Runtime Checking of Datatype Signatures in MPI*

William D. Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439

Abstract. The MPI standard provides a way to send and receive complex combinations of datatypes (e.g., integers and doubles) with a single communication operation. The MPI standard specifies that the *type signature*, that is, the basic datatypes (language-defined types such as `int` or `DOUBLE PRECISION`), must match in communication operations such as send/receive or broadcast. Because datatypes may be defined by the user in MPI, there is a limitless collection of possible type signatures. Detecting the programmer error of mismatched datatypes is difficult in this case; detecting all errors essentially requires sending a complete description of the type signature with a message. This paper discusses an alternative: send the value of a function of the type signature so that (a) identical type signatures always give the same function value, (b) different type signatures often give different values, and (c) common cases (e.g., predefined datatypes) are handled exactly. Thus, erroneous programs are often (but not always) detected; correct programs never are flagged as erroneous. The method described is relatively inexpensive to compute and uses a small (and fixed, independent of the complexity of the datatype) amount of space in the message envelope.

1 Introduction

The Message Passing Interface (MPI) [3, 2] provides a standard and portable way of communicating data from one process to another, even for heterogeneous collections of computers. A key part of MPI's support for moving data is the description of data not as a series of undifferentiated bytes but as typed data corresponding to the datatypes natural to the programming language being used with MPI. Thus, when sending C `int`s, the programmer specifies that the message is made up of type `MPI_INT` (because MPI is a library rather than a language extension, MPI cannot use the same names for the types as the programming language). MPI further requires that the type of the data sent match the type of the data received; that is, if the user sends `MPI_INT`s, the user must

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

receive MPI_INTs.[1] MPI also allows the definition of new MPI datatypes, called *derived types*, by combining datatypes with routines such as MPI_TYPE_VECTOR, MPI_TYPE_STRUCT, and MPI_TYPE_HINDEXED. Because the matching of basic types is required for a correct program, a high-quality development environment should detect when the user violates this rule. This paper describes an efficient method for checking that datatype signatures match in MPI communication.

One of the reasons such error checking is important for MPI programs is that MPI allows messages containing collections of different datatypes to be communicated in a single message. Further, the sender and receiver are often in different parts of the program, possibly in different routines (or even programs). User errors in the use of MPI datatypes are thus difficult to find; adding this information can catch errors (such as using the same message tag for two different kinds of messages) that are difficult for the user to identify by looking at the code.

An additional complexity is that MPI requires only that the basic types of the data communicated match for example, that ints match ints and chars match chars. This ordered set of basic datatypes (i.e., types that correspond to basic types supported by the programming language) is called the *type signature*. The type signature is a tuple of the basic MPI datatypes. For example, three ints followed by a double is

$$(\texttt{MPI_INT}, \texttt{MPI_INT}, \texttt{MPI_INT}, \texttt{MPI_DOUBLE}).$$

A type signature has as many types as there are elements in the message. This makes it impractical to send the type signature with the message.

MPI also defines a *type map*; for each datatype, a displacement in memory is given. While the type map specifies both what and where data is moved, a type signature specifies only what is moved. Only the signatures need to match; this allows scatter/gather-like operations in MPI communication. For example, it is legal to send 10 MPI_INTs but receive a single vector (created with MPI_TYPE_VECTOR) that contains at least 10 MPI_INTs. Communicating with different type maps is legal as long as the type signatures are the same. Thus, it isn't correct to check that the datatypes match; only the type signatures must match.

Note that when looking at the type signature, the comparison is made with the basic types, even if the type was defined using a combination of derived datatypes. Thus, when looking at the type signature, any consecutive subsequence may have come from a derived datatype.

