{\mathsf{idxbuc}} + c(K_{\text{ix}} + K_{\text{bucket}}) \\
\text{cost}(\mathsf{struc}(c,\ldots)) &= K_{\mathsf{struc}} + c(K_{\text{ix}} + K_{\text{type}})
\end{aligned}$$

The constants can be adjusted to reflect other overheads

Listing 2: A possible structure for representing idxbuc nodes in type trees or DAGs.

```
1  enum kind = {leaf, vec, idx, idxbuc, struc};
2  struct {
3      kind kind ← idxbuc;
4      int c;                          /* count */
5      int s;                  /* bucket stride */
6      int ix[];               /* displacements */
7      int a[];                 /* bucket sizes */
8      Typenode S;                   /* sub-type */
9  } Idxbuc;
```

related to representing and processing a node. We define the *cost* of a type tree T as the *additive cost* of its nodes T_i: $\text{cost}(T) = \sum_i \text{cost}(T_i)$. Listing 2 gives a C-style structure representing an idxbuc node. With this representation, the constant cost of an idxbuc node is $K_{\mathsf{idxbuc}} = 4$ (the arrays of displacements and bucket sizes are covered by the per-element costs K_{ix} and K_{bucket}). The other constructors are represented analogously.

We can now formally define the problem that we will solve in the next section. We say that a type tree T *represents* a type map M if $\texttt{Flatten}(T, 0) = M$.

MPI TYPE TREE RECONSTRUCTION PROBLEM
Instance: A type map M of length n.
Task: Find a least-cost (or optimal) type tree T representing M; that is, $\text{cost}(T) \leq \text{cost}(T')$ for any tree T' representing M.

A type map may have several least-cost type tree representations and thus an instance of the problem does not necessarily have a unique solution.

IV. THE ALGORITHM

We now present our polynomial-time algorithm for the TYPE TREE RECONSTRUCTION PROBLEM.

Theorem 1. *For any input type map M of length n, the MPI* TYPE TREE RECONSTRUCTION PROBLEM *can be solved in $O(n^4)$ time and $O(n^2)$ space.*

PROOF OUTLINE: We first give a characterization of the structure of optimal type trees (Lemma 1), which allows for a simple and elegant procedure to solve the special case of *aligned* type maps (Definition 4). The fundamental observation for the proof is that any type map of length greater than one can be described by either a catenation of the same kind of shorter type maps (and thus by either a vec, an idx or an idxbuc constructor) or by a catenation of different, but shorter type maps (and thus by the struc constructor). In both cases, for an optimal description, the description of the shorter type maps must likewise be optimal, and the optimal sub-structure property (or dynamic programming

principle) applies. This intuition is formalized in Lemma 2 and Lemma 3. Lemma 4 proves the claim for the special case of aligned type maps, with a detailed procedure given in Listing 4. Finally, Lemma 5 shows how to construct an optimal type tree for any type map out of an optimal tree representation of the aligned type map.

A *segment* of an n-element displacement sequence from index i to index j is denoted by $D[i,j] = \langle d_i, d_{i+1}, \ldots, d_j \rangle$, $0 \leq i \leq j < n$. A *prefix* of length q is the segment $D[0, q-1]$. Segments and prefixes are analogously defined for type signatures and a segment of a type map is defined as $M[i,j] = (B[i,j], D[i,j])$. Intuitively, a prefix is *repeated* in a type map M if M is equal to the catenation of the prefix at certain indices. Repeated prefixes allow to represent M with the idx, idxbuc or vec constructors.

Definition 2 (Repeated prefix, strided prefix)**.** A *repeated prefix* in a displacement sequence D of length n is a prefix $D[0, q-1]$ of length q s.t. q is a divisor of n and for all i, j, $1 \leq i < n/q$, $0 \leq j < q$ we have that $D[j] - D[0] = D[iq + j] - D[iq]$. A *strided prefix* of length q additionally fulfills $D[(i+1)q] - D[iq] = D[q] - D[0]$ for all i, $0 \leq i < n/q - 1$, where $s = D[q] - D[0]$ is the *stride* of the repeated prefix. A prefix $B[0, q-1]$ of a type signature B of length n is repeated if q divides n and $B[i] = B[i \bmod q]$, for all i, $0 \leq i < n$. A prefix $M[0, q-1]$ of a type map $M = (B, D)$ is repeated if the prefix $B[0, q-1]$ is repeated in B and $D[0, 1-q]$ is repeated in D; if $D[0, q-1]$ is additionally a strided prefix, $M[0, q-1]$ is a strided prefix of M.

For a given type map M of length n and a given divisor q of n, it can trivially be checked in $O(n)$ time whether the prefix of length q is repeated and strided in M. Although more efficient algorithms to find all repeated and strided prefixes exist [12], this approach suffices for the purposes of this paper.

