2018 IEEE International Parallel and Distributed Processing Symposium Workshops

Non-Intrusively Avoiding Scaling Problems in and out of MPI Collectives

Hongbo Li, Zizhong Chen, and Rajiv Gupta
Department of Computer Science and Engineering
University of California, Riverside
Riverside, USA
{*hli035, chen, gupta*}*@cs.ucr.edu*

Min Xie
College of Computer
National University of Defense Technology
Changsha, China
xiemin@nudt.edu.cn

***Abstract*—It has been observed that scaling problems are highly likely to manifest when MPI applications are launched at a large scale where the scale is characterized by the degree of *parallelism* and the *problem size*. As the complexity of MPI collectives is directly impacted by both parallelism scale and problem size, their use often triggers scaling problems. Scaling problems' root cause can be outside of MPI libraries and these can be easily exposed via the dynamic interaction between user code and MPI library as the scale goes up. Specifically, irregular collectives suffer the most as the *C int* displacement array can easily be corrupted with integer overflow. Scaling problems can also result from a bug inside the released MPI libraries due to the lack of a systematic testing of MPI libraries as well as the platform or environment dependency of some scaling problems. Hence it is important for library users to perform testing on their platform to expose potential scaling problems. Fixing a scaling problem is challenging, and thus it usually takes much time for users to wait for an official fix, which sometimes is not even possible due to the difficulty of bug reproduction, root-cause identification, and fix development. To improve users' productivity, we establish the necessity of user side testing and provide a protection layer to avoid scaling problems non-intrusively, i.e., without requiring any changes to the MPI library or user programs. This provides an immediate remedy when an official fix is not readily available.**

I. Introduction

MPI has been the de facto standard of message passing based parallel programming model on distributed memory systems. However, application developers face the challenge of dealing with bugs whose root-cause is often hard to locate because errors in one process can easily propagate to other processes via communication. To make matters worse, some errors, such as integer overflow and resource exhaustion, manifest only at *large scale*. We refer to them as *scaling problems* [36], [37], [33], [26].

It has been recognized that program *scale* has two dimensions: *parallelism scale*, i.e. the number of parallel processes; and *problem size* [37] that impacts the *message size* that must be handled by the MPI library. Thus, scaling problems can be triggered by large values in either one dimension or both leading to the following natural classification: *Type-1* problems are only triggered by a large parallelism scale; *Type-2* problems are only triggered by a large problem size; and *Type-3* problems are triggered by the combination of the two. We collected a list of well-documented scaling problems reported online as shown in Table I. Also, we detected new bugs in various MPI versions that are listed in Table II. A scaling problem is classified as Type-3 if the description stresses both parallelism scale and message size, as Type-2 if it is only related to message size, and as unknown (either Type-2 or Type-3) if it depends on message size while its dependence on parallelism scale is unknown. With such classification, ten scaling problems are Type-3 (Prob. 1-5, 9-13), one is Type-2 (Prob. 6), and two are unknown (Prob. 7-8). To our knowledge, Type-3 is the most common type of scaling problem, Type-2 is next, and Type-1 is the least common as in our investigation we are yet to observe a Type-1 problem. *This paper focuses on Type-2 and Type-3 scaling problems – Type-3 problems are discussed in the context of MPI collectives as collective communication directly depends on* both *the parallelism scale and the problem size.*

A. Scaling Problems Observed

Scaling problems can be exposed in the dynamic interaction between user code and MPI library. In the interaction, the target program runs with various number of processes and demands the passing of messages of differing lengths. In extreme cases, the use of too many processes (too large messages) causes the corruption of MPI routines though it only demands communications of messages of moderate lengths (a moderate number of processes). Among all the MPI routines, *irregular collectives*, that enable processes to transfer varying amounts of data, suffer from this problem the most due to their use of *C int displacement array* that characterizes irregular collectives. Take MPI_Gatherv as an example and suppose P processes are used, the address of the root' buffer for received messages is $recvbuf$, and the displacement array is $displs$. With MPI_Gatherv, one process, known as the root, gathers messages from all P processes and stores them in $recvbuf$ according to $displs$ — the i-th entry of $displs$ specifies the displacement relative to $recvbuf$ at which to place the incoming message from

978-1-5386-5555-9/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPSW.2018.00076

Table I
WELL-DOCUMENTED *scaling problems* REPORTED ONLINE [1], [8], [36], [3], [6], [7]. NOTES: (1) EFFECT - *H*ANG, *C*RASH AND PERFORMANCE *D*EGRADATION; (2) FAILING SCALE (P, M) - THE *P*ARALLELISM SCALE AND *M*ESSAGE SIZE THAT TRIGGER THE PROBLEM.

Prob.	Collective	MPI library	Type	Effect	Scale (P, M)	Root cause (inside MPI)
1	MPI_Gather	OpenMPI 1.4.3	3	H	(64, 4KB)	Environment setting dependency
2	MPI_Alltoall	OpenMPI 1.4.3	3	H	(44, 4MB)	Environment setting dependency
3	MPI_Allgather	OpenMPI 1.4.3	3	H	(64, 4MB)	—
4	MPI_Alltoallv	OpenMPI 1.7	3	H	(96, 512KB)	Network connection failure
5	MPI_Allgather	MPICH 2	3	D	$P \cdot M >$ INT_MAX	Integer overflow in MPI
6	MPI_Send + Recv	Intel MPI 5.1.2	2	H	(2, 64KB)	OS (ubuntu) dependency
7	MPI_Bcast	Intel MPI 5.1.2	2 or 3	H	(2, 64KB)	Unknown to developers
8	MPI_Bcast	Intel MPI 2017	2 or 3	H	(—, 16KB)	Platform (KNL & BDW) dependency

Table II
NEWLY UNCOVERED SCALING PROBLEMS.

Prob.	Collective	MPI library	Type	Effect	Scale (P, M)	Root cause
9	MPI_Gatherv(I)	MPI Standard	3	C	(48, 44MB)	Outside MPI
	MPI_Scatterv(I)		3	C/H		
	MPI_Allgatherv(I)		3	C		
	MPI_Alltoallv(I)		3	C		
10	MPI_Igather	OpenMPI 1.7 & 1.10	3	C	(48, 44MB)	Inside MPI
11	MPI_Iscatter		3	C/H		
12	MPI_Gather	MPICH 3.1.3	3	C	(48, 128MB)	
13	MPI_Scatter		3	C	(48, 44MB)	

Table III
WHO CAN FIX? MPI DEVELOPER (M), APP. DEVELOPER (D), APP. USER (U), OR OUR PROTECTION LAYER (P).

