Hash Functions for Datatype Signatures in MPI

Julien Langou, George Bosilca, Graham Fagg, and Jack Dongarra

Dept. of Computer Science, The University of Tennessee, Knoxville, TN 37996

Abstract. Detecting misuse of datatypes in an application code is a desirable feature for an MPI library. To support this goal we investigate the class of hash functions based on checksums to encode the type signatures of MPI datatype. The quality of these hash functions is assessed in terms of hashing, timing and comparing to other functions published for this particular problem (Gropp, 7th European PVM/MPI Users' Group Meeting, 2000) or for other applications (CRCs). In particular hash functions based on Galois Field enables good hashing, computation of the signature of unidatatype in $\mathcal{O}(1)$ and computation of the concatenation of two datatypes in $\mathcal{O}(1)$ additionally.

1 Introduction

MPI datatypes are error prone. Detecting such errors in the user application code is a desirable feature for an MPI library and can potentially provide an interesting feedback to the user. Our goal is to detect when the type signature from the sender and the receiver do not respect the MPI standard. The cost of doing so should be negligible with respect to the cost of the communication.

The idea was previously mentioned by Gropp [3] and we merely agree with his solution and specifications that we recall in Section 2. In Section 3, we give a more general framework to his study that enables us to rederive his solution but also create new solutions. In particular, our hash functions have the property of being $\mathcal{O}(1)$ time to solution for computing the signature of an unidatatype. We conclude with some experimental results that assess the quality of our hash functions in term of hashing and timing. The codes to reproduce the experiments are available online [7].

2 Specifications of the Problem

The MPI standard [1,2] provides a full set of functions to manipulate datatypes. Using the basic datatypes predefined by the standard, these functions allow the programmer to describe most of the datatypes used in parallel applications from regular distributions in memory (i.e. contiguous, vector, indexed) to more complex patterns (i.e. structures and distributed arrays). The type signature of a datatype is defined as the (unique) decomposition of a datatype in a succession of basic datatypes. From the MPI standard point of view, a communication is correct if and only if the type signature of the sender exactly matches the

B. Di Martino et al. (Eds.): EuroPVM/MPI 2005, LNCS 3666, pp. 76–83, 2005.

beginning of the type signature of the receiver. For the remaining of this paper we consider that the type signature of the sent datatype exactly matches the type signature of the received datatype (in its whole). (The case of a type signature of the datatype of the receiver longer than the one of the sender is dealt in the last paragraph of the paper.)

In the framework developed by Gropp [3], a hash function is used to create a signature for the type signature of the datatype of the sender, the receiver then checks whether the signature of the sender matches the signature of the type signature of its datatype. Note that some errors might still be unnoticed when two type signatures have the same hash value.

At this point of the specification, any hash functions existing in the literature would be adequate. However, the fact that MPI datatypes can be combined together implies that we would like to be able to efficiently determine the signature of the concatenatenation of two datatypes. (Note that this is not only a desirable feature but also mandatory since we want to check the datatype when the count of the sender and the receiver mismatch.) The datatype signature function σ shall be such that the signature of the concatenation of two datatypes can be obtained from the signatures of the two datatypes, we therefore need an operator $\odot$ such that:

$$\sigma([\alpha_1, \alpha_2]) = \sigma(\alpha_1) \odot \sigma(\alpha_2).$$

If there exists such a $\odot$ for σ, we call σ assossiative.

We call unidatatype a derived datatype made of just one datatype (derived or not). This type or class of datatype is fairly frequent in user application codes and we therefore also would like to be able to efficiently compute their signature.

3 Some Associative Hash Functions

3.1 Properties of Checksum-Based Hash Functions

Considering a set of elements, E, and two binary operations in E, $\oplus$ (the addition, required to be associative) and $\otimes$ (the multiplication), the checksum of $X = (x_i)_{i=1,\ldots,n} \in E^n$ is defined as

$$f(X) = f((x_i)_{i=1,\ldots,n}) = \bigoplus_{i=1}^{n} (x_i \otimes \alpha_i), \tag{1}$$

where α_i are predefined constants in E.

Let us define the type signature of the datatype X, $\sigma(X)$, as the tuple

$$\sigma(X) = (f(X), n). \tag{2}$$

In this case, we can state the following theorem.

