Optimal MPI Datatype Normalization for Vector and Index-block Types*

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

The derived datatypes of MPI is an extremely powerful mechanism for specifying the layout of data in communication operations. It is desirable that MPI libraries internally simplify complex datatype descriptions into representations that are efficient for the communication operations in which they are used. This process is called *datatype normalization*, and MPI libraries typically employ simple heuristics for this task. In this paper we embark on a study of the inherent complexity of datatype normalization, and show that the problem with a specific, but flexible cost model can be solved cost-optimally for MPI vector and index-block types in polynomial time. For a type map consisting of n displacement-basetype pairs, we first give a type reconstruction algorithm running in $O(n\sqrt{n})$ time steps. We then use this algorithm for datatype normalization of given derived datatypes.

1. INTRODUCTION

The derived datatypes of MPI is an extremely powerful mechanism for specifying the layout of data to be used in MPI communication operations [10, Chapter 4]. Derived datatypes delegate the responsibility of accessing the data at the described memory locations in the described order to the MPI library implementation, that can (and often does) employ sophisticated, tailored packing and pipelining techniques. Hereby, the user is freed from implementing and maintaining packing and unpacking routines for accessing non-contiguous data before and after communication operations working on contiguous buffers. Datatypes are completely orthogonal to the MPI communication models, and can be used without restrictions in both point-to-point, one-sided and collective communication. Many promising uses of derived datatypes in applications with descriptive and/or performance advantage have been reported, e.g., [1, 7, 9, 17].

Although application data structures are often in principle relatively simple, nested, strided layouts that could be described using the MPI vector constructor (`MPI_Type_vector`), there are often corner cases that make the use of the more complex, irregular constructors for indexed and structured types (`MPI_Type_indexed`, `MPI_Type_create_struct`) necessary. This complexity may detract users from applying derived datatypes, also because it may not be obvious at all what the "most efficient" description of some given application data layout may be.

On the other hand, it is one of the key design decisions of MPI not to define specific performance requirements for the features of the standard, but to leave it to the concrete implementation to do the best possible for the given system ("high quality implementation"). In this vein, use of derived datatypes should let the application be the guide to the most natural way of specifying the relevant data layouts and trust the MPI library to do the best possible with any given derived datatype and communication operation combination. The natural description could for instance make generous use of indexed and structured type constructors, even when it might be clear that the layouts may have a (partially) regular structure.

With such an advice to users, it becomes desirable that MPI libraries internally simplify complex datatype descriptions into representations that are efficient for the communication operations in which they are used. This process is sometimes called *datatype normalization* [5], and MPI libraries typically employ simple heuristics to accomplish this task.

For MPI implementers there are a number of issues to consider in order to support a derived datatype heavy programming style. The most important issue is to design efficient datatype engines for packing, unpacking and pipelining non-contiguous data described by derived datatypes. There has been considerable amount of work in this direction, see e.g., [2, 3, 6, 11, 12, 13, 14, 16, 18], and we will not consider it here. The issue we consider is the normalization operation, namely how to optimally simplify any given application derived datatype so as to best fit the lower-level datatype type engine functionality. There has been little systematic work in the MPI community in this direction.

In this paper we investigate aspects of the datatype normalization problem. We use a simple cost model, discussed

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

EuroMPI/ASIA '14, September 09-12 2014, Kyoto, Japan

http://dx.doi.org/10.1145/2642769.2642771.

previously in [5], which we think is (closely) related to the actual costs incurred (by any datatype engine) when processing a derived datatype in combination with a communication operation. As a first step towards characterizing the complexity of finding optimal, least-cost datatype descriptions, we show that when the allowed constructors are restricted to only contiguous, vector and index-block constructors (`MPI_Type_contiguous`, `MPI_Type_vector`, `MPI_Type_create_indexed_block`, and their Byte counterparts) optimal descriptions can indeed be computed in polynomial time under the given cost model. Perhaps surprising, the optimal algorithm does *not* correspond to the naive, greedy, type promotion heuristic (struct to indexed to indexed block to vector to contiguous) often used in MPI libraries (see also [8]), and in fact often does the opposite as what this idea would suggest. More precisely, for type maps of n displacement-basetype pairs, the algorithm runs in $O(n\sqrt{n})$ time steps (but it is likely that this may be improved).