We refer to the group of constructors that take an array of indices, i.e., the idx, idxbuc and struc constructors, as *irregular constructors* (or nodes). We call an irregular node with first index $i_0 \neq 0$ a *shifting node*; $x = i_0$ is called the node's *shift*. Adding a value x to all indices of an irregular node T shifts the type map represented by the tree rooted at T by x. As mentioned above, any type map M can be described by either a catenation of the same, shorter type map or by a catenation of different but shorter type maps. Additionally, a representation via a leaf node is possible if M is a trivial type map of length 1 with base type b_0 and displacement 0. In terms of type trees, this means that an optimal tree T for a type map $M = (B, D)$ is either

1) $T = \mathsf{leaf}(b_0)$, a leaf node with base type b_0 if $M = (\langle b \rangle, \langle 0 \rangle)$; or
2) $T = \mathsf{vec}(c, s, S)$, where the prefix $M[0, q-1]$ of length $q = n/c$ is strided in M with stride s and S is an optimal type tree for $M[0, q-1]$; or
3) $T = \mathsf{idx}(c, \langle i_0, \ldots, i_{c-1} \rangle, S)$, where the prefix $M[0, q-1]$ of length $q = n/c$ is repeated in M, S is an optimal tree for $M[0, q-1]$ shifted by i_0 and the indices $i_0, \ldots, i_{c-1}$ are s.t. $\texttt{Flatten}(T, 0) = M$; or
4) $T = \mathsf{idxbuc}(c, s, \langle a_0, \ldots, a_{c-1} \rangle, \langle i_0, \ldots, i_{c-1} \rangle, S)$, where $M[0, q-1]$ is repeated $\sum_{i=0}^{c-1} a_i$ times in M, S is an optimal tree for $M[0, q-1]$ shifted by i_0 and the indices $i_0, \ldots, i_{c-1}$ together with the bucket sizes $a_0, \ldots, a_{c-1}$ are such that $\texttt{Flatten}(T, 0) = M$; or
5) $T = \mathsf{struc}(c, \langle i_0, \ldots, i_{c-1} \rangle, \langle S_0, \ldots, S_{c-1} \rangle)$, where the S_j for $0 \leq j < c$ are optimal trees for some type maps L_j which together with the indices $i_0, \ldots, i_{c-1}$ are such that $\texttt{Flatten}(T, 0) = M$.

Definition 3 (Nice type tree)**.** A *nice type tree* contains at most one shifting node, which is the first irregular node on every root to leaf path.

Lemma 1. *For any tree T representing a type map M, a nice tree representation $\tilde{T}$ of M of equal cost exists.*

Proof: A node is *bad* if it is a shifting node and it is not the first irregular node on every root to leaf path. Let M be a fixed type map and let T be a tree representing M with a minimum number of bad nodes. We will show that T is, in fact, nice.

Assume that a bad idx node (the proof is analogous for bad idxbuc and struc nodes) $N_I = \mathsf{idx}(c, \langle i_0, \ldots, i_{c-1} \rangle, \ldots)$ is present in the k-th subtree of a struct node $N_S = \mathsf{struc}(c', \langle i'_0, \ldots, i'_k, \ldots, i'_{c'-1} \rangle, \langle \ldots \rangle)$ s.t. there is no other shifting node on the path from N_I to N_S. We can change N_I to a non-shifting idx node by subtracting its shift $x = i_0$ from all indices i_j, for $0 \leq j < c$ and adding x to the k-th index i'_k of N_S, i.e., $\tilde{N}_I = \mathsf{idx}(c, \langle 0, i_1 - x, \ldots, i_{c-1} - x \rangle, \ldots)$ and $\tilde{N}_S = \mathsf{struc}(c', \langle i'_0, \ldots, i'_k + x, \ldots, i'_{c'-1} \rangle, \langle \ldots \rangle)$. The resulting tree represents the same type map M but contains one fewer bad node, and hence the existence of such a node N_I would contradict our choice of T. Hence there is no struc node on the path from a bad node N_I to the root node. If this path contains an index node $N'_I \neq N_I$, proceed analogously to the previous case: $\tilde{N}_I = \mathsf{idx}(c, \langle 0, i_1 - x, \ldots, i_{c-1} - x \rangle, \ldots)$ and $\tilde{N}'_I = \mathsf{idx}(c', \langle i'_0 + x, \ldots, i'_{c'-1} + x \rangle, \ldots)$. Again, the obtained tree represents M but contains one fewer bad node, contradicting our original choice of T. The proof is analogous for idxbuc nodes. Consequently, T does not contain any bad nodes and thus must be a nice tree. ∎