Theorem 1. *Given $(E, \oplus, \otimes)$ and an $\alpha \in E$, defining the checksum function, f, as in Equation (1), and the type signature σ as in Equation (2) and providing that*

$$\oplus \text{ and } \otimes \text{ are associative ,} \tag{3}$$

$$\text{the } \alpha_i \text{ are chosen such that } \alpha_i = \bigotimes_{j=1,\dots,i} \alpha = \alpha^i, \tag{4}$$

$$\text{for any } x, y \in E, \quad (x \otimes \alpha) \oplus (y \otimes \alpha) = (x \oplus y) \otimes \alpha, \tag{5}$$

then the concatenation datatype operator, $\odot$, is defined as

$$\sigma(X) \odot \sigma(Y) = ((f(X) \otimes \alpha^m) \oplus f(Y), n+m), \tag{6}$$

and satisfies

$$\sigma([X,Y]) = \sigma(X) \odot \sigma(Y). \tag{7}$$

for any $X \in E^n$ and $Y \in E^m$.

Proof. The proof is as follows, let us define $X = (x_i)_{i=1,\dots,n}$, $Y = (y_i)_{i=1,\dots,m}$ and $Z = [X,Y] = (z_i)_{i=1,\dots,n+m}$, then

$$\begin{aligned} f(Z) &= \bigoplus_{i=1}^{n+m} (z_i \otimes \alpha^i) = (\bigoplus_{i=1}^{n} x_i \otimes \alpha^{m+i}) \oplus (\bigoplus_{i=1}^{m} y_i \otimes \alpha^i) \\ &= (\bigoplus_{i=1}^{n} ((x_i \otimes \alpha^i) \otimes \alpha^m)) \oplus (\bigoplus_{i=1}^{m} y_i \otimes \alpha^i) \\ &= (\left(\bigoplus_{i=1}^{n} (x_i \otimes \alpha^i) \right) \otimes \alpha^m) \oplus (\bigoplus_{i=1}^{m} y_i \otimes \alpha^i) \\ &= (f(X) \otimes \alpha^m) \oplus f(Y) \end{aligned}$$

The equality of the first line is the consequence of the associativity of $\oplus$ (3), the second line is the consequence of the associativity of $\otimes$ (3) and the definition of the α_i (4), the third line is the consequence of the distributivity of $\oplus$ versus $\otimes$ (5).

In our context, E is included in the set of the integers ranging from 0 to $2^w - 1$ (i.e. E represents a subset of the integers that we can be encoded with w bits). In the next three sections we give operators, $\oplus$ and $\otimes$ that verifies (3), (4) and (5) over E.

Any binary operations over E, $\oplus$ and $\otimes$, such that $(E, \oplus, \otimes)$ is a ring verifies the necessary properties (3), (4) and (5), therefore our study will focus on rings over E.

3.2 Checksum `mmm-bs1`

Given any integer a, $(E, \oplus, \otimes)$ where E is the set of integer modulo a, $\oplus$ the integer addition modulo a and $\otimes$ the integer multiplication modulo a defines a ring. (Even a field iff a is prime.)

A natural choice for α and a is $\alpha = 2$ and $a = 2^w - 1$ which defines the signature `mmm-bs1`.

The multiplication of integers in E by power of 2 modulo $2^w - 1$ corresponds to a circular leftshift over w bits, $\ll_{c,w}$. This operation as well as the modulo $2^w - 1$ operation can both be efficiently implemented on a modern CPU.

Remark 1. Note that the type signature is encoded on w bits but only $2^w - 1$ values are taken.

Remark 2. The computation of the signature of a unidatatype, X, that is composed of n identical datatypes x, costs as much as a single evaluation thanks to the formula:

$$f(X) = f([x, .., x]) = (x \otimes 2^0) \oplus ... \oplus (x \otimes 2^{n-1}) = x \otimes (2^n \ominus 1), \qquad (8)$$

where $\ominus$ denotes the substraction operation in the group $(E, \oplus)$. The evaluation of 2^n in formula (6) is efficiently computed thanks to $1 \ll_{c,w} n$.