Prob.	M	D	U	P
1-4, 6-8	✓			✓
9		✓		✓
5, 10-13	✓			✓

process i $(0 \leq i < P)$, i.e., the starting address of the message from process i is

$$recvbuf + displs[i] * s, \tag{1}$$

where s denotes the size of the messages' data type. The maximum of a *int* type in C is denoted as INT_MAX. Since $displs[P-1] \leq$ INT_MAX, the number of elements that the root receives from the first $P-1$ processes must be no bigger than INT_MAX-1, which is about $1/(P-1)$ of the number of elements the root receives from the first $P-1$ processes when using MPI_Gather (INT_MAX $*(P-1)$). In addition, C int is represented with 32 bits on most current platforms [22]. When $P = 1024$, each process sending a few megabytes (2^{20} Bytes) can easily corrupt MPI_Gatherv's $displs$ as well as MPI_Gatherv. Hence irregular collectives face an urgent scalability issue that must be dealt with.

Scaling problems can result from a bug *inside* released MPI libraries due to the following two reasons. First, the lack of systematic testing over MPI software stack has caused scaling problems to go undetected – Type-2 problems triggered when operating on large messages have seen little test coverage [22] and the fact that Type-3 scaling problems manifest due to the combined force of parallelism scale and message size has not been adequately appreciated. Second, manifestation of some scaling problems is platform or environment dependent [1], [2], [3], [4], [6], [7] and completely removing them is extremely challenging. *Therefore, it is important for the library users, including both MPI application developers and the application users, to perform testing by themselves to detect potential scaling problems of MPI routines of their interest.*

B. Challenges

A scaling problem caused by breaking the aforementioned limits of irregular collectives can be fixed by MPI application developers via changing the application code so as to avoid corrupting the irregular collectives' displacement array. But application developers might not be willing to fix it when most often the application is used at small scale without breaking the limit. In addition, many — surely not all — application developers argue that MPI standard should replace all the uses of *C int* with *C long long int* to avoid the scaling problem due to integer overflow on MPI routines. However, it has been a struggle for MPI standard to make this replacement. The issue of *C int* has been discussed since at least 2011. However, MPI forum believes that developers can support large count by themselves, like by building big data types, and persists using *C int* till today to provide backward compatibility [9], [10].

For a scaling problem whose root cause is inside MPI, MPI library developers are responsible for fixing it, but it takes time to release an official fix and sometimes even not possible due to the difficulty of platform or environment dependent bugs' reproduction. Reproducing a scaling problem is challenging since some scaling problems are platform-dependent [3], [4], [6], [7] and some occur due to an incompatible environment setting [1], [2]. Because of these reasons some scaling problems might never be reproduced [6]. After a bug is reproduced, it can still take much time to issue a fix due to the difficulties of root-cause identification and the development of a safe fix [30].

C. Our Solution

To relieve the tension among *MPI developers*, *application developers* and *application users* as shown in Table III, this paper proposes *user-side testing* to uncover scaling problems and provides a framework that non-intrusively bypasses the uncovered problems. First, we eliminate the aforementioned limits of irregular collectives: based on interception, we check if the displacement array is corrupted, i.e., if it contains negative values, recover the value if a corruption occurs, and avoid the scaling problem via either (1) chopping the communication into smaller ones or (2) building big data types. Second, we bypass scaling problems inside the MPI collectives based on testing and the same avoidance strategies. We provide an automated testing tool set for MPI collectives that users can use to test the correctness of MPI routines of interest at large scale either when MPI is installed or when they suspect that some routines trigger scaling problems for applications built on them. If a scaling problem is detected for an MPI routine, the testing procedure reveals the problem trigger point, i.e. the parallelism scale or message size that triggers a scaling problem. When running an application, MPI routines are intercepted to dynamically check if the problem trigger is reached. If this is the case, our avoidance routine as discussed above is invoked to bypass the problem. The key contributions of this paper are:

- It makes a clear classification of scaling problems, and this classification leads to a useful observation — testing for Type-3 scaling problems does not necessarily require a large scale supercomputer if we exploit the interplay between message size and parallelism scale.
- It establishes the necessity of *user-side testing* to manifest scaling problems inside MPI collectives. We uncover two kinds of Type-3 scaling problems as shown in Table II: (1) an inherent defect in MPI standard on irregular collectives that impacts eight MPI routines; and (2) four hidden scaling problems inside the released MPI libraries including OpenMPI and MPICH.
- It provides a protection layer to avoid scaling problems without requiring any changes to the MPI library or user programs. It is an immediate remedy when an official fix is not readily available.
- It evaluates the practicality of our protection layer consisting of three potential avoidance strategies for four representative MPI collectives.

II. Overview

To affect a non-intrusive fix, we need the following: (1) *problem trigger* which is the scale at which a scaling problem manifests itself; and (2) *an avoidance* that alters the execution to avoid the problem. Once both of them are known we intercept an MPI Collective as shown below.

```
int MPI_Collective(...) {
  if (check_problem_trigger() )
      MPI_Collective_Avoidance(...);
    else
      PMPI_Collective(...);
}
```

The interception permits the default collective $PMPI\_Collective()$ only when a scaling problem's trigger is not reached; otherwise, it invokes the avoidance routine $MPI\_Collective\_Avoidance()$.

The trigger of the scaling problems caused by the corruption of the displacement array of irregular collectives is obvious: it is when at least one element in the array is negative (corrupted). To identify the triggers for other problems, we employ *testing*. As both Type-2 and Type-3 scaling problems relate to the message size, they can be triggered even on a small cluster by testing using large message sizes. Testing not only tells us if a scaling problem exists, it also identifies the scale that triggers the problem.

The avoidance we develop either (1) replaces the default large scale communication specified by multiple communications at a smaller scale or (2) exploits the interplay between *element count* and *data type size*, whose product equals the message size, by building a big data type. Without involving uncovered details, we just give an example of one strategy for the above approach (1). As shown in Table I, MPI_Gather (Prob. 1) breaks when the message size is 4KB when running with 64 processes. Suppose users use it at 8 KB message size with 64 processes. By testing we supposedly get the *maximum workable message size* like 3KB. Our avoidance bypasses it by carrying out two rounds of 3KB message transfers and one round of 2KB transfer.

Table IV
Notations.

Symbol	Meaning
n	Element count in one message
s	Size of the data type in bytes
P	Total number of processes
G_b	Global data buffer size in bytes
G_e	Global data buffer size in *element count*, $\frac{G_b}{s}$

Table V
MPI collectives and their global data buffer size. If (I) follows a collective, the collective has a non-blocking variation; if v follows a collective, the collective has an irregular variation.

Type	Function	G_b
One-to-All	MPI_Bcast(I)	sn
	MPI_Scatter(I, v)	snP
All-to-All	MPI_Allgather(I, v)	snP
	MPI_Allreduce(I)	sn
	MPI_Alltoall(I, v)	snP
	MPI_Reduce_scatter	sn
All-to-One	MPI_Gather(I, v)	snP
	MPI_Reduce(I)	sn

III. Manifesting Scaling Problems

A. Basics of MPI Collectives

Table IV lists the notations we use. With a collective, P processes communicate with each message having n elements whose data type's size is s. MPI collectives can be classified into four types: *All-to-All*, *All-to-One*, *One-to-All*, and *other collectives* that do not fit into any type above [32]. Table V lists the collectives considered in this paper, which covers both the blocking/non-blocking regular collectives and blocking/non-blocking irregular collectives. **Root process** is the process holding the final result for All-to-One collectives and the one holding the data sent out to all processes for One-to-All collectives. No root exists in symmetrical All-to-All collectives.