Corollary 1. *Any optimal tree T contains at most one irregular node with count 1, i.e., at most one* idx *or* struc *node with $c = 1$ or an* idxbuc *node with $c = 1$ and $a_0 = 1$. Additionally, there is no other irregular node on the path from this node to the root node.*

Proof: Assume that T contains two index nodes with count 1. In a cost-equivalent nice tree representation $\tilde{T}$, at least one of the corresponding index nodes is $\tilde{N} = \mathsf{idx}(1, \langle 0 \rangle, S)$ with some sub-type S, since the two idx nodes

are either on one root to leaf path in T or in two disjoint subtrees (and thus descendants of a struc node). The type tree rooted at $\tilde{N}$ represents exactly the same type map as its subtype S. Thus a representation of smaller cost exists, which contradicts the assumption that T is optimal. The proof is analogous for any pair of two irregular nodes with count 1 and, for idxbuc nodes, bucket size $a_0 = 1$. ∎

Definition 4. A type map or displacement sequence is *aligned* if $D[0] = 0$.

Any displacement sequence can be aligned by setting $D[i] = D[i] - D[0]$ for all i, $0 \le i < n$.

Corollary 2. *An optimal tree for an aligned type map does not contain any shifting nodes or nodes with count 1.*

Proof: It follows directly from Lemma 1 and Corollary 1 that there exists an optimal tree T for the type map M which does not contain any shifting nodes. Note that a vec or non-shifting idx, idxbuc or struc node with count 1 represents the same type map as the node's sub-type. Removing such nodes from a tree reduces the cost while not changing the represented type map and thus no such node can be part of an optimal tree. ∎

Observe that since there are no shifting nodes in an optimal tree T for an aligned type map, any subtree of T represents an aligned segment of M. In the following, we will use $T_{i,j}$ to denote an optimal tree representation for the aligned segment $M[i, j]$ of M, which is also called a *sub-problem*. For convenience, we define the function $\texttt{Min}(S, T)$ which for two trees S and T returns the one with least cost (if either is `null`, the other is returned). The cost of a tree can trivially be computed by a simple traversal. In our bottom-up construction, we keep for each node the cost of the subtree rooted at that node, such that the cost of a tree can be queried in constant time.

Lemma 2. *Let M be any aligned type map of length n and assume that optimal tree representations $T_{0,j}$ are known for all sub-problems of length less than or equal to $\lfloor n/2 \rfloor$, i.e., for all aligned segments $M[0, j]$ with $j + 1 \le \lfloor n/2 \rfloor$. A representation T_r, where the root node of T_r is either an* idx*, an* idxbuc *or a* vec *node and T_r is of least cost w.r.t. all possible representations of that form, can be computed in $O(n\sqrt{n} \log n)$ time, if such a representation exists.*

Proof: Listing 3 enumerates all possible representations of the desired form and chooses the one with least cost among them. Representations via idx and vec nodes are easy to construct: Given a type map M of length n and a divisor q of n, M can be presented as $\mathsf{idx}(n/q, \langle D[0], D[q], \ldots, \rangle, T_{0,q-1})$ if the prefix $M[0, q-1]$ is repeated in M. If the prefix is additionally strided, M can alternatively be represented as $\mathsf{vec}(n/q, D[q]-D[0], T_{0,q-1})$. The cost of these two representations is $K_{\mathsf{idx}} + n/q \cdot K_{\mathrm{ix}} + cost(T_{0,q-1})$ and $K_{\mathsf{vec}} + cost(T_{0,q-1})$ respectively.

Listing 3: Least-cost type tree for an aligned type map with an idx, idxbuc or vec node as root node.

```
1  Function Repetition(B, D, n)
2      T_r ← null;
3      foreach divisor q of n, q < n do
4          c ← n/q;
5          if prefix of length q repeated in (B,D) then
6              /* Representation via idx      */
7              for i ← 0; i < c; i++ do ix[i] ← D[iq] ;
8              T_idx ← idx(c, ix, T_{0,q-1});
9              T_r ← Min (T_idx, T_r);
10             /* Representation via idxbuc   */
11             E ← {D[iq] − D[(i − 1)q] | 1 ≤ i < n/q};
12             sort E;
13             s ← most frequently occurring item in E;
14             ix[0] ← D[0]; a[0] ← 1; j ← 0;
15             for i ← 0; i < n/q; i++ do
16                 if D[iq] − D[(i−1)q] = s then a[j]++
                   else j++; ix[iq] ← D[iq]; a[j] ← 1;
17             T_idxbuc ← idxbuc(j + 1, s, a, ix, T_{0,q-1});
18             T_r ← Min (T_idxbuc, T_r);
19             /* Representation via vec      */
20             if prefix of length q strided in (B, D) then
21                 s ← D[q] − D[0];
22                 T_vec ← vec(c, s, T_{0,q-1});
23                 T_r ← Min (T_vec, T_r);
24     return T_r;
```