Remark 3. In [3], $\oplus$ is set to the integer addition modulo 2^w and $\otimes$ is set to the circular leftshift over w bits (that is the integer multiplication by power of 2 modulo $2^w - 1$ for numbers between 0 to $2^w - 2$ and the identity for $2^w - 1$). This mix of the moduli breaks the ring property of $(E, \oplus, \otimes)$, (the distributivity relation (5) is not anymore true,) and consequently the Equation (6) is not true. We definitely do not recommend this choice since it fails to meet the concatenation requirement.

3.3 Checksum `xor-bs1`

Gropp [3] proposed to use the xor operation, $\wedge$, for the addition and a circular leftshift over w bits by one, $\ll_{c,w}$, for the multiplication operation. E represents here the integers modulo 2^w. The condition of the Theorem 1 holds and thus this represents a valid choice for $(E, \oplus, \otimes)$.

Note that in this case, we can not evaluate in $\mathcal{O}(1)$ time the checksum of unidatatype datatype. (See [3, §3.1], for a $\mathcal{O}(log(n))$ solution.)

3.4 Checksum `gfd`

Another ring to consider on the integers modulo 2^w is the Galois field GF(2^w). (A comprehensive introduction to Galois field can be found in [4].)

The addition in GF(2^w) is xor. The multiplication in GF(2^w) is performed thanks to two tables corresponding to the logarithms, `gflog`, and the inverse logarithms, `gfilog` in the Galois Field thus

$$a \otimes b = \texttt{gfilog}(\texttt{gflog}[a] + \texttt{gflog}[b]).$$

where $+$ is the addition modulo 2^w. This requires to store the two tables `gflog` and `gfilog` of 2^w words of length w bits.

Since $(E, \oplus, \otimes)$ is a ring, the Theorem1 applies and thus the signature of the concatenation of two datatypes can be computed thanks to formula (6). Note that the value 2^n in the formula (6) is directly accessed from the table of the inverse logarithms since $2^n = gfilog[n-1]$ (see [4]).

Finally since we have a field, we can compute type signatures of unidatatype in $\mathcal{O}(1)$ time via

$$x \otimes 2^0 \oplus \ldots \oplus x \otimes 2^n = x \otimes (2^{n+1} \ominus 1) \oslash (2 \ominus 1) = x \otimes (2^{n+1} \ominus 1) \oslash (3), \quad (9)$$

where $\oslash$ denotes the division operation in the field $(E, \oplus, \otimes)$.

`gfd` needs to store two tables of 2^w word each of length w therefore we do not want w to be large. (A typical value would be $w = 8$.) In this case to encode the derived datatype on 32 bits, we would use $m = 4$ checksums in GF($2^{w=8}$).

3.5 Cyclic Redundancy Check `crc`

Since we are considering hash functions, we also want to compare to at least one class of hash functions that are known to be efficient in term of computation time and quality. We choose to compare with the cyclic redundancy check.

Stating briefly if we want to have a w-bit CRC of the datatype X, we interpret X as a binary polynomial and choose another binary polynomial C of degree exactly equal to w (thus representing a number between 2^w and $2^{w+1} - 1$). The CRC, R, is the polynomial $X \times x^w$ modulo C, that is to say:

$$X.x^w = Q.C + R.$$

For a more detailed explanations we refer to [5]. Various choices for C are given in the literature, we consider in this paper some standard value, for more about the choice of C, we referred to [6].

The concatenation operation is possible with CRC signatures. If $X = (x_i)_{i=1,\ldots,n}$ and $Y = (y_i)_{i=1,\ldots,m}$ are two datatypes and we have computed their signatures $\sigma(X) = (R_X, n)$ $\sigma(Y) = (R_Y, m)$, then

$$\sigma([X, Y]) = (R_Z, n + m)$$

where R_Z is $R_X x^m + R_Y$ modulo Q.

Even though it is possible given the signatures of two datatypes to compute the signature of the concatenation, the cost of this operation is proportional to the size of the second data-type (in our case m). This is an important drawback in comparison with the checksum where the cost of a concatenation is $\mathcal{O}(1)$. Although, it is not possible to compute quickly the signatures of a datatype composed of all the same datatype.

4 Experimental Validation of the Hash Functions to Encode MPI Datatype

4.1 Software Available

Our software is available on the web [7]. We believe it is high quality in the sense of efficiency, robustness and ease of use. It is provided with a comprehensive set of testing routines, timing routines and hash function quality assessment routines. The experimental results presented thereafter are based on this software and thus are meant to be easily reproducible. We can also verify the correctness of the theoretical results in part 3 through the software.