For the general problem of constructing least-cost types when all MPI type constructors are allowed, we (as in [5]) conjecture that the problem is NP-hard. A proof is outside the scope of this paper. Should this be the case, this result will set (theoretical) limits to the unrestricted use of derived datatypes, but may also constructively suggest new datatype constructors that are more convenient and efficient both from a complexity and a usability point of view.

2. THE PROBLEM

A *data layout* is a(n ordered) sequence of displacements and basetypes like chars, integers, floats etc., corresponding to the basic type of either C or FORTRAN. In MPI terms, such a layout is called a *type map* [10, p. 84]. Derived datatypes are a mechanism for more compactly describing type maps. A derived datatype can be thought of as a tree (DAG) in which each node describes how the type maps described by its children are repeated and offset so as to form a longer type map. By representing a type map, derived datatypes can be used in communication operations as directives telling the datatype and communication engine where the individual data elements to be communicated are found. Simple strided layouts where a type map is repeated a certain number of times with a fixed stride can be described by using the `MPI_Type_contiguous` and `MPI_Type_vector` constructors. Irregular layouts where the children layouts are repeated at different, not linearly related displacements can be described with the `MPI_Type_create_indexed_block`, `MPI_Type_indexed` and `MPI_Type_create_struct` constructors.

Processing a derived datatype entails locating each of the basetypes (leaves), which means that the type tree (DAG) has to be traversed in a depth-first order. The processing and storage cost is therefore proportional to the number of type nodes in the tree. For nodes corresponding to the irregular constructors, associated arrays (or lists) are required for the displacements, and these lists both take up space and require time to parse. The processing cost is therefore also related to the size of such arrays. Many short arrays may furthermore be preferable to fewer, longer lists, since precompilation of the required copy operations may be feasible for smaller arrays [13].

In the following, we restrict consideration to the contiguous, vector and index-block constructors. This means that we can describe only homogeneous layouts where all basetypes are the same basic type. We can therefore omit the basetypes from the type map, and concentrate on using derived datatypes to describe more compactly given sequences of displacements.

We will use the following representation for derived datatype nodes (see also [15]):

1. A *leaf node*, $\mathsf{con}(c)$, with *count* c describes a sequence of c adjacent relative indices, $0, 1, 2, \ldots, c-1$
2. A *vector node* $\mathsf{vec}(c, d, C)$ with *count* c and stride d describes the catenation of c sequences C at relative indices $0, d, 2d, \ldots, (c-1)d$
3. an *index node*, $\mathsf{idx}(c, \langle i_0, i_1, \ldots, i_{c-1}\rangle, C)$, with *count* c and *indices* $\langle i_0, i_1, \ldots, i_{c-1}\rangle$ describes the catenation of c sequences C at relative indices $i_0, i_1, \ldots, i_{c-1}$

For instance, the data layout $[2, 4, 6, 8, 9, 11, 13, 15, 17, 1, 3, 5, 7]$ can be described by $\mathsf{idx}(3, \langle 2, 9, 1\rangle, \mathsf{vec}(4, 2, \mathsf{con}(1)))$, and the data layout $[5, 6, 8, 9, 10, 12, 13, 14, 16]$ by

$$\mathsf{idx}(1, \langle 5\rangle, \mathsf{vec}(3, 4, \mathsf{idx}(3, \langle 0, 1, 3\rangle, \mathsf{con}(1))))$$

As can be seen, the type maps that can be constructed by application of the MPI constructors `MPI_Type_contiguous`, `MPI_Type_vector` and `MPI_Type_create_indexed_block` can all be represented by combinations of these three node types. For instance, an MPI vector over type C of extent e with count c, stride s, and blocklength b is represented by

$$\mathsf{vec}(c, se, \mathsf{vec}(b, e, C))$$

We first consider the storage costs entailed by a derived datatype, and define the *cost of a type node* to be proportional to the number of words that must be stored to process the node. This includes the node type ($\mathsf{con}, \mathsf{vec}, \mathsf{idx}$), count, displacement or pointer to index array, pointer to child node plus for idx-nodes the size of the displacement sequence:

$$\begin{aligned} \mathrm{cost}(\mathsf{con}(c)) &= K_{\mathsf{con}} \\ \mathrm{cost}(\mathsf{vec}(c, d, C)) &= K_{\mathsf{vec}} \\ \mathrm{cost}(\mathsf{idx}(c, \langle \ldots\rangle), C) &= K_{\mathsf{idx}} + c \end{aligned}$$

The constants can be adjusted to reflect other overheads related to representing and processing a node. We define the *cost of a type tree* T to be the *additive cost* of its nodes T_i: $\mathrm{cost}(T) = \sum_i \mathrm{cost}(T_i)$.