Representations with an idxbuc node as the root node also require a repeated prefix, but finding the most cost-efficient one is more involved. For each possible bucket stride, the number of buckets that this stride gives rise to has to be counted. The stride s leading to a smallest number of buckets is a candidate for the representation of Mvia an idxbuc node. Observe that each i with $D[i + 1] - D[i] = s$ joins two s-strided segments $D[j, i]$ and $D[i + 1, k]$ into one bucket starting at index j. Therefore, the stride that occurs most often in the stride sequence $S[i] = D[i + 1] - D[i]$, $0 \le i < n - 1$ leads to the smallest number of buckets. To count the number of occurrences of each stride, we either sort S or count by hashing during the scan of D. Let s be the stride with the most occurrences. A final scan of D suffices to compute the start indices and sizes of the buckets with stride s. The cost of this representation is $K_{\mathsf{idxbuc}} + c(K_{\mathrm{bucket}} + K_{\mathrm{ix}}) + cost(T_{0,q-1})$. Due to Corollary 2, nodes with count 1 cannot be part of a least-cost representation of the desired form and thus need not be considered. The number of divisors of n is upper-bounded by $2\lfloor\sqrt{n}\rfloor$ and, by assumption, optimal representations $T_{0,j}$ for all sub-problems $M[0, j]$ with $j \le \lfloor n/2 \rfloor$ are known, which implies the claimed runtime bound. ∎

Lemma 3. *Let M be any aligned type map of length n and assume that optimal tree representations $T_{i,j}$ are known for all sub-problems of length strictly less than n, i.e., for all aligned segments $M[i,j]$ with $j-i+1 < n$. A representation T_s, where the root node of T_s is a* struc *node and T_s is of least cost w.r.t. to all possible representations of that form, can be computed in $O(n^2)$ time.*

Proof: Construct a weighted, directed acyclic graph $G = (V, E, w)$ with $V = \{v_0, \ldots, v_n\}$, $E = \{(v_i, v_j) \mid 0 \leq i < j \leq n,\ j - i < n\}$. The weight for each edge (v_i, v_j) in E is defined to be $w(v_i, v_j) = K_{\text{ix}} + K_{\text{type}} + \text{cost}(T_{i,j-1})$. The intended meaning of this construction is as follows. A node v_i corresponds to the i-th base type/displacement pair of M (v_n is a special vertex that corresponds to the hypothetical first element after the end of M). An edge (v_i, v_j) with $i < j$ corresponds to the aligned segment $M[i, j-1]$. The weight of an edge (v_i, v_j) is equal to the cost of the optimal representation $T_{i,j-1}$ of the sub-problem $M[i, j-1]$ (which exists by the assumption) plus a cost of $K_{\text{ix}} + K_{\text{type}}$ for including this representation as a sub-type in a struc node. The edge (v_0, v_n), which is not part of the constructed graph, can be thought of as corresponding to the type tree $T_{0,n-1}$, i.e., the optimal type tree representation of M that we want to compute.

Let $P = \langle v_0, u_1, \ldots, u_k, v_n \rangle$ be a shortest path in G from v_0 to v_n with $u_i \in V$ for $1 \leq i \leq k$. Then the tree $\mathsf{struc}(k+1, \langle D[0], D[u_1], \ldots, D[u_k] \rangle, \langle T_{0,u_1-1}, T_{u_1,u_2-1}, \ldots, T_{u_k,n-1} \rangle)$ represents M. By construction, for any possible representation of M of the desired form, a corresponding path from v_0 to v_n exists and thus a shortest path represents the desired least-cost solution. Given P, this representation can be constructed in linear time, since optimal representations for all required sub-problems are known by the assumption. The resulting graph is a topologically sorted DAG with $\binom{n-1}{2} - 1$ edges, in which the single source shortest path (SSSP) problem can be solved in $O(|V| + |E|)$ [14], i.e., in $O(n^2)$ steps. ∎

We can now give the complete dynamic programming algorithm (Listing 4) for constructing optimal trees for aligned type maps. Due to Lemma 1, it suffices to construct an optimal nice tree, which according to Corollary 2 cannot contain any shifting nodes nor any nodes with count 1.

Lemma 4. *For any aligned type map M of length n, the* TYPE TREE RECONSTRUCTION PROBLEM *can be solved in $O(n^4)$ time and $O(n^2)$ space.*

Proof: The input to the algorithm is an aligned type map $M = (B, D)$ of length n. The algorithm computes an optimal tree $T_{i,j}$ for each aligned segment $M[i,j]$, $0 \leq i \leq j < n$, which is stored with edge (v_i, v_{j+1}) in the constructed graph G. The solution for the whole input type map M can be read off of the edge (v_0, v_n).