4.2 Quality of the Hash Functions

To evaluate the quality of a hash function we consider a sample of datatypes and check how often collisions appear. (Note, that since the number of values taken by the hash function is finite (encoded on 16 or 32 bits) and the number of datatypes is infinite, collisions in the value of the hash functions are unavoidable.) Considering the fact that we are interested in the quality of the hash function when applied to MPI datatypes, it makes sense to have a sample that reflects what an MPI library might encounter in an application code. In this respect, we follow the experimental method of Gropp [3]. In our experiments, we consider 6994 datatypes that are made of $wg = 13$ different pre-defined datatypes. (We refer to [7] for the exact description of the datatypes.)

The results of the hash functions are given in Table 1. Two quantities are given: the percentage of collisions and the percentage of duplicates. A collision is when a type signature has its hash value already taken by another type signature in the sample. A duplicate is a hash value that has strictly more than one type signature that maps to it in the sample.

Since we are mapping 6994 words on 2^{16} value, it is possible to give a *perfect hash function* for our sample example. (That is to say a function where no collision happens.) However this is not the goal. We recall the fact that if we apply a random mapping to m out of n possible states. Then we expect that this mapping will produce about

$$n/(n/m + (1/2) + \mathcal{O}(m/n)) \tag{10}$$

distinct results providing $\sqrt{n} < m$. Thus, a random mapping from $m = 6994$ to $n = 2^{16}$ shall give about 5.07% collisions. This number is representative of a good hash functions.

We considered three different CRCs (namely 16-bit CRC/CCITT,XMODEM and ARC) but only report the one of CRC/CCITT that is best suited to the considered panel.

Table 1. Percentage of collisions and duplications for some 16-bit and 32-bit hash functions. We have used the panel of 6994 type-signatures described in [7].

16 bit type signature			32 bit type signature		
	Collisions	Duplicates		Collisions	Duplicates
`16-bit CRC/CCITT`	4.76 %	4.88 %	`CRC-04C11DB7`	0.00 %	0.00 %
`xor-bs1`(w=16,m= 1)	61.10 %	26.75 %	`xor-bs1`(w=16,m= 2)	37.32 %	15.76 %
`mmm-bs1`(w=16,m= 1)	52.03 %	19.67 %	`mmm-bs1`(w=16,m= 2)	23.29 %	11.48 %
`gfd` (w=16,m= 1)	12.65 %	8.48 %	`gfd` (w=16,m= 2)	0.00 %	0.00 %
`xor-bs1`(w= 8,m= 2)	64.40 %	23.04 %	`xor-bs1`(w= 8,m= 4)	60.71 %	13.65 %
`mmm-bs1`(w= 8,m= 2)	51.44 %	27.47 %	`mmm-bs1`(w= 8,m= 4)	30.08 %	14.91 %
`gfd` (w= 8,m= 2)	3.73 %	3.77 %	`gfd` (w= 8,m= 4)	0.00 %	0.00 %

From Table 1, `gfd` performs fairly well, it is as good as `16-bit CRC-CCITT` or a random mapping (5.07%). To have a better hash function, one can simply increase the number of bits on which the data is encoded. In Table 1, we also give the percentage of collisions and duplications for 32-bit signatures, we obtain a perfect hash function for `gfd` and `CRC-04C11DB7`. In conclusion, `gfd` has the same quality of hashing as some well known CRCs on our sample and is much better than `xor-bs1` and `mmm-bs1`.

4.3 Timing Results for the Hash Functions

We present timing results of optimized routine for `crc` and `gfd`. The code is compiled with GNU C compiler with -O3 flag and run on two architectures: `torc5.cs.utk.edu` a Pentium III 547 MHz and `earth.cs.utk.edu` an Intel Xeon 2.3 GHz, both machines are running Linux OS. Results are presented in Table 2. `signature` represents the time to encode a word of length nx, `concatsignature` represents the time to encode the concatenation of two words of size $nx/2$, `unisignature` represents the time to encode a word of length nx with all the same datatype (basic or derived). Either for `crc` or for `gfd`, the computation of a signature lasts $\mathcal{O}(n)$ time. The $\mathcal{O}(1)$ time for the `concatsignature` and `unisignature` is an obvious advantage of `gfd` over `crc`. Note that in Table 2, the encoding with `gfd` is done by a pipeline algorithm ($\mathcal{O}(n)$ work) whereas encoding with a tree algorithm by using `concatsignature`. Such an algorithm should have given us better performance for `gfd signature` (see Gropp [3, §]).