Global data buffer stands for the data buffer whose contents are either contributed by or distributed to all processes. The global data buffer is the root's receiving buffer for All-

Table VI
EXPERIMENT SETUP.

Platform	• Tianhe-2, each node having 2 E5-2692 processors (24 cores) and 64GB memory
MPI	• MPICH 3.1.3 based on InfiniBand • OpenMPI 1.7 & 1.10.0 based on TCP/IP
Programs	• OMB adapted for automated testing

to-One and the root's sending buffer for One-to-All. Each data buffer for All-to-All is a global data buffer, but we refer to the largest data buffer when discussing its size. We denote the global data buffer size in *bytes* as G_b and in terms of *element count* as G_e. G_b can be expressed as functions of s, n, and P (see Table V), and $G_e = G_b/s$.

B. Testing

Experiment Setup. Table VI provides an overview of our experiment setup. The MPI libraries we study include MPICH 3.1.3, OpenMPI 1.7, and OpenMPI 1.10.0. MPICH 3.1.3 runs over of TH-express—a specialized high performance network interconnect of Tianhe-2 [31]. OpenMPI on the other hand is run by using TCP/IP over TH-express, which can be achieved by assigning *btl* framework to *tcp* [21]. We modified collective benchmark set from the OSU micro-benchmark suite (OMB) [11] so that it enables us to set a time limit for the test at each message size and to vary n and s.

Testing Scheme. We perform testing by scaling both parallelism and message size. To increase parallelism P, we increment the number of nodes while allocating 24 processes per node (1 process per core). To increase message size sn we increase n while fixing s – for MPI_Reduce, MPI_Reduce_scatter, and MPI_Allreduce, $s = 4B$ as data type MPI_FLOAT is used and for the rest $s = 1B$ as MPI_CHAR is used. We perform testing for $P = 48$ and 96. Given P, *the testing is fully automated via a Linux shell script* that submit time-limited tests (jobs) to job scheduler — each test is denoted as $test(n, t)$, where t stands for the time limit requested to run the job. If a test crashes or cannot finish in time t, a failure is reported. Testing steps are:

– *Step 1* iterates until (1) the message size grows to INT_MAX, the maximum allowed by its data type, (2) memory limit is hit [1] or (3) a failure is encountered. This process starts from $n = 1$ with $t = 60$ seconds as it is far more than enough to complete a transfer of 1 or 4 bytes with 60 seconds for any collective in our configuration. If it succeeds, we update t as the real time cost of the current run. We continue tests by increasing message size via $n \leftarrow 2*n$ as well as sufficiently increasing the time limit via $t = 10*t$. In this step, the testing procedure terminates without finding any scaling problem if condition (1) is met; the testing with the next P configuration starts if condition (2) is met; and the testing proceeds to *Step 2* upon condition (3) with the detected largest n that passes the test, denoted as n'_s.

[1]During execution if a test runs out of memory due to the huge memory footprint, we can identify this error from the error logs showing some processes being killed by the kernel, or more specifically by OOM killer.

Table VII
SAFE BOUNDS.

Root cause	Prob.	Safe bound n_s (Δ)	
		$P = 48$	$P = 96$
Outside MPI	9	42M (2M)	21M (1M)
Inside MPI	10-11, 13	42M (2M)	21M (1M)
Inside MPI	12	124M (4M)	62M (2M)

– *Step 2* refines n'_s found in Step 1 as follows. We know n'_s succeeds and $2n'_s$ fails, so we test at interval $\Delta = n'_s/f$ (we use $f = 16$ in our testing and users can vary f to configure Δ to satisfy their requirement) in the range $[n'_s + \Delta, 2n'_s)$. Finally the largest n that passes the test at interval Δ is found. The **safe bound**, n_s, is the largest n that passes the test under our testing scheme for the given s and P, i.e., the test is able to pass if $n \leq n_s$ but it fails if $n > n_s + \Delta$.

C. Scaling Problems Uncovered

Using the above testing scheme we uncovered scaling problems shown in Table II. These scaling problems can result from (1) *Displacement array corruption outside MPI library* that impacts all 8 irregular collectives from any MPI library and (2) *A corruption inside MPI library*, which maps to 4 corrupted functions in various released MPI libraries.

Outside MPI (Prob. 9). In the default setting that allocates 24 processes per node, all irregular collectives except MPI_Alltoallv(I) are found to be susceptible to this problem, and MPI_Alltoallv(I) are not as it hits the memory limit first due to its higher memory consumption. The scaling problem occurs with the use of MPI_Alltoallv(I) when we reduce the memory consumption by allocating one process per node. These scaling problems are invariably caused by an integer overflow error when calculating the *C int* displacement array for irregular collectives. On the other hand, even this error does not occur in user code, i.e., users calculate correctly based on a larger data type like *C long long int*, the scaling problems would still occur due to *truncation error* in the data type conversion.

Inside MPI (Prob. 10, 11, 12, 13). Table II shows two collectives — each in both OpenMPI 1.7 and 1.10.0 and MPICH 3.1.3 — encounter a scaling problem due to an integer overflow inside the MPI library. Next we illustrate this problem using MPI_Igather from OpenMPI 1.10. In MPI_Igather's underlying function *ompi_coll_libnbc_igather*, the root process needs to calculate the starting address $rbuf$ for storing the message from process i with

$$rbuf = (\text{char }*)recvbuf + i * recvcount * rcvext, \quad (2)$$

where $recvbuf$ is the starting address of the root's receiving buffer, $recvcount$ (*C int*) is the number of elements in one message, and $recvext$ is the size of the used data type. Integer overflow occurs when $i * recvcount >$ INT_MAX, which results in a negative integer as well as an invalid address assigned to $rbuf$. Considering there are P processes in total, the problem is triggered once $n(P-1) \geq$ INT_MAX.

Table VIII
SCALING PROBLEM DETECTORS.

Class	Detector
D	$displs[i_0] < 0$
G	$G_b > B_h$ or $G_e > B_h$
X	$n > B_h$ given s and P
• $displs$, the displacement array for an irregular collective • i_0, the index of the first corrupted element • B_h, a bound restricted by an unknown scaling problems	

Safe bound. For each MPI routine having a scaling problem, we report the safe bound n_s, where test is able to pass if $n \leq n_s$ but it fails if $n > n_s + \Delta$. The safe bound for each scaling problem is reported in Table VII.

Useful insights have been gained based on the problems reported online as well as new problems detected by us. First, manifesting a Type-3 scaling problems does not necessarily demand a supercomputer and many scaling problems can be found by interplaying the message size and parallelism scale. Second, the testing coverage of MPI software stack is inadequate as shown by newly uncovered problems and scaling problems resulting from platform and environment dependency are hard be removed. Third, it takes time to obtain an official fix and sometimes the fix is not possible considering the platform-dependent scaling problems that are hard to be reproduced [6] as well as the displacement array corruption for irregular collectives. All these inspires us to propose user-side testing and to provide an easy-to-use protection layer, which acts as an immediate remedy when an official fix is not available.