We first consider the *type reconstruction problem*: given a displacement sequence x of size n, find a least-cost type tree T built from $\mathsf{con}, \mathsf{vec}$ and idx nodes representing x. Note that with the restriction to only these types of nodes the trees are actually paths, since vec and idx nodes have only a single child.

3. TYPE RECONSTRUCTION

We first solve the type reconstruction problem by proving:

THEOREM 1. *For any input displacement sequence x of length n, a least-cost type tree representing x using only $\mathsf{con}, \mathsf{vec}$ and idx nodes can be constructed in $O(n\sqrt{n})$ steps.*

For the algorithm we need the following structural lemma. Let x be a given displacement sequence of length n, and let y be a prefix of x of length c. We say that y is *repeated* in x if c divides n and $y[i \bmod c] - y[0] = x[i] - x[c\lfloor i/c\rfloor]$ for

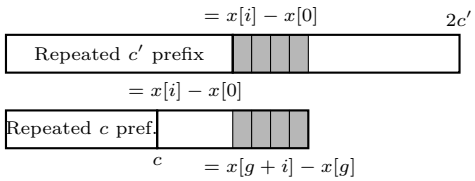

Figure 1: For repeated prefixes of length c and c' with $c' > c$ and c' not a multiple of c, a shorter repeated prefix of length $g = \gcd(c', c)$ can be constructed.

all $i, 0 \leq i < n$. A sequence x is said to be *regularly strided*, if the difference between any two successive indices is some constant d, i.e. $x[i+1] - x[i] = d$ for each $i, 0 \leq i < n-1$.

LEMMA 1. *Let x be a displacement sequence that is not regularly strided, and let y be the shortest prefix of length $c, c \geq 2$ that is repeated in x. Then for any longer prefix y' of length $c', c' > c$ that is also repeated in x, it holds that c' is a multiple of c.*

PROOF. If c' is a multiple of c, $c' = kc$ for some $k, k > 1$, we have that $x[i] - x[c'\lfloor i/c' \rfloor] = (x[i] - x[c\lfloor i/c \rfloor]) + (x[c\lfloor i/c \rfloor] - x[c'\lfloor i/c' \rfloor]) = (x[i \bmod c] - x[0]) + (x[c\lfloor i/c \rfloor] - x[c'\lfloor i/c' \rfloor])$ since the shorter prefix y is repeated in x. For the longer prefix to be repeated, x must fulfill that $x[c'\lfloor i/c' \rfloor + jc] - x[c'\lfloor i/c' \rfloor]) = x[jc] - x[0]$ for $j = 1, \ldots k-1$.

Now assume that c' is not a multiple of c, and let $g = \gcd(c, c')$. The (shorter) prefix of length g is repeated in x, contradicting if $g > 1$ that y was the shortest repeated prefix. This can be seen as follows. Since both prefixes are repeated in x we have for any $j, 0 \leq j < n/c'$ and $k, 0 \leq k < n/c$ that $x[jc'+i] - x[jc'] = x[i] - x[0]$ and $x[kc+i] - x[kc] = x[i] - x[0]$ and thus $x[jc'+i] - x[jc'] = x[kc+i] - x[kc]$ for $i = 1, \ldots c-1$. By choosing pairs of k and j such that $jc' - kc = g$, $jc' - kc = 2g$, $\ldots$, $jc' - kc = c - g$ (possible since g is the greatest common divisor of c' and c) we infer that the prefix of length g is repeated in y and therefore also repeated in x, as illustrated in Figure 1. Also, it follows that the difference between $x[ig-1]$ and $x[ig]$ is the same constant. Thus, if $g = 1$ the difference between any two successive indices in x is the same constant $x[1] - x[0]$, contradicting that x was not regularly strided. □