Consider the derived type t2 defined by the following MPI code fragment:

[1] Two exceptions to this rule are mentioned in Section 4. A third, mentioned in the MPI standard, is for the MPI implementation to cast the type; for example, if MPI_INT is sent but MPI_FLOAT is specified for the receive, an implementation is permitted to convert the integer to a float, following the rules of the language. As this is not required, it is nonportable. Further, no MPI implementation performs this conversion, and because it silently corrects for what is more likely a programming error, no implementation is ever likely to implement this choice.

```
MPI_Datatype t1, t2, types[2];
int          blen[2];
MPI_Aint     displ[2];
types[0] = MPI_INT;
types[1] = MPI_DOUBLE;
blen[0]  = 1;
blen[1]  = 1;
displ[0] = ...;
displ[1] = ...;
MPI_Type_struct( 2, blen, displ, types, &t1 );
types[0] = t1;
types[1] = MPI_SHORT;
blen[0] = 2;
MPI_Type_struct( 2, blen, displ, types, &t2 );
```

The derived type `t2` has the type signature

```
( (MPI_INT, MPI_DOUBLE), (MPI_INT, MPI_DOUBLE), MPI_SHORT ) =
( MPI_INT, MPI_DOUBLE, MPI_INT, MPI_DOUBLE, MPI_SHORT ).
```

The approach in this paper is to define a *hashing function* that maps the type signature to an integer tuple (the reason for the tuple is discussed in Section 3). The communication requirement is thus bounded independent of the complexity of the datatype; further, the function is chosen so that it can be computed efficiently; finally, in most cases, the cost of computing and checking the datatype signature is a small constant cost for each communication operation. Since this approach is a many-to-one mapping, it can fail to detect an error. However, the mapping is chosen so that it never erroneously reports failure. Further, for the important special case of communication with basic datatypes (e.g., `MPI_DOUBLE`), the test succeeds if and only if the type signatures match.

Other approaches are possible. The datatype definitions (just enough to reproduce the signature, not the type map) could be sent, allowing sender and receiver to agree on the datatypes. The definitions could be cached, allowing a datatype to be reused without resending its definition. The special case of (count,datatype) would reduce the amount of data that needed to be communicated in many common cases. Still, comparison of different datatypes in general would be complex, even if common patterns were exploited. Another approach is to send the complete type signature; this is the only approach that will catch all failures (various compression schemes can be used to reduce the amount of data that must be sent to describe the type signature, of course). Such an approach could be implemented over MPI by using the MPI-2 routines to extract datatype definitions, along with the MPI profiling interface. For systems with some kind of globally accessible memory, such as the Cray T3D, it is possible to make all datatype definitions visible to all processes, as in [1].

2 Datatype Hashing Function

We are looking for a function f that converts a type signature into a small bit range, such as a single integer or pair of integers. The cost of evaluating f should be relatively small; in particular, the cost of evaluating f for a type signature containing n copies of the same type (derived or basic) should be $o(n)$; for example, $\log n$. Because a type signature may contain an arbitrary number of terms, the easiest way to define f is by a binary operation applied to all of the elements of the type signature. That is, define a binary operation $\oplus$ that can be applied to a type signature $(\alpha_1, \ldots, \alpha_n)$ as follows:

$$f(\alpha_1) = \alpha_1$$
$$f((\alpha_1, \alpha_2, \ldots, \alpha_n)) = \bigoplus_{i=1}^{n} \alpha_i.$$

For example,

$$f(int, double) = (int) \oplus (double)$$

and

$$f(int, double, char) = (int) \oplus (double) \oplus (char).$$

In order to make it inexpensive to compute the hash function for datatypes built from an arbitrary combination of derived datatypes, the hash function must be associative. Since we want `(int,double)` to hash to a different value from `(double,int)`, we want the operation $\oplus$ to be noncommutative.

For this approach to be useful, the hash function must hash different datatypes to different hash values, particularly in the case of "common" errors, such as mismatched predefined datatypes.

3 A Simple Datatype Hashing Function

We need an operation that is both associative and noncommutative. Our approach is to define a tuple (α, n) where α is a datatype (derived or basic) and n is the number of basic datatypes in α. The action of $\oplus$ is given by

$$(\alpha, n) \oplus (\beta, m) \equiv (\alpha + (\beta << n), n + m),$$

where the operators $+$ and $<<$ are chosen to have the following properties:

$$(\alpha << n) << m = \alpha << (n + m) \quad (1)$$
$$(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma) \quad (2)$$
$$(\alpha << n) + (\beta << n) = (\alpha + \beta) << n. \quad (3)$$

One choice for these operators is bitwise exclusive or (xor) for $+$ and circular left shift for $<<$. These operations are often chosen for hash functions because they are very cheap to apply. They have the necessary properties, as can be

proven by writing the α and so forth as bit vectors and then applying the operations xor and circular shift to those bit vectors. Another choice of operators is integer addition modulo 2^{32} for $+$ and circular left shift by 3 for $<<$ (that is, $a << 1$ is a, shifted left three bits).