The algorithm first constructs an optimal solution for each sub-problem of length 1. An aligned type map of length 1 is of the form $(\langle b_0 \rangle, \langle 0 \rangle)$, and is optimally represented as $\mathsf{leaf}(b_0)$. It then computes optimal tree representations for all aligned segments of M, via a bottom up dynamic programming approach. The dynamic programming table to be filled in is implicit in the graph G, where each sub-problem $M[i,j]$ is associated with an edge (v_i, v_{j+1}). After the preprocessing step, solutions for all sub-problems of length 1 are known. By incrementally computing optimal representations for all sub-problems of length $2, \ldots, n$, it is ensured that Lemmas 2 and 3 can be applied to compute an optimal representation for each sub-problem as follows. First, a sub-problem $M[i, i+l-1]$ of length l is extracted by aligning the corresponding type map segment. A tree T_r, whose root node is either an idx, an idxbuc or a vec node, and a tree T_s, whose root node is a struc node, are computed. Both are of least cost w.r.t. all tree representations

Listing 4: Computing an optimal type tree for an aligned type map.

```
1  Function Typetree(B, D, n)
2      G = ({v_0, ..., v_n}, ∅);
3      /* All sub-problems of length 1   */
4      for i ← 0; i < n; i++ do
5          T_{i,i} ← leaf(B[i]);
6          w_{i,i+1} ← K_ix + K_type + K_leaf;
7          add edge (v_i, v_{i+1}) with T_{i,i} and w_{i,i+1} to G;
8      /* Solutions for all subproblems  */
9      for l ← 2; l ≤ n; l++ do
10         for i ← 0; i ≤ n − l; i++ do
11             j ← i + l − 1;
12             /* Compute optimal tree for
                   aligned segment M[i,j] of
                   length l                */
13             for k ← 0; k < l; k++ do
14                 D'[k] ← D[i + k] − D[i];
15             /* Representation with idx,
                   idxbuc or vec node as root  */
16             T_r ← Repetition (B[i,j], D', l, i);
17             T_{i,j} ← Min (T_r, T_{i,j});
18             /* Representation with struc
                   node as root            */
19             Find shortest path P from v_i to v_{j+1} in G;
20             Assume P = ⟨v_i, u_1, ... u_k, v_j⟩;
21             ix ← ⟨0, D'[u_1], ..., D'[u_k]⟩;
22             S ← ⟨T_{v_i,u_1−1}, T_{u_1,u_2−1}, ..., T_{u_k,v_{j−1}}⟩;
23             T_s ← struc(k + 1, ix, S);
24             T_{i,j} ← Min (T_s, T_{i,j});
25             Add edge (v_i, v_{j+1}) with representation T_{i,j}
               and weight K_ix + K_type + cost(T_{i,j}) to G;
26     return T_{0,n−1};
```

of the desired form. The optimal tree for a sub-problem $M[i, i+l-1]$ is therefore either T_r or T_s.

To compute T_r, a small, technical extension of Listing 3 for finding representations via idx, idxbuc or vec nodes is necessary. The procedure requires access to optimal representations of the prefixes of the argument type map $M = (B, D)$. However, in the general case, the passed type map is a segment of the input type map M and its prefixes start at index i. To account for this, we pass an additional argument o representing the offset of the segment within the input type map M (i.e., for a segment $M[i, j]$, we have $o = i$), and in lines 9, 25 and 30 replace the argument $T_{0,q-1}$ with $T_{o,o+q-1}$.

When computing T_s, in Listing 4 we do not need to construct a new graph for each sub-problem when computing its representation T_s. Instead, a single dynamic, incrementally built graph G suffices for all sub-problems of M. By construction, when computing the desired representation for a sub-problem $M[i, i+l-1]$, G contains edges representing optimal representations for all sub-problems of length less than l (and possibly some edges representing solutions for sub-problems of length l). A shortest path from node v_i to v_{i+l} in G therefore leads to the same representation as the one constructed by Lemma 3. To find such a shortest path, for each sub-problem $M[i, i+l-1]$ of length l, an SSSP on a weighted DAG with $l+1$ nodes and $O(l^2)$ edges has to be solved. To compute the desired representations for all sub-problems of length l, a shortest path must be computed for each of the $n+1-l$ node pairs (v_i, v_{i+l}), for $0 \leq i \leq n+1-l$. The total time is thus upper bounded by $\sum_{l=1}^{n+1} l^2(n+1-l)$, which is $O(n^4)$.