A repeated prefix in a displacement sequence x allows to represent x by a displacement sequence of length n/c of sequences of length c corresponding to the repeated prefix. The prefix can be represented either by a vector node, if the repeated prefix y is itself regularly strided, or an index node. Lemma 1 states that in a not regularly strided displacement sequence it suffices to find the shortest repeated prefix, since all other repeated prefixes will be multiples of c, and can thus be discovered by analyzing recursively the n/c sequence consisting of every cth index of x. This gives an algorithm for finding a path of type nodes representing any given displacement sequence. Such a path will not necessarily be of least (additive) cost, but it *almost* has the property that any other path representing the given displacement sequence can be constructed from it by combining subpaths of consecutive type nodes. Two successive index nodes, $T_k = \mathsf{idx}(c_k, \langle i_{k,0}, i_{k,1}, \ldots, i_{k,c_k-1} \rangle, T_{k+1})$ and $T_{k+1} = \mathsf{idx}(c_{k+1}, \langle i_{k+1,0}, i_{k+1,1}, \ldots, i_{k+1,c_{k+1}-1} \rangle, T_{k+2})$, can be combined into a single index node, $T'_k = \mathsf{idx}(c_k c_{k+1}, \langle i_{k,0} + i_{k+1,0}, i_{k,0} + i_{k+1,1}, \ldots, i_{k,0} + i_{k+1,c_{k+1}-1}, i_{k,1} + \ldots, \ldots, i_{k,c_0-1} + i_{k+1,c_{k+1}-1} \rangle, T_{k+2})$ by repeating the displacement sequence of T_{k+1} c_k times in the displacement sequence of T_k. Index nodes cannot be interchanged. Likewise, two successive index and vector, or vector and index nodes can be combined into a single index node. Two vector nodes can be combined into an index node, or into a vector node if the stride of the first vector node equals the stride times the count of the second.

Listing 1 Constructing the type path for given displacement sequence.

```
typenode *TypePath(int ix[], int n, typenode *T)
{
  if (n==1) {
    if (IsIdx(T,&nix,&c,&S)) {
      for (i=0; i<c; i++) nix[i] += ix[0];
      return idx(nix,c,S);
    } else return idx(1,ix,T);
  } else {
    c = LongestVector(ix,n,&d);
    if (c>1) {
      // displacement sequence for parent
      for (i=0; i<n/c; i++) six[i] = ix[i*c];
      S = vec(c,d,T); // new vector subtype
      return TypePath(six,n/c,S);
    } else {
      for (c=2; c<=n; c++)
        if (n%c==0&&Repeat(ix,c,n)) break;
      // displacement sequence for parent
      for (i=0; i<n/c; i++) six[i] = ix[i*c];
      // displacement sequence
      for (i=0; i<c; j++) nix[i] = ix[i]-ix[0];
      S = idx(c,nix,T); // new index subtype
      return TypePath(six,n/c,S);
    }
  }
}
```

A regularly strided sequence x of length n and stride d between consecutive indices can be represented differently by $\mathsf{vec}(n, d, C)$, by $\mathsf{vec}(n/c, dc, \mathsf{vec}(c, d, C))$ and by $\mathsf{vec}(c, dn/c, \mathsf{vec}(n/c, d, C))$; and each of the possibilities may have different cost. Thus, two successive vector nodes may be interchangeable, and there can therefore be different paths representing the given displacement sequence where one cannot be derived from the other by combination of successive nodes. In order to avoid this, and have a path with the property that any other path representing the given displacement sequence can be constructed from this one by simple operations on successive nodes, regularly strided, repeated prefixes have to be treated differently.

Our algorithm for constructing a type path representing the given displacement sequence is described by the recursive procedure shown in Listing 1 which constructs the path starting from the leaf. First `LongestVector` determines whether the given input sequence can be represented as a repeated vector (regularly strided prefix), and returns the length c of the longest possible such prefix. The parent node in the path is constructed by the recursive call with a displacement sequence that is c times shorter. If the index sequence cannot be represented by a vector node, a shortest repeated prefix is determined by calls to Repeat for each divisor c of n, and the procedure again called recursively on the corresponding, shorter displacement sequence. Given a

Listing 2 Subroutines for determining whether a prefix is repeated, and for finding the longest repeated vector.

```
1  int Repeat(int ix[], int c, int n)
2  {
3    for (i=0; i<n-c; i+=c) {
4      for (j=1; j<c; j++) {
5        if (ix[i+j]-ix[i]!=ix[i+c+j]-ix[i+c]) return 0;
6      }
7    }
8    return 1;
9  }

1  int LongestVector(int ix[], int n, int *d)
2  {
3    *d = ix[1]-ix[0]; c = n; i = 0;
4    do {
5      j = i; while (i<j+c-1&&ix[i+1]-ix[i]==*d) i++; i++;
6      if (i==n) return c; // longest subvector in ix
7      if (i<j+c) {
8        c = gcd(c,(i-j)); if (c==1) return 1;
9      }
10   } while (1);
11 }
```

displacement sequence of length n, the type path is constructed by calling procedure `TypePath` with this displacement sequence and leaf type con(1).