These properties allow us to prove that the operation $\oplus$ is associative:

$$\begin{aligned}((\alpha, n) \oplus (\beta, m)) \oplus (\gamma, p) = \\ ((\alpha + (\beta << n), n + m)) \oplus (\gamma, p) = \\ ((\alpha + (\beta << n) + (\gamma << n + m), n + m + p) = \\ ((\alpha + ((\beta + (\gamma << m) << n)), (n + (m + p)) = \\ ((\alpha, n) \oplus (\beta + (\gamma << m), m + p)) = \\ ((\alpha, n) \oplus ((\beta, m) \oplus (\gamma, p))).\end{aligned}$$

The operation $\oplus$ is not commutative:

$$\begin{aligned}(\alpha, n) \oplus (\beta, m) = \\ (\alpha + (\beta << n), n + m) \\ (\beta, m) \oplus (\alpha, n) = \\ (\beta + (\alpha << m), n + m),\end{aligned}$$

but

$$(\alpha + (\beta << n)) \neq (\beta + (\alpha << m))$$

except in special cases.

Note that addition and xor by itself are commutative; the shift operation provides a noncommutative operation.

We will use this operation to build f. Specifically, we will apply $\oplus$ to a type signature where we have replaced every basic type with a tuple containing an integer representing the type and a one, indicating a single basic type. That is,

$$(int, double, char)$$

becomes

$$((int, 1), (double, 1), (char, 1))$$

and

$$f((int, double, char)) = (int, 1) \oplus (double, 1) \oplus (char, 1).$$

3.1 Cost of Evaluating f

Several identities can be used to reduce the cost of computing f. One important case is a type signature containing a large number of the same basic type. This is the signature that represents the most common MPI usage: a send with a basic datatype and a count that is greater than one. Using a method that is very similar to the approach for evaluating integer powers of matrices, we can

compute $\bigoplus_{i=1}^{m}(\alpha, n)$ in $O(\log(m))$ time by induction. Let m be 2^k for some k. Then

$$\bigoplus_{i=1}^{m}(\alpha, n) = \left(\bigoplus_{i=1}^{m/2}(\alpha, n)\right) \oplus \left(\bigoplus_{i=1}^{m/2}(\alpha, n)\right),$$

the terms on the right are evaluated by inducation. This can be evaluated with $\log_2 m$ evaluations. The generalization to arbitrary m is left to the reader.

Further, note that $v << n = v << (n + wordsize) = v << (n \bmod wordsize)$; this can be used to reduce the cost of evaluating f.

Finally, by exploiting the associative property of $\oplus$, evaluating f for a new derived datatype involves only the values of f for the datatypes that make up the new datatype (with the exception of those containing types `MPI_PACKED` or `MPI_BYTE`). Thus, computing f for a datatype has cost proportional only to the number of different datatypes (either user-defined or basic) used in the definition and proportional to the log of the number of instances of each datatype.

3.2 Hash Function Quality

For the hash function to be useful, collisions should be rare. Since in a typical program, MPI type signatures are *not* randomly distributed, it makes the most sense to experimentally evaluate some common datatype patterns. Further, while there are 13 distinct basic MPI datatypes in the C binding, most programs use only a few types, such as `MPI_INT` and `MPI_DOUBLE`. Types such as `MPI_UNSIGNED_CHAR` are rarely used. Thus, for most applications, only a few basic datatypes will appear. To see how likely a collision in the hash function might be, we tested the following patterns:

$$n : \alpha_i \tag{4}$$

$$m : (1 : \alpha_i, \quad (n-1) : \alpha_j) \tag{5}$$

$$1 : \alpha_i, \quad m : (1 : \alpha_i, \quad (n-1) : \alpha_j), \tag{6}$$

where $n : x$ means n copies of x. These correspond to the cases of count (n) of a basic datatype (4), count m of a structure containing n members (5), and a structure containing count m of another structure (6). Various values of n and m were used.