The algorithm constructs a graph with $O(n^2)$ edges, where a tree $T_{i,j}$, representing the solution for the sub-problem $M[i, j]$, is associated with each edge (v_i, v_j). Note that for each edge (v_i, v_j) it suffices to store the root node of the associated tree $T_{i,j}$ plus pointers to its child nodes, which are already stored with the respective edges. To meet the desired space bound, only a constant amount of space may be used by each edge and associated tree. This is trivially true for leaf and vec nodes: Apart from one word indicating the node's kind, only (a pointer to) the base type b needs to be stored. For vec nodes, two integer values and one pointer to the child node are required in addition to the node's kind. However, irregular nodes may require $\Omega(n)$ space in the worst case, e.g., if $\mathsf{struc}(n, D, B)$ is the optimal representation of M. We employ a standard trick often used in dynamic programming algorithms and store for each node only the information required to reconstruct the full solution once the algorithm has terminated. If for an idx node the count c is known, the full idx node is easily derived as $\mathsf{idx}(c, \langle D[0], D[q], \ldots, D[(c-1)q] \rangle, T_{0,q-1})$ with $q = n/c$. For idxbuc nodes, only (a pointer to) the sub-type $T_{i,j}$ is required, since all parameters are derivable by the same approach as used in Listing 3 if the length $q = j-i+1$ of the prefix is known. The parameters of a struc node associated with an edge (v_i, v_j) can be reconstructed by again computing the shortest path from node v_i to v_j as done in Lemma 3. This reconstruction step does not change the asymptotic runtime bound. The space required for each edge is $O(1)$, from which the claimed $O(n^2)$ space upper bound follows directly. ■

In the following, we show how the algorithm in Listing 4 can be applied to general (non-aligned) type maps.

Corollary 3. *For any optimal nice tree with an irregular node N with count 1, i.e., a node $N = \mathsf{idx}(1, \langle i_0 \rangle, \ldots)$, $N = \mathsf{idxbuc}(1, s, \langle 1 \rangle, \langle i_0 \rangle, \ldots)$, or $N = \mathsf{struc}(1, \langle i_0 \rangle, \langle \ldots \rangle)$, a representation T' of equal cost s.t. N is the root node of T' exists.*

Proof: Due to Corollary 1, there is no idx, idxbuc or struc node on the path from N to the root and thus N shifts the type map represented by T by i_0. This shift can be represented equivalently by removing N from the tree and adding a new root node to represent the shift, i.e., by letting $T' = \mathsf{idx}(1, \langle i_0 \rangle, T \setminus N)$ for an index node N. ■

Corollary 4. *The height of an optimal tree is $O(\log_2 n)$.*

Proof: It is easy to see that an optimal tree does not contain two consecutive struc nodes, as they can always be merged into one at smaller cost. For any tree T that represents a type map of length n, a tree $\mathsf{idx}(c, \langle \ldots \rangle, T)$ or $\mathsf{vec}(c, \ldots, T)$ with $c \geq 2$ represents a type map of length at least $2n$. The same holds for a tree of the form $\mathsf{idxbuc}(c, \ldots, \langle a_0 \rangle, \langle \ldots \rangle, T)$ with $c > 1$ or $a_0 > 1$. Let P be a maximum-length path from a leaf to the root of an arbitrary optimal tree. Since any optimal tree contains at most one irregular node with count $c = 1$ (Corollary 1) and no vec nodes with $c = 1$, the length of the represented type map at least doubles with at least every other node on P. ■

Lemma 5. *Given optimal trees $T_{i,j}$ for all sub-problems $M[i, j]$ of a type map M, an optimal tree representing M can be computed in $O(n^2)$ time and $O(n)$ space.*

Proof: By Lemma 1, for any optimal tree T a cost-equivalent nice tree $\tilde{T}$ representing the same type map M exists and it therefore suffices to find an optimal nice tree representation $\tilde{T}$ for M. By assumption, an optimal nice tree representation $T = T_{0,n-1}$ for the aligned type map exists. To construct $\tilde{T}$, find the first node N on any root to leaf path in T that is either an idx, idxbuc or a struc node and add the displacement sequence's shift $x = D[0]$ to the indices of this node, i.e., if $N = \mathsf{idx}(c, \langle i_0, \ldots, i_{c-1} \rangle, S)$ in T, set $\tilde{N} = \mathsf{idx}(c, \langle i_0 + x, \ldots, i_{c-1} + x \rangle, S)$ in $\tilde{T}$ and analogously for the case of N being an idxbuc or a struc node. Since the height of an optimal tree is $O(\log n)$, and the array of indices of a node is at most of length n, this step is feasible in linear time. Note that $\tilde{T}$ has the same cost as T and thus is an optimal tree representation for M.