The check for and determination of longest repeated, regularly strided prefix can be performed in linear time. Checking whether prefix y of length c is repeated in x also takes linear time. Both of these procedures are shown in Listing 2.

The path constructed by procedure `TypePath` consists of d nodes, where d is at most $\lceil \log_2 n \rceil$ in the length of the input sequence. This is clear, since each recursive call is done on a displacement sequence of length n/c where c is a (particular) non-trivial divisor of n. We index the nodes T_i on this path from 0 to $d-1$. As we have argued, the path constructed has the following properties.

LEMMA 2. *The type path $T = T_0, T_1, \ldots T_{d-1}$ describing the given displacement sequence x of length n constructed by algorithm `TypePath` has length at most $\lceil \log_2 n \rceil$, and fulfills that any other type path description of x can be obtained by repeated application of*

- *combination of two successive nodes on the path, and*
- *splitting a vector node with count c into two vector nodes of counts c_0 and c_1 where $c = c_0 c_1$.*

Lemma 2 will be the basis for a dynamic programming algorithm [4] for constructing a least-cost path out of T. Let T_k and T_{k+1} be a node and its child, and let the counts in the two nodes be c_k and c_{k+1}. Two such nodes can always be replaced by a single node T'_k with count $c_k c_{k+1}$, simply by repeating the sequence described by T_{k+1} c_k times in T'_k offset by the indices of T_k. The cost of the combined node may be smaller than the cost of the two given nodes, as shown in Table 1. A vector node can be split into two nodes, with as many possibilities as there are divisors in the count n of the vector node. We cannot consider all possible splits and stay within the desired complexity bounds. We therefore assume that our cost function for vector nodes fulfills

$$\text{cost}(\text{vec}(n, d, T)) \quad \leq \quad \text{cost}(\text{vec}(n/c, dc, \text{vec}(c, d, T)))$$

for any split of n with divisor c. Splitting a vector node can therefore only lead to a reduction of the cost of the path if the split parts are combined with child and parent node. Table 1 also gives the cost of a splitting and combing a vector node that is sandwiched between two index nodes. Note that the rule for combining two successive vector nodes into a single vector node will actually never apply, since the longest repeated, regularly strided prefix is used for vector nodes. A path formed by combining successive nodes of the path constructed by algorithm `TypePath` will be called a *compressed path*.

We use the possible cost reduction at node k in a path $T_0, T_1, \ldots, T_k, \ldots T_{d-1}$ in a dynamic programming algorithm to compute a least-cost compressed path. Let $C[i,j]$ represent the cost of a least-cost compressed path between (original) typenodes T_i and T_j, $0 \leq i \leq j < d$; let $F_1[i,j]$ and $F_2[i,j]$ represent the first and second node on such a compressed path, and $L[i,j]$ the last node on such a path. The cost of the path we eventually seek is $C[0, d-1]$. Each entry $C[i,j]$ can be computed by dynamic programming by noting that $C[i,j]$ is either the smallest $C[i,k] + C[k+1,j]$ for $i \leq k < j$, or the cost of the path resulting by combining paths of cost $C[i,k]$ and $C[k+1,j]$, or the cost of the path resulting from splitting and combining a vector node sandwiched between two other nodes. The node information in $L[i,k]$ and $F_1[k+1,j]$ (node type and count) makes it possible to evaluate the cost saving by combining two nodes in constant time (see Table 1), since the actual displacement sequences of idx nodes do not have to be analyzed. Similarly, the information in $L[i,k]$ and $F_1[k+1,j]$ and $F_2[k+1,j]$ is needed to compute the cost saving by splitting and combining a vector node (see again Table 1). For the splitting operation all divisors of the count c of the vector node have to be considered. Over all nodes this is upper bounded by $2\lfloor\sqrt{n}\rfloor$, since there are at most this number of different divisors in n. The dynamic programming table has d^2 entries, and filling in line $i, 1 \leq i < d$ can be done in $O(i(d + \sqrt{n}))$ steps. Summarizing, we have:

LEMMA 3. *Finding the least-cost type path from a path $T_0, T_1, \ldots, T_k, \ldots T_{d-1}$ constructed by the `TypePath` algorithm takes $O(d^3 + d^2\sqrt{n})$ time steps.*

We can now complete the proof of Theorem 1.

