ParaStack: Efficient Hang Detection for MPI Programs at Large Scale

Hongbo Li, Zizhong Chen, Rajiv Gupta
University of California, Riverside
CSE Department, 900 University Ave
Riverside, CA 92521
{hli035,chen,gupta}@cs.ucr.edu

ABSTRACT

While program hangs on large parallel systems can be detected via the widely used timeout mechanism, it is difficult for the users to set the timeout – too small a timeout leads to high false alarm rates and too large a timeout wastes a vast amount of