IV. ONLINE PROBLEM DETECTORS

Depending upon the difficulty of *detection*, we classify the problems into 3 classes as shown in Table VIII: (1) Class *D* caused by displacement array corruption, (2) Class *G* triggered when the global data buffer is too big, and (3) Class *X* whose trigger form is not known.

A. Class D: Displacement Array Corruption

To detect the occurrence of a scaling problem (e.g., Prob. 9) because of displacement array $displs$ corruption is very straightforward. One pass over the array is enough. Below we detail the validity of our assumptions and how we detect the corruption as well as how $displs$ can be recovered [2].

Assumptions. We assume (1) uncorrupted values in array $displs$ are non-descending; (2) $n \leq$ INT_MAX; and (3) *two's complement* is used to represent integers. Assumption (1), though not specified by MPI standard, is based on a commonly used programming convention of organizing the data in global data buffer by MPI rank. Using this convention makes programming less error-prone. Assumption (2) implies that the number of elements sent by each process is at most INT_MAX, which is specified by the standard. Assumption (3) is true on nearly all modern machines [13].

[2]Proofs omitted due to space limitation – will be provided upon request.

Table IX
DETECTOR G'S LOOKUP TABLE.

1	2	3	Detector	Type
(s_s, P_s)	$(s_s, 2P_s)$	$(2s_s, P_s)$		
n_s	$n_s/2$	$n_s/2$	$snP > s_s n_s P_s$	3
	$n_s/2$	n_s	$nP > n_s P_s$	
	n_s	$n_s/2$	$sn > s_s n_s$	2
	n_s	n_s	$n > n_s$	

Detector D. A corruption is detected if $displs[i_0] < 0$, where $0 \leq i_0 \leq P-1$ and the i_o^{th} entry is the first element being corrupted.

Recovery. Suppose the *a*ctual values of array $displs$ are a_0, a_1, a_2, ..., a_{P-1} and the supposed *c*orrect values are c_0, c_1, c_2, ..., c_{P-1}. Upon two's complement system, we have:

$$a_i = c_i \% (2 \text{ INT\_MAX} + 2) - (2 \text{ INT\_MAX} + 2). \quad (3)$$

This implies that for a corrupted array the actual values will have several *segment*s, where the actual values are sorted increasingly in the range of $[-\text{INT\_MAX} - 1, \text{INT\_MAX}]$. We can always recover $displs$ based on the corrupted values as below: (1) if $i = 0$,

$$c_i = a_0 \ ; \quad (4)$$

(2) else if $i > 0$ and $a_i \geq a_{i-1}$

$$c_i = c_{i-1} + a_i - a_{i-1} \ ; \quad (5)$$

and (3) else

$$c_i = c_i + a_i + 2\text{INT\_MAX} + 2 - a_{i-1} \ ; \quad (6)$$

B. Class G: Global Data Buffer Too Large

This class of scaling problem manifests when the global data buffer size exceeds a certain *bound* B_h. For example, Prob. 5, 10, 11, and 13 fall in this class.

Detector G: $G_b > B_h$ or $G_e > B_h$, i.e., the *global data buffer size*, evaluated in either *bytes* or *elements*, exceeds a bound caused by an unknown scaling problem.

This problem trigger is built based upon the analysis of certain scaling problems – Prob. 5, 10 and 11. Prob. 5 is a Type-3 scaling problem that was found in MPI_Allgather in MPICH2 [36]. It was tracked down to an integer overflow that caused a non-optimal communication algorithm to be selected and this leads to serious performance degradation. Its problem trigger relation is

$$snP > \text{INT\_MAX}. \quad (7)$$

The triggers of Prob. 10 and 11 can be expressed as

$$n(P-1) \approx nP > \text{INT\_MAX}, \quad (8)$$

where $P \gg 1$. *All these problem triggers represent cases where the global data buffer size exceeds a certain bound.*

Based on the global data buffer size, we can classify MPI collectives into two types: (1) collectives with $G_b = snP$ including MPI_Alltoall(I,v), MPI_Allgather(I,v), MPI_Gather(I,v), and MPI_Scatter(I,v), whose trigger (*Type-3*) can be expressed as

$$nP > B_h, \text{or } snP > B_h; \quad (9)$$

and collectives with $G_b = sn$ including MPI_Allreduce(I), MPI_Reduce(I), MPI_Reduce_scatter(I) and MPI_Bcast(I), whose triggers (*Type-2*) are

$$n > B_h, \text{or } sn > B_h. \quad (10)$$

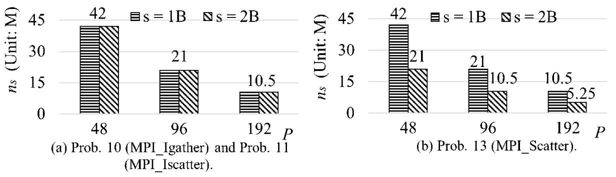

Figure 1. Safe bounds of G problems (Prob. 10, 11 and 13).

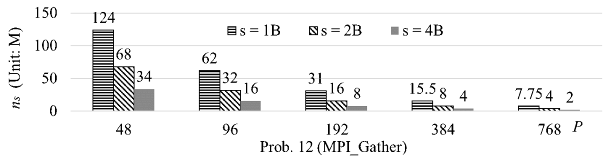

Figure 2. Safe bounds of an X problem (Prob. 12).

Identifying class G and its detector. Based on one round of testing, we can get the safe bound n_s given (s_s, P_s, Δ). To tell whether such scaling problem is from Class G, we simply check two additional safe bounds at different (s, P) configurations by varying each parameter – $(2s_s, P_s)$ and $(s_s, 2P_s)$ as given in Table IX. To avoid unnecessary brute-force stress testing on finding these two additional safe bounds, we verify if the safe bound is $n_s/2$ or n_s. If the test passes at $n_s/2$ and fails at $(n_s + \Delta)/2$, the safe bound is $n_s/2$; otherwise, we continue checking n_s: if the test passes at n_s while failing at $n_s + \Delta$, the safe bound is verified to be n_s. If all the safe bounds match any row of Table IX, we claim this problem is from Class G and its detector is given in the fourth column. Otherwise, it falls in class X as discussed below.

C. Class X: Trigger Form Not General

Class X represents scaling problems that cannot be quantitatively expressed using a general form. Although Prob. 1, 2, 3, 4, 7 were reported, we cannot conclude that its trigger can be expressed in a general form like for Class G and thus we capture them in a restrictive condition.

Detector X: $n > B_h$ given s and P. Note it is not a general method; it works only within the restriction.

D. Case Studies: Class G and X

Detector D is sound enough based on proof. Here we show how to find detectors for Class G and Class X.