PROOF. By Lemma 2 any other path than the one constructed by the `TypePath` algorithm can be obtained by combining or splitting sequences of successive nodes. The least-cost path can therefore be found by considering all possible split and combination points. Finding the shortest repeated prefix takes $O(n\sqrt{n})$ time steps since all divisors of n must be tried in the worst case (this is the bottleneck in the algorithm). The recursive call is done on a displacement sequence that has been reduced by the factor c of the length of the prefix, therefore also the overall running time is $O(n\sqrt{n})$ (solve the worst-case recurrence $T(n) \leq n\sqrt{n} + T(n/2)$). This is not compromised by the dynamic programming step, which takes $O(\log^3 n + \log^2 \sqrt{n}) = O(\log^2 \sqrt{n})$ steps. □

3.1 Examples

Table 2 lists a (partial) path (Column 1) of four nodes and two shorter paths that can possibly be constructed by combining two successive nodes (Columns 2 and 3); alternatively, from the path with a longest prefix vector node

Nodes T_k and T_{k+1}	$\text{cost}(T_k) + \text{cost}(T_{k+1})$	Combined node	Combined $\text{cost}(T_k)$
$\mathsf{idx}(c_k, \langle\ldots\rangle, \mathsf{idx}(c_{k+1}, \langle\ldots\rangle, T_{k+2}))$	$2K_{\mathsf{idx}} + c_k + c_{k+1}$	$\mathsf{idx}(c_k c_{k+1}, \langle\ldots\rangle, T_{k+2})$	$K_{\mathsf{idx}} + c_k c_{k+1}$
$\mathsf{idx}(c_k, \langle\ldots\rangle, \mathsf{vec}(c_{k+1}, d_{k+1}, T_{k+2}))$	$K_{\mathsf{idx}} + K_{\mathsf{vec}} + c_k + c$	$\mathsf{idx}(c_k c_{k+1}, \langle\ldots\rangle, T_{k+2})$	$K_{\mathsf{idx}} + c_k c_{k+1}$
$\mathsf{vec}(c_k, d_k, \mathsf{idx}(c_{k+1}, \langle\ldots\rangle, T_{k+2}))$	$K_{\mathsf{vec}} + K_{\mathsf{idx}} + c_{k+1}$	$\mathsf{idx}(c_k c_{k+1}, \langle\ldots\rangle, T_{k+2})$	$K_{\mathsf{idx}} + c_k c_{k+1}$
$\mathsf{vec}(c_k, d_k, \mathsf{vec}(c_{k+1}, d_{k+1}, T_{k+2}))$	$2K_{\mathsf{vec}}$	$\mathsf{vec}(c_k c_{k+1}, d_{k+1}, T_{k+2})$ if $c_{k+1} d_{k+1} = d_k$	K_{vec}
		$\mathsf{idx}(c_k c_{k+1}, \langle\ldots\rangle, T_{k+2})$ otherwise	$K_{\mathsf{idx}} + c_k c_{k+1}$
$\mathsf{vec}(c_k, d_k, \mathsf{con}(c_{k+1}))$	$K_{\mathsf{vec}} + K_{\mathsf{con}}$	$\mathsf{con}(c_k c_{k+1})$ if $c_{k+1} = d_k$	K_{con}

Nodes T_{k-1}, T_k, T_{k+1}	$\text{cost}(T_{k-1}) + \text{cost}(T_k) + \text{cost}(T_{k+1})$
$\mathsf{idx}(c_{k-1}, \langle\ldots\rangle, \mathsf{vec}(c_k, d_k, \mathsf{idx}(c_{k+1}, \langle\ldots\rangle, T_{k+2})))$	$2K_{\mathsf{idx}} + c_{k-1} + c_{k+1} + K_{\mathsf{vec}}$
$\mathsf{vec}(c_{k-1}, d_{k-1}, \mathsf{vec}(c_k, d_k, \mathsf{idx}(c_{k+1}, \langle\ldots\rangle, T_{k+2})))$	$K_{\mathsf{idx}} + c_{k+1} + 2K_{\mathsf{vec}}$
$\mathsf{idx}(c_{k-1}, \langle\ldots\rangle, \mathsf{vec}(c_k, d_k, \mathsf{vec}(c_{k+1}, d_{k+1}, T_{k+2})))$	$K_{\mathsf{idx}} + c_{k-1} + 2K_{\mathsf{vec}}$
$\mathsf{vec}(c_{k-1}, d_{k-1}, \mathsf{vec}(c_k, d_k, \mathsf{vec}(c_{k+1}, d_{k+1}, T_{k+2})))$	$3K_{\mathsf{vec}}$
Split-and-combined node	New, combined $\text{cost}(T_{k-1}) + \text{cost}(T_{k+1})$
$\mathsf{idx}(c_{k-1}c', \langle\ldots\rangle, \mathsf{idx}(c_k/c' c_{k+1}, \langle\ldots\rangle, T_{k+2}))$	$2K_{\mathsf{idx}} + c_{k-1}c' + c_k/c' c_{k+1} + K$