Table 1 shows the results of the tests. Clearly, only the choice of integer addition with medium-sized integers provides an effective hash function; with this choice, only one in one hundred different type signatures hashed to the same value. Further experiments may identify improved hash functions.

3.3 Improving the Type Signature Test

One modification of the approach is to optimize for the special case of count copies of a datatype (basic or otherwise), since this is the fundamental unit in MPI (all MPI communication operations send count copies of a given datatype).

Table 1. Results of tests of the hash function. Collisions is the percentage of type signatures whose hash value was the same as a different type signature. Duplicates gives the percentage of hash values that were duplicated. Operand indicates whether the representation for a basic datatype is a small integer (less than 32) or a larger integer (less than 2^{16}). We tested 4625 different type signatures.

Operator1	Operator2	Operand	Collisions	Duplicates
xor	rotate 1	small	57.4	13.4
xor	rotate 3	small	48.9	10.5
+	rotate 1	small	24.9	11.5
+	rotate 3	small	29.4	10.3
xor	rotate 1	medium	45.6	9.8
+	rotate 1	medium	1.2	0.58

if (α_{send} != α_{recv}) then
  if (α_{send} and α_{recv} is basic) then error
  else if ($\bigoplus_{i=1}^{count_{send}}(\alpha_{send}, n_{send})$!=
      $\bigoplus_{i=1}^{count_{recv}}(\alpha_{recv}, n_{recv})$) then error
  endif
endif

Fig. 1. Modification to test to provide exact handling of the most common case.

In this case, we send $(count, \alpha, n)$. The modified test is shown in Figure 1.

Note that the *count* applied in the receive case is the actual count, not the maximum count that is provided by user in the `MPI_RECV` call. In addition, we do not need to send the count separately; we can simply use a single bit to indicate that the datatype is basic and the count can be computed, if necessary, from the length of data sent. With this modification, basic datatypes are handled exactly (all errors are detected).

4 Limitations

MPI allows users to send partial datatypes. That is, the user can define a datatype representing, for example, an `int` followed by ten `doubles`, and receive this into a datatype of an `int` followed by fifty `doubles`, as long as the type signature of the data that is sent matches the type signature at the receiver for all of the types that are used. This allows the user to define a maximum-sized datatype on the receive end but an actual sized datatype on the send end.

In MPI, the user can detect this by examining the `MPI_Status` value returned by the receive. If the routine `MPI_GET_COUNT` returns `MPI_UNDEFINED`, then the routine `MPI_GET_ELEMENTS` may be used to determine how many elementary (predefined) MPI datatypes were sent. In the case above, `MPI_GET_ELEMENTS` would return eleven (one `int` plus ten `doubles`).

Our test does not handle this. Thus, it must also test for `MPI_GET_COUNT` being `MPI_UNDEFINED`; in that case, the test passes (even if the type signature do not, in fact match).

In principle, a corresponding value of f could be constructed by using the same process that is used in an MPI implementation to evaluate `MPI_GET_ELEMENTS`; by integrating the computation of f with this routine, this test can be performed with low additional cost.

The MPI datatype `MPI_PACKED` and `MPI_BYTE` also present special problems. `MPI_PACKED` can be handled by exploiting the availability of a header in a packed buffer. `MPI_BYTE` explicitly turns off type signature matching and is best handled with a reserved hash value (e.g., `0xFFFFFFFF,-1`).

5 Conclusion

We have shown an efficient way to catch many user errors caused by type signature mismatch at run time in MPI programs. The cost is relatively small; consuming only an additional 32 to 64 bits (4 to 8 bytes) of message header and evaluation cost that is bounded by $O(m \log n)$ for derived datatypes containing m different types with repeat count $\leq n$. The most common cases (count of a basic datatype) take constant time. We note that this approach can be used for any system that incrementally packs and unpacks data, such as XDR or PVM.

Acknowledgments

I thank Lloyd Lewins for the suggestion of using a hashing function to support error checking of derived datatypes, and Rusty Lusk for his valuable comments.

References

1. Jason Hunter. Datatype checking in Cray T3D native MPI. Technical Report EPCC-SS95-07, Edinburgh Parallel Computing Centre, 1995.
2. Message Passing Interface Forum. MPI: A message-passing interface standard. `http://www.mpi-forum.org`.
3. Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.