If such a node does not exist, it follows from Lemma 1 and Corollary 3 that the optimal solution is either

- $\tilde{T} = \mathsf{idx}(n/q, \langle \ldots \rangle, T_{0,q-1})$ or $\tilde{T} = \mathsf{idxbuc}(c, s, \langle \ldots \rangle, \langle \ldots \rangle, T_{0,q-1})$ for some divisor q of n, if the prefix of length q is repeated in M, or
- $\tilde{T} = \mathsf{struc}(c, \langle \ldots \rangle, \langle T_0, \ldots, T_{c-1} \rangle)$ for some c, $1 \leq c \leq n$.

Due to the general cost model, the trivial representations with prefixes of length 1 and n need to be checked for all three constructors. Since solutions for all sub-problems are already known, this construction is feasible in $O(n^2)$ time and $O(n)$ space by the same steps as used in Listings 3 and 4. ■

Proof of Theorem 1: The TYPE TREE RECONSTRUCTION PROBLEM for a type map M of length n can be solved by computing an optimal tree representation for the aligned type map (Lemma 4) and the post-processing step given in Lemma 5. The claimed space and time bounds follow. ■

V. TYPE RECONSTRUCTION INTO DAGS

A type tree describing some given type map may have multiple instances of the same sub-tree. In this case, an even more concise representation would result by folding the tree into a directed acyclic graph (DAG) with only one node for each occurrence of some sub-tree.

Type DAGs represent type maps by the same flattening procedure as shown in Listing 1 for trees. Each path from the root node in the type DAG to a leaf is traversed in order to generate the corresponding type map, and the processing of type DAGs would be similar to processing type trees. A type DAG can easily be unfolded into an equivalent type tree. With the same additive cost model (cost of a DAG is the sum of the costs of the nodes in the DAG), the *MPI type DAG reconstruction problem* is to find the least-cost DAG representing the given type map. Our algorithm for constructing optimal type tree representations does not work in this case. One crucial problem is that the best representation for a type map segment in a DAG no longer needs to be locally optimal, since cost savings may be achieved by reusing this node in other parts of the DAG, i.e., the dynamic programming principle does not apply. This is illustrated in Fig. 3. Also the shortest path computation no longer correctly accounts for the costs under reuse of nodes. The type tree constructed by unfolding a cost-optimal DAG is not necessarily a cost-optimal tree, and conversely, the DAG obtained by folding a given, cost-optimal type tree is not necessarily a cost-optimal DAG. This is a fundamental problem for our approach to handling type trees. New ideas are needed to solve the type DAG reconstruction problem.

VI. TYPE NORMALIZATION PROBLEMS

The type normalization problem subsumes the type reconstruction problem that we have considered so far. Type normalization asks to improve the cost of an already given

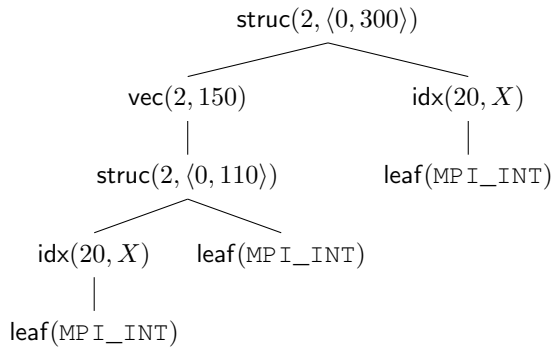


Figure 3: An unfolding of an optimal type DAG representing type map M; X is an arbitrary displacement sequence of length 20 over $0, 1, \ldots, 99$. In the type DAG, the subtrees rooted at $\mathsf{idx}(20, X)$ only contribute to the cost function once. Notice that the subtree rooted at $\mathsf{struc}(2, \langle 0, 300 \rangle)$ is not a least-cost type tree representation of the represented type map segment, as it can be represented with less cost as $\mathsf{idx}(21, \langle X, 110 \rangle, \mathsf{leaf}(\texttt{MPI\_INT}))$.

derived datatype tree. Since any data layout can be represented as a single struc node with the whole displacement sequence as index sequence and the sequence of base types as sub-types, type reconstruction is a special case of type normalization. Normalization is the problem that compiler or MPI library implementers are typically faced with: application data structures described as trees are given by the programmer, and an internal, optimal representation is to be constructed by the programming system. The trivial solution is to flatten the given type tree and perform type reconstruction on the resulting type map. Since the length of the resulting type map is not bounded by the total length of the index sequences of the tree, this is highly undesirable. We would like a procedure where the complexity can be bounded by the size of the type tree.