Table 1: Potential cost reduction by combining successive typenodes T_k and T_{k+1}, or by splitting and combining vector node T_k in the context of T_{k-1} and T_{k+1} where c' is a divisor of c_k.

idx(3) vec(2) vec(2) idx(3)	idx(3) vec(4) idx(3)	idx(6) idx(6)
$2K_{idx} + 6 + 2K_{\mathsf{vec}}$	$2K_{idx} + 6 + K_{\mathsf{vec}}$	$2K_{idx} + 12$
With $K_{\mathsf{idx}} = 3, K_{\mathsf{vec}} = 4$		
20	**16**	18
With $K_{\mathsf{idx}} = K_{\mathsf{vec}} = 10$		
46	36	**32**

Table 2: Example of partial type paths and their cost.

(Column 2) two other paths could be constructed by splitting the vector node (only the path in Column 3 would be considered by our algorithm). Depending on the concrete costs associated with the node types, different paths may be preferable. The dynamic programming step (applied to the path in Column 2) in any case finds the least-cost path.

4. PROCESSING COSTS

The cost model used so far serves to minimize the storage cost associated with a derived datatype description of a given layout. When processing a derived datatype, the displacement sequence represented by typenode T_k is repeated $c_0 c_1 \cdots c_{k-1}$ times where each c_i is the count associated with typenode T_i. Processing costs per node could be modeled by $\text{cost}(T_k) = c_0 c_1 \cdots c_{k-1}(K_T + c_k C_T)$ where K_T and C_T are constants associated with the specific nodetype of T_k. This cost function has the property that the cost of T_i is unaffected by the possible combining of nodes T_k and T_{k+1}; only the cost at T_k will change. The dynamic programming algorithm will therefore work also for this cost model and produce datatype paths of least processing costs. As can be seen, this cost measure has the consequence that datatype paths will tend to collapse into single nodes which are least efficient in terms of storage. The dynamic programming approach guarantees optimality for additive cost models ($\text{cost}(T) = \sum_i \text{cost}(T_i)$), and other, concrete models that better compromise between storage and (modeled) processing costs may well be possible.

5. TYPE NORMALIZATION

The algorithm of Section 3 shows that the type reconstruction problem can be solved in polynomial time. We can trivially use it to also solve the type normalization problem. Here a derived datatype (path) is already given, and we want to normalize this path into one of least cost. To do so, we could explicitly compute the type map described by the given derived datatype, and then apply type reconstruction to reconstruct a least-cost path. This is tedious, and highly undesirable, since type maps can be arbitrarily larger than the given derived datatype: a single vector node of constant cost and count n describes a type map of size n.

Fortunately, the type reconstruction algorithm tells us what to do. First convert index nodes where the associated displacement sequence is regularly strided into vector nodes. Then combine nested vectors into single vector nodes where this is possible (see Table 1). Then split index nodes into paths of nodes with smaller counts by applying the `Type-Path` algorithm on the associated displacement sequences. Finally, on this expanded path, the dynamic programming algorithm will construct the same, least-cost path as would the straight-forward, full expansion idea. We summarize as follows.