Class G. To check if a scaling problem is of Class G or not, we first *find* the safe bound at $(s = 1\text{B}, P = 48)$, and then *verify* the two safe bounds at $(s = 1\text{B}, P = 96)$ and $(s = 2\text{B}, P = 48)$. The detected safe bounds of Prob. 10, 11 and 13 are shown in Figure 1, where the required three as well as three additional safe bounds are shown so as to provide a clear picture of how the safe bounds vary given different (s, P) settings. By checking Table IX, we conclude that these problems are from Class G. However, their detectors are different. For Prob. 10 and 11, the detectors are the same:

$$G_e = nP > 2016\text{M}. \tag{11}$$

Prob. 13's detector is:

$$G_b = snP > 2016\text{MB}. \tag{12}$$

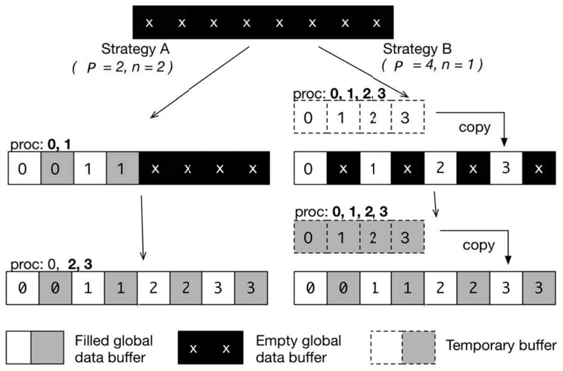

Figure 3. Illustration of the partitioning strategies for MPI_Gatherv ($P = 4$ and $n = 2$) by breaking down the filling process of the global data buffer. Process 0 is the root and the bug would be triggered when $nP > 4$.

Class X. Similarly we found that the three safe bounds at $(s = 1\text{B}, P = 48)$, $(s = 2\text{B}, P = 48)$ and $(s = 1\text{B}, P = 96)$ are 124M, 68M and 62M respectively as shown in Figure 2. However, these do not map to any row in Table IX and thus we classify this problem into Class X. Based on Figure 2, it follows:

$$\begin{cases} snP > 5952\text{MB} & \text{if } s = 1 \text{ and } P \geq 48 \\ snP > 6144\text{MB} & \text{if } s \geq 2 \text{ and } P \geq 96 \end{cases} \tag{13}$$

Note that users do not necessarily need to find the exact bound in all situations. Easily users can find a bound though overly restrictive. For example, an application uses the buggy MPI_Gather at $s = 1\text{B}$ and $P = 96$. Based on testing it is easy to obtain 62M as the safe bound. Thus, we assume that the problem can occur if $n > 62\text{M}$.

V. Non-intrusive Avoidance

To avoid the risk of introducing other scaling problems, we keep our design clean via following protocols: (1) the avoidance of an MPI routine's scaling problem is based on the routine itself; (2) the avoidance uses the minimal number of MPI routines other than the target routine, i.e. other routines at most do some control messages' passing involving only a few bytes. For example, though avoiding MPI_Gather with MPI_Gatherv is easy, it it not allowed to avoid any problems existing in MPI_Gatherv.

A. Workaround 1: Communication Partitioning

Partitioning strategies. Consider a Type-3 scaling problem of Class G that manifests when $G_b > B_h$ (or $G_e > B_h$). An inherent workaround (`W1`) is to partition the communication such that for each sub-communication $nP \leq B_h$ (or $G_e \leq B_h$). Specifically, `W1` has two partitioning strategies: (`A`) shrink P while fixing n; and (`B`) shrink n while fixing P. Consider MPI_Gatherv, for which the scaling problem is triggered when $nP > 4$. Figure 3 shows a *simplified view* of how the two strategies are applied. `W1-A` creates two process groups – $\{0, 1\}$ and $\{0, 2, 3\}$. Since process 0 is the root that receives messages from all, it is present in every group. In the 2^{nd} group process 0 can be configured to send out nothing; thus the real number of processes participating in each sub-communication is still two, i.e., $P = 2$ and $n = 2$.

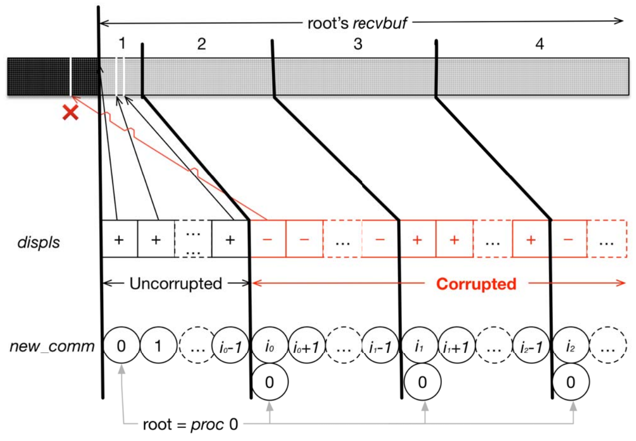

Figure 4. Workaround 1-A for MPI_Gatherv.

With `W1-B`, $P = 4$ and $n = 1$ in each sub-communication. Since $nP = 4$ with either strategy, the scaling problem is avoided.

Applying Workaround 1-A to MPI_Gatherv: Applying the workarounds to Class G and X is straightforward, but it involves the tricky issue of displacement array's corruption for Class D. We hence illustrate how `W1-A` works for MPI_Gatherv (Class D) here. As shown in Figure 4, one corrupted displacement array $displs$ consists of at least two *segments* with all elements in each segment are either non-negative or negative. Each such segment maps to one segment of root process' $recvbuf$ as well as a group of processes, in which the root process should be added if it is not included as it is the one that holds $recvbuf$. The communication then could be naturally partitioned, each of which is performed within one group of processes.

Constructing uncorrupted displacement array. Recall that we have recovered $displs$ from corruption with all of its correct values stored in array c. For each sub-communication, we construct a new displacement array ($disps2$) as

$$displs2[i] = \begin{cases} 0 & \text{if } i = 0, \\ c_{s_0+i} - c_{s_0} & \text{if } i > 0, \end{cases} \qquad (14)$$

where the range $[s_0, s_1 - 1]$ depicts the process id range of one process group. In one run, we have $c_{s_1-1} - c_{s_0} \leq$ INT_MAX and thus $displs2$ would not be corrupted.

B. Workaround 2: Big Data Type

Building a big data type is a potential alternative strategy (`W2`) for scaling problems that are *unrelated to data type size* s such as Prob. 8, 9, and 10. With the newly created big data type of size d-bytes, an original message having x elements with each element accounting for y bytes can be converted to a new message containing xy/d elements with each element having d-bytes. That is, the number of elements in one message (n) is decreased by a factor of d. Suppose the safe bound limit for an s-irrelevant scaling problem is n_s. This could increase the safe bound from n_s to dn_s. In addition, `W2`'s performance is expected to be comparable to the original's as the cost of building new data type is trivial.

Table X
WORKAROUNDS APPLICABILITY: "✓" - APPLY; "✗" - DOES NOT APPLY; "✓✗" - APPLY WITH RESTRICTIONS.