Table 2. Time in μs to compute the datatype signatures of different MPI datatype type signatures. for `gfd`$(w = 8, m = 2)$ and $w = 16$ for `crc`$(w = 16)$.

Pentium III (547 MHz)					
$nx = 100$			$nx = 1000$		
	`gfd`	`crc`		`gfd`	`crc`
`signature`	17.80μs	11.68μs	`signature`	176.88μs	114.87μs
`concatsignature`	0.30μs	5.95μs	`concatsignature`	0.34μs	57.48μs
`unisignature`	0.36μs	11.68μs	`unisignature`	0.37μs	114.87μs
$nx = 100$ Intel Xeon 2.392 GHz					
$nx = 100$			$nx = 1000$		
	`gfd`	`crc`		`gfd`	`crc`
`signature`	4.60μs	2.04μs	`signature`	48.26μs	20.21μs
`concatsignature`	0.09μs	1.12μs	`concatsignature`	0.10μs	10.07μs
`unisignature`	0.09μs	2.04μs	`unisignature`	0.10μs	20.21μs

5 Conclusions

In this paper, we assess that checksums based on Galois field provide a quality of hashing comparable to CRCs (on our test problem). The big advantage of `gfd` is that the computation of signatures of unidatatypes is $\mathcal{O}(1)$ work and the computation of concatenations of two datatypes is $\mathcal{O}(1)$ work (CRCs are $\mathcal{O}(n)$ for those two special cases). In a typical application, having `concatsignature` and `unisignature` in $\mathcal{O}(1)$ leads to the computation of most signatures in $\mathcal{O}(1)$ times as well.

Notes and Comments. The MPI standard requires the type signature (datatype, count) of the sender to fit in the first elements of the type signature (datatype, count) of the receiver. When the number of basic datatypes in the (datatype, count) of the receiver is longer than the number of basic datatypes in the (datatype, count) of the sender, the receiver needs to look inside the structure of its datatype to find the point when the number of basic datatypes is the same as the one sent. The MPI correctness of the communication can then be assessed by checking if the signature of this part of the datatype matches the signature of the sender.

Special care has to be taken for the datatypes MPI_PACKED and MPI_BYTE (see Gropp [3]).

More information theory could have been exploited, for example, `gfd`$(w = 8, m = 2)$ guarantees that any swap between two basic datatypes is detected as long as there is less than 253 basic datatypes in the two derived datatypes considered. These considerations are out of the scope of this paper.

References

1. Message Passing Interface Forum: MPI: A message-passing interface standard. http://www.mpi-forum.org
2. Message Passing Interface Forum: MPI: A message-passing interface standard. International Journal of Supercomputer Applications **8** (1994) 165–414
3. Gropp, W.D.: Runtime checking of datatype signatures in MPI. In Dongarra, J., Kacsuk, P., Podhorszki, N., eds.: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 1908 in Springer Lecture Notes in Computer Science (2000) 160–167. 7th European PVM/MPI Users' Group Meeting, http://www-unix.mcs.anl.gov/~gropp/bib/papers/2000/datatype.ps
4. Plank, J.S.: A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Software – Practice & Experience **27** (1997) 995–1012. http://www.cs.utk.edu/~plank/plank/papers/CS-96-332.html
5. Knuth, D.E.: The Art of Computer Programming, 2nd Ed. (Addison-Wesley Series in Computer Science and Information). Addison-Wesley Longman Publishing Co., Inc. (1978)
6. Koopman, P., Chakravarty, T.: Cyclic redundancy code (CRC) polynomial selection for embedded networks. IEEE Conference Proceeding (2004) 145–154. 2004 International Conference on Dependable Systems and Networks (DSN'04)
7. Langou, J., Bosilca, G., Fagg, G., Dongarra, J.: TGZ for hash functions of MPI datatypes (2004) http://www.cs.utk.edu/~langou/articles/LBFD:05/2005-LBFD.html