THEOREM 2. *Let T be a given derived datatype of depth d with n the length of the longest displacement sequence in any node. Then a least-cost tree describing the same type map as T can be constructed in $O(dn\sqrt{n} + (d \log n)^3)$ operations.*

6. INDEX TYPE NORMALIZATION

The MPI standard constructor `MPI_Type_indexed` allows to describe a displacement sequence as a sequence of blocks at given displacements, each of which consists of a regularly strided sequence of a given block length. The storage cost associated with this constructor is therefore $K_{\mathsf{idx}} + 2c$ since two arrays are needed for displacements and block lengths. Although `MPI_Type_indexed` can also only describe homogeneous layouts and the corresponding type tree is a path (index nodes have only one child), neither Lemma 1 nor Lemma 2 hold; many essentially different paths describing the layout may be possible. Thus, optimality proof and algorithm break down, and we do not know whether a polynomial time algorithm is possible in this case.

As a heuristic for both type reconstruction and normal-

ization that reduces the path cost, we suggest to convert a repeated prefix of length c into an indexed type when the number k of regularly strided blocks in the prefix is such that $2k < c$, conversely convert an indexed node with k blocks into an index-block node idx when $2k > c$.

7. MPI STANDARD

Optimal type normalization can be an expensive procedure, and different cost models may be suitable in different cases. As also pointed out in [8], it might be desirable to be able to control whether and which type normalization should be performed for given types. The `MPI_Type_commit` operation does not provide for this (no info argument or other handle). Note that the dynamic programming step precludes normalizing datatypes incrementally at the constructor calls.

8. CONCLUSION

By restricting the allowed derived datatype constructors to contiguous, vector and index-block, we showed that the datatype normalization problem of finding a least-cost datatype describing the same type map can be solved in polynomial time (and reasonably efficiently). The algorithm (as shown by the example) is more elaborate and on certain points (splitting of vector nodes; shortest prefixes in non-strided sequences) doing the opposite of the simple, greedy heuristics normally applied by MPI libraries. We conjecture that the problem is NP-hard when the `MPI_Type_create_struct` constructor is allowed. It is unclear if the index constructor alone will make the problem NP-hard.

9. REFERENCES

[1] E. Bajrović and J. L. Träff. Using MPI derived datatypes in numerical libraries. In *Recent Advances in Message Passing Interface. 18th European MPI Users' Group Meeting*, volume 6960 of *Lecture Notes in Computer Science*, pages 29–38. Springer, 2011.

[2] S. Byna, W. D. Gropp, X.-H. Sun, and R. Thakur. Improving the performance of MPI derived datatypes by optimizing memory-access cost. In *IEEE International Conference on Cluster Computing (CLUSTER 2003)*, pages 412–419, 2003.

[3] S. Byna, X.-H. Sun, R. Thakur, and W. Gropp. Automatic memory optimizations for improving MPI derived datatype performance. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, number 4192 in Lecture Notes in Computer Science, pages 238–246. Springer, 2006.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[5] 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.

[6] W. D. Gropp, E. Lusk, and D. Swider. Improving the performance of MPI derived datatypes. In *Third MPI Developer's and User's Conference (MPIDC'99)*, pages 25–30, 1999.

[7] 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*, volume 6305 of *Lecture Notes in Computer Science*, pages 132–141. Springer, 2010.

[8] 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.

[9] Q. Lu, J. Wu, D. K. Panda, and P. Sadayappan. Applying MPI derived datatypes to the NAS benchmarks: A case study. In *33rd International Conference on Parallel Processing Workshops (ICPP 2004 Workshops)*, pages 538–545, 2004.

[10] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.0*, September 21st 2012. `www.mpi-forum.org`.

[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*, volume 2840 of *Lecture Notes in Computer Science*, pages 404–413. Springer, 2003.

[12] G. Santhanaraman, J. Wu, W. Huang, and D. K. Panda. Designing zero-copy message passing interface derived datatype communication over Infiniband: Alternative approaches and performance evaluation. *International Journal on High Performance Computing Applications*, 19(2):129–142, 2005.

[13] 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.

[14] N. Tanabe and H. Nakajo. Introduction to acceleration for mpi derived datatypes using an enhancer of memory and network. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 15th European PVM/MPI Users' Group Meeting*, volume 5205 of *Lecture Notes in Computer Science*, pages 324–325. Springer, 2008.

[15] J. L. Träff. Alternative, uniformly expressive and more scalable interfaces for collective communication in MPI. *Parallel Computing*, 38(1–2):26–36, 2012.

[16] 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*, volume 1697 of *Lecture Notes in Computer Science*, pages 109–116. Springer, 1999.

[17] 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)*, pages 135–144, 2014.

[18] J. Wu, P. Wyckoff, and D. K. Panda. High performance implementation of MPI derived datatype communication over InfiniBand. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 14, 2004.