As shown in [6], if the set of constructors excludes the struc constructor, it is possible to perform type normalization by only rechecking optimality of the idx nodes. When the struc constructor is included, arbitrarily more concise representations are possible as shown in Fig. 2. Optimality of a sub-tree that does not use the struc constructor therefore does not imply optimality when the struc constructor is allowed. It is therefore necessary to perform type reconstruction on the flattened input type tree.

VII. CONCLUSION

The main result of this paper is that the MPI TYPE TREE RECONSTRUCTION PROBLEM is actually solvable in polynomial time. However, an $O(n^4)$ algorithm is not useful for larger values of n, as might be the case in some applications where n could be proportional to the number of MPI processes. Our bottom-up dynamic programming algorithm performs a considerable amount of redundant

checking for (strided) repetitions in type map segments. An asymptotically more efficient algorithm is likely to exist. Whether an exact, practically efficient algorithm for the full problem is possible, we do not know at the point of writing.

Restricting the power of the constructors can permit more efficient algorithms. As shown in [12], if only leaf, vec and idx nodes are allowed, then the type reconstruction problem for type maps of length n can be solved in $O(n \log n / \log \log n)$ time steps. However, the resulting restricted trees can and often will be more costly, as shown in Fig. 2. The high complexity of our algorithm is caused by the unbounded branching constructor node struc. An alternative would be to look for low-complexity approximation algorithms with provable approximation guarantees. Or, even weaker, for heuristics that perhaps work well for the intended application cases. This reflects the state in current MPI libraries.

As discussed, type trees can be represented more concisely as directed acyclic graphs (DAGs). To the best of our knowledge, it is still open whether a cost-optimal DAG representation for an arbitrary type map can likewise be constructed in polynomial time.

A related problem is the following. Given two type maps of the same length, construct a least-cost tree (or DAG) representing a mapping between the two type maps. Such a tree (DAG) has uses when copying between different data layouts; this arises, e.g., in matrix transposition. In the MPI context this operation has been called *transpacking* [15], [16]. Our dynamic programming algorithm may extend to this case as well.

We believe that the derived datatype idea is applicable beyond MPI in a much wider context of (parallel) programming interfaces and languages (e.g., for deep copying of statically defined structures), and that the type normalization and reconstruction problems as defined here have relevance beyond the MPI context.

REFERENCES

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

[2] T. Hoefler and S. Gottlieb, "Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes," in *Recent Advances in Message Passing Interface. 17th European MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, vol. 6305. Springer, 2010, pp. 132–141.

[3] J. L. Träff, A. Rougier, and S. Hunold, "Implementing a classic: Zero-copy all-to-all communication with MPI datatypes," in *28th ACM International Conference on Supercomputing (ICS)*, 2014, pp. 135–144.

[4] T. Prabhu and W. Gropp, "DAME: A runtime-compiled engine for derived datatypes," in *22nd European MPI Users' Group Meeting (EuroMPI)*, 2015.

[5] 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)*, 2013, pp. 19–24.

[6] J. L. Träff, "Optimal MPI datatype normalization for vector and index-block types," in *21st European MPI Users' Group Meeting (EuroMPI/ASIA)*, 2014, pp. 33–38.

[7] T. Schneider, R. Gerstenberger, and T. Hoefler, "Application-oriented ping-pong benchmarking: how to assess the real communication overheads," *Computing*, vol. 96, no. 4, pp. 279–292, 2014.

[8] 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*, ser. Lecture Notes in Computer Science, vol. 6960. Springer, 2011, pp. 150–159.

[9] F. Kjolstad, T. Hoefler, and M. Snir, "A transformation to convert packing code to compact datatypes for efficient zero-copy data transfer," University of Illinois at Urbana-Champain, Tech. Rep., 2011, retrieved from https://www.ideals.illinois.edu/handle/2142/26452, last visited on 06/29/2015.

[10] ——, "Automatic datatype generation and optimization," in *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012, pp. 327–328.

[11] 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*, ser. Lecture Notes in Computer Science, vol. 2840. Springer, 2003, pp. 404–413.

[12] M. Kalany and J. L. Träff, "Efficient, optimal MPI datatype reconstruction for vector and index types," in *22nd European MPI Users' Group Meeting (EuroMPI)*, 2015.

[13] J. L. Träff, R. Hempel, H. Ritzdorf, and F. Zimmermann, "Flattening on the fly: efficient handling of MPI derived datatypes," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, vol. 1697. Springer, 1999, pp. 109–116.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[15] F. G. Mir and J. L. Träff, "Constructing MPI input-output datatypes for efficient transpacking," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 15th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, vol. 5205. Springer, 2008, pp. 141–150.

[16] R. B. Ross, R. Latham, W. Gropp, E. L. Lusk, and R. Thakur, "Processing MPI datatypes outside MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 16th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, vol. 5759. Springer, 2009, pp. 42–53.