Scaling problems		W1-A	W1-B	W2
Type-3	Class D	✓✗	✓	✓✗
	Class G	✓✗		
	Class X	✗		
Type-2		✗		

The size of big data type in byte (d) can be set as following: (1) $d = sn_s$ for *regular* collectives; and (2) $d = sn_{gcd}$ with n_{gcd} being the *greatest common divisor* of all values in $displs$ and $recvcounts$ for *irregular* collectives, which ensures that using the new data type the collective is able to work as intended. Note that for case (2) `W2` is effective only when $n_{gcd} > 1$.

C. Applicability and Limitation

Table X summarizes the applicability of all strategies on various scaling problems. `W1-A` can tackle the majority of scaling problems of Class D and G from Type-3, its restriction is for MPI_Alltoall(I,v) as this routine has the highest communication complexity and partitioning the parallelism scale will only lead to complex error-prone logic. It cannot handle Class X as we use the detector $n > B_h$. It does not handle Type-2 as only message size matters for this type. `W1-B` that cuts message size is the most general avoidance that applies unconditionally. `W2` is less general compared with `W1-B` because of following limitations: (1) it only works for scaling problems that are unrelated to s; and (2) it does not work for irregular collectives when $n_{gcd} = 1$.

`W2`'s limitation resides in its limited applicability. `W1` has limitations as well. First, non-blocking communication routines has been turned into its blocking communication using `W1-A` and `W1-B`. Second, additional memory overhead is incurred in the implementation of some workarounds like `W1-B` for MPI_Gather. Third, the performance of `W1-A` and `W1-B` is not as good as the performance of `W2`.

D. Evaluation

We evaluate our non-intrusive workarounds based on 4 representative MPI routines. They stand for *all-to-one* and *all-to-all* — *one-to-all* is ignored as it is very similar to *all-to-one*, and also they represent *irregular*, *regular*, *blocking* and *non-blocking* collectives. Our default setting is the same as mentioned earlier—24 processes per node (1 process per core), $s = 1$B and $f = 16$.

1) ***Effectiveness:*** The effectiveness is evaluated by the degree to which a workaround can increase the *safe bounds* of the default buggy functions – the greater the safe bounds are increased the more effective is the workaround. In the evaluation, our workarounds increase the safe bounds significantly, but the workarounds' safe bounds are limited by the physical memory size. To show this point, we also report the *maximum memory consumption on one node*,

Table XI
WORKAROUNDS' EFFECTIVENESS FOR MPI_GATHERV (D) AND MPI_IGATHER (G). THE UNIT OF n_s IS 1 M, I.E. 2^{20}, AND THAT OF R_M IS GB.

Scale ↓		MPI_Gatherv								MPI_Igather							
		Original		W1-A		W1-B		W2		Original		W2		W1-B		W2	
		n_s	R_M	n_s	R_M	n_s	R_M	n_s	R_M	n_s	R_M	n_s	R_M	n_s	R_M	n_s	R_M
P	192	10.5	2.21	256	54.00	256	54.0	272	57.38	10.5	2.21	256	54.00	256	54.00	272	57.38
	768	2.625	2.03	68	52.60	72	55.69	72	55.69	2.625	2.03	72	57.02	42	32.48	42	32.48

Table XII
EFFECTIVENESS OF WORKAROUND 1-B FOR MPI_GATHER (X).

Scale ↓		Original		W1-B	
		n_s	R_M	n_s	R_M
P	192	31	6.75	240	50.63
	768	7.75	6.19	64	49.50

denoted as R_M, which is calculated according to the MPI standard.

I. Class *D* (Prob. 9: MPI_Gatherv). As shown in Table XI, all three workarounds have comparable effectiveness, and their safe bounds are roughly 24 times the safe bound of the default MPI_Gatherv. `W1` and `W2` have comparable effectiveness. The workarounds do not go further because the physical memory limit is reached — note MPI has hidden memory footprint besides the obvious R_M.

II. Class *G* (Prob. 10: MPI_Igather). Its evaluation is shown in Table XI. At scale $P = 192$, three workarounds are of comparable effectiveness and their safe bounds are 24+ times the default MPI_Igather's safe bound. At $P = 768$, `W1-A` is the best, `W2` and and `W1-B` are worse, where the first is limited by the memory size while the last two are not. The last two's worse performance is traced down to the error of *connection time out*, i.e., when too many processes connect to the root process that is only capable of responding a portion of the connection requests at a time due to the ongoing communication with large message sizes, some connections fail to be established within a time limit. This error doesn't negatively impact `W1-A` as each time it communicates with only a small portion of processes.

III. Class *X* (Prob. 12: MPI_Gather). The detection only works under a specific restriction as mentioned earlier. Here the restrictions are $s = 1$ and $P \geq 48$ and the scaling problem manifests when $n \geq \frac{5952}{P}$. `W1-B` is the only workable solution. Table XII shows `W1-B`'s safe bounds are 7+ times of the default's. As `W1-B` incurs 5.8 GB memory overhead, R_M is smaller compared with the previous experiments.

*2) **Performance:*** The performance is measured as *time cost* in seconds. As each process might have varying time costs in one run, we report both the average and the maximum. Given a (s, n, P) configuration, a collective is run Y times, where $Y = 500$ if $n <= 1M$ and $Y = 20$ otherwise. We evaluate the performance of workarounds using the *above three buggy collectives* first and then two correctly-functioning routines with *supposed* scaling problems for which all the workarounds apply.

I. Class *D*. For MPI_Gatherv, all workarounds are effective; they detect scaling problems of class D by detecting if the displacement array $displs$ is corrupted, which

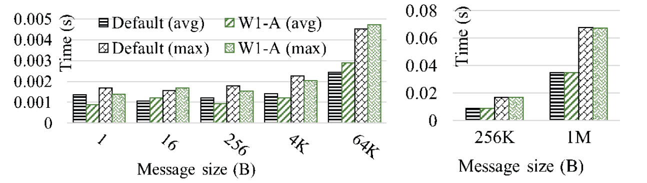

Figure 5. Performance comparison between W1-A and the default MPI_Gatherv (MPICH) before the scaling problem's occurrence.

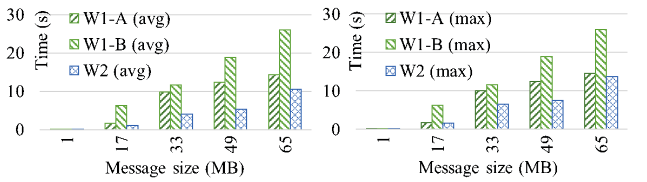

Figure 6. Performance comparison among the three workarounds for MPI_Gatherv (MPICH) whose scaling problem (Class D) is triggered once $sn > 2.625MB$ when $P = 768$.

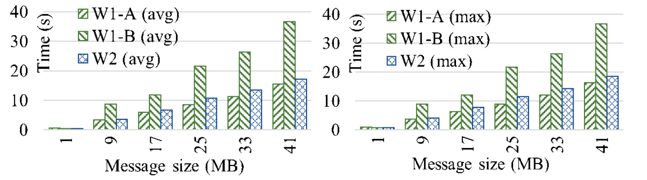

Figure 7. Performance comparison among the three workarounds for MPI_Igather (OpenMPI) whose scaling problem (Class G) is triggered once $sn > 2.625$MB when $P = 768$.

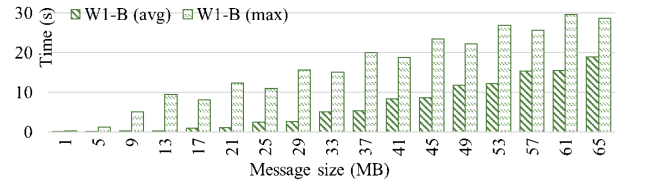

Figure 8. Performance trend of W1-B for MPI_Gather (MPICH) whose scaling problem (Class X) is triggered once $sn > 7.75$MB when $P = 768$.

involves checking all elements in $displs$ and broadcasting the judgment to all P processes with respectively $O(P)$ and $O(logP)$ time complexity. Considering such overhead, we evaluate their performance both before and after the problem's occurrence. Figure 5 shows the comparison between the default MPI_Gatherv and `W1-A` before the problem occurs. Note all workarounds detect the scaling problem in the same way and thus have the same detection overhead before the problem's occurrence, so we only evaluate `W1-A`. We observe that the performance of `W1-A` is comparable to the default as the detection overhead is trivial. Figure 6 shows that the time costs of `W1-A` and `W1-B` grow linearly with message size as it cuts communication into roughly equal-sized pieces. `W2`'s performance is better as it retains the communication only by varying the parameter setting, but it should be noted it is not guaranteed to work for

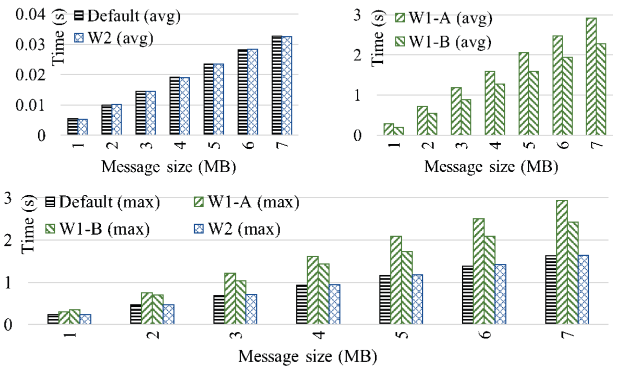

Figure 9. Performance comparison based on MPI_Gather (MPICH) supposing a Class G problem is triggered when $n > 128\text{K}$ at $P = 768$.

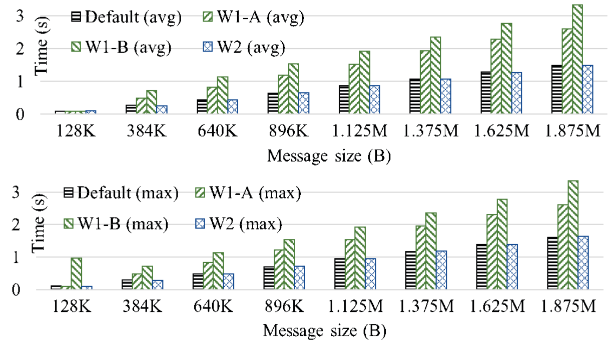

Figure 10. Performance comparison based on MPI_Allgatherv (MPICH) supposing a Class G problem is triggered when $n > 128\text{K}$ at $P = 768$.

irregular collectives as explained previously. To make W2 work in this case, we configure all processes transfer equal-sized messages in the experiment.

II. Class *G*. Since the problem detection is only based on checking an inequality which is far less overhead than the detection overhead discussed above, we only measure the performance after the problem's appearance. Figure 7 shows the performance comparison of W1-A, W1-B, and W2 for MPI_Igather. W1-A is better than W2 by only a small margin, and W1-B's performance is about half of W2's.

III. Class *X*. As the detection is also trivial, the performance is measured only after the problem's appearance, which is shown in Figure 8.

IV. Evaluation of all workarounds based on correct MPI_Gather. To make sure *all workarounds apply*, we suppose a class G scaling problem would be triggered if $nP > 96\text{M}$. Figure 9 shows the performance comparison among all and the default. It is observed that: (1) For the default and W2 the maximum time cost is about 50 times the average, but for W1-A and W1-B the maximum is at most 1.2 times the average, which results from the fact that the partitioning method of W1-A and W1-B delays all the non-root processes while W2 does not; (2) W2 is of comparable performance to the default; (3) Based on the maximum, W1-A's time cost is 1.8 times the default's and W1-B's is 1.5 times the default's.

V. Evaluation of all workarounds based on correct MPI_AllGatherv. For the same reason, we assume a class G scaling problem would be triggered if $nP > 96M$. Figure 10 shows the performance comparison. We have following observations: (1) the average and the maximum has little difference as the communication is symmetrical, i.e., all the processes transfer the same amount of data; (2) W2 and the default have comparable performance; (3) W1-A and W1-B respectively demands about 1.7 and 2.2 times the time cost of the default.

Summary. Before a scaling problem's occurrence, the performance of any workaround is comparable to that of the default. After its occurrence, W2's performance is comparable to the default's. W1-A's and W1-B's performance are worse because they partition the default communication, and their time cost increases linearly as the message size goes up. In conclusion, W1-B is the a general solution, and W2 has the best performance.

VI. Related Work

General bug detection. Though MPI has been the de facto standard for message passing, the difficulty of using MPI remains a challenge due to various programming errors and library errors. Many works [18], [34], [25], [20], [23], [24] have focused on detecting programming errors like resource errors, parameter errors and deadlocks. There are also quite a few works [19], [16], [17] that detect defects such as data movement error and synchronization error inside MPI library. All these focus on specific errors that are not necessarily related to running scale or problem size. Our work instead studies scaling problems. Testing has been used to uncover non-deterministic bugs [35] and program logic errors [15]. We further the study of testing by applying it on scaling problems.

Scaling problems. Much progress has been made on detecting and diagnosing scaling problems. Zhou et al. [36], [37] predict the happening of scaling problems based on a model built from bug-free runs at small scale. To aid the diagnosing of hangs and deadlocks, many works studied how to automatically identify the root-cause process of scaling problems [12], [28], [27], [29] as well as how to efficiently detect hangs at runtime [14]. These focus on only one step of the whole debugging process, i.e., the root cause identification at the process level. Laguna et al. [38] identify the scale-dependent integer overflow bugs at code-level in large-scale parallel applications using both static and dynamic analysis so as to help programmers fix the bug. All of the above are complementary to our work. Our work is different from them as it aims to find an integrated solution that detects and bypasses scaling problems of the MPI libraries without the necessity to locate the root cause. Hammond et al. [22] extends MPI to support the need of sending a message having a large element count that exceeds INT_MAX based on building big data types. Our work used the idea of big data types as one of our approaches to solve a different problem: (1) the element count does not disobey the MPI standard, i.e. it is smaller than INT_MAX; and (2) we provide non-intrusive workarounds for more than integer overflow, e.g. the workarounds can also work for environment-dependent scaling problem.

VII. Conclusion

We demonstrate the necessity of user-side testing. We show that testing with limited computing resources can manifest scaling problems based on the interplay between message size and parallelism scale. We provide a protection layer consisting of three potential avoidance strategies and evaluate its practicality based on representative MPI routines. Our strategies can also be easily applied to point-to-point communication.

References

[1] OMPI 1.4.3 hangs in IMB Gather. https://github.com/open-mpi/ompi/issues/125 (Retrieved: 2016).

[2] OMPI v1.6 running out of registered memory. https://svn.open-mpi.org/trac/ompi/ticket/3134(Retrieved: 2016).

[3] MPI code hangs when send/recv large data. https://software.intel.com/en-us/forums/intel-clusters-and-hpc-technology/topic/610561 (Retrieved: 2016).

[4] MPI has bad performance in user mode. https://software.intel.com/en-us/forums/intel-clusters-and-hpc-technology/topic/607259 (Retrieved: 2016).

[5] MPI_Get_count and large messages. http://trac.mpich.org/projects/mpich/ticket/1005 (Retrieved: 2016).

[6] Code hangs when variables value increases. https://software.intel.com/en-us/forums/intel-clusters-and-hpc-technology/topic/601182 (Retrieved: 2016).

[7] MPI_Bcast in Intel MPI Library 2017 Hangs on Large User-Defined Datatypes. https://software.intel.com/en-us/articles/intel-mpi-library-2017-known-issue-mpi-bcast-hang-on-large- user-defined-datatypes (Retrieved: 2017).

[8] OMPI-1.7 MPI_Alltoallv hangs. https://github.com/open-mpi/ompi/issues/1620 (Retrieved: 2016).

[9] Can we count on MPI to handle large datasets? http://blogs.cisco.com/performance/can-we-count-on-mpi-to-handle-large-datasets (Retrieved: 2016).

[10] MPI: A Message-Passing Interface Standard Version 3.1. http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[11] OMB. http://mvapich.cse.ohio-state.edu/benchmarks.

[12] D.H. Ahn, B.R. De Supinski, I. Laguna, G.L. Lee, B. Liblit, B.P. Miller, and M. Schulz. Scalable temporal order analysis for large scale debugging. In *SC*, pp. 1–11, 2009.

[13] R.E. Bryant and D.R. O'Hallaron. *Computer systems: a programmer's perspective*. Addison-Wesley, 2010.

[14] H. Li, Z. Chen, and R. Gupta. Parastack: Efficient hang detection for MPI programs at large scale. In *ACM/IEEE SC*, Article No. 63, 2017.

[15] H. Li, S. Li, Z. Benavides, Z. Chen, and R. Gupta. COMPI: Concolic testing for MPI applications. In *IEEE IPDPS*, 2018.

[16] Z. Chen, J. Dinan, Z. Tang, P. Balaji, H. Zhong, J. Wei, T. Huang, and F. Qin. MC-Checker: Detecting memory consistency errors in MPI one-sided applications. In *ACM/IEEE SC*, pp. 499–510, 2014.

[17] Z. Chen, X. Li, J-Y. Chen, H. Zhong, and F. Qin. Syncchecker: Detecting synchronization errors between MPI applications and libraries. In *IPDPS*, pp. 342–353, 2012.

[18] J. Coyle, J. Hoekstra, G. R. Luecke, Y. Zou, and M. Kraeva. Deadlock detection in MPI programs. *Conc. and Computation: Practice & Experience*, 14(11), 2002.

[19] Q. Gao, F. Qin, and D. K. Panda. DMTracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements. In SC, Article No. 15, 2007.

[20] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with Intel message checker. In *SE-HPCS Workshop*, pp. 78–82, 2005.

[21] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2004.

[22] J. R. Hammond, A. Schafer, and R. Latham. To INT_MAX... and beyond! Exploring large-count support in MPI. In *Workshop on Exascale MPI at Supercomputing Conf.* ,pp. 1–8, 2014.

[23] T. Hilbrich, B. R. de Supinski, Martin Schulz, and Matthias S. Müller. A graph based approach for MPI deadlock detection. In SC, pp. 296–305, 2009.

[24] T. Hilbrich, B. R. de Supinski, W. E. Nagel, J. Protze, C. Baier, and M. S. Müller. Distributed wait state tracking for runtime MPI deadlock detection. In *SC*, 2013.

[25] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. Marmot: An MPI analysis and checking tool. In *PARCO*, pp. 493–500, 2003.

[26] I. Laguna, D. H. Ahn, B. R. de Supinski, T. Gamblin, G. L. Lee, M. Schulz, S. Bagchi, M. Kulkarni, B. Zhou, Z. Chen, et al. Debugging high-performance computing applications at massive scales. *CACM*, 58(9), 2015.

[27] I. Laguna, D. H. Ahn, B. R. de Supinski, S. Bagchi, and T. Gamblin. Diagnosis of performance faults in large scale MPI applications via probabilistic progress- dependence inference. In TPDS, 26(5), 2015.

[28] I. Laguna, D. H. Ahn, B. R. de Supinski, S. Bagchi, and T. Gamblin. Probabilistic diagnosis of performance faults in large-scale parallel applications. In *PACT*, pp. 213–222, 2012.

[29] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin. Accurate application progress analysis for large-scale parallel debugging. In *PLDI*, 2014.

[30] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, pp. 902–912, 2015.

[31] Z. Pang, M. Xie, J. Zhang, Y. Zheng, G. Wang, D. Dong, and G. Suo. The TH express high performance interconnect networks. *Frontiers of CS*, 8(3), 2014.

[32] A. Skjellum, N. E. Doss, and K. Viswanathan. Intercommunicator extensions to MPI in the MPIX (MPI extension) library. Tech. report, Citeseer, 1994.

[33] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, et al. Addressing failures in Exascale computing. *IJHPCA*, 2014.

[34] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *SC*, Article No. 51, 2000.

[35] R. Vuduc, M. Schulz, D. Quinlan, B. De Supinski, and A. Sæbjørnsen. Improving distributed memory applications testing by message perturbation. In *Workshop on Parallel and Distributed Systems: Testing and Debugging*, pp. 27–36, 2006.

[36] B. Zhou, M. Kulkarni, and S. Bagchi. Vrisha: Using scaling properties of parallel programs for bug detection and localization. In *HPDC*, 2011.

[37] B. Zhou, J. Too, M. Kulkarni, and S. Bagchi. Wukong: automatically detecting and localizing bugs that manifest at large system scales. In *HPDC*, 2013.

[38] I. Laguna and M. Schulz. Pinpointing scale-dependent integer overflow bugs in large-scale parallel applications. In *SC*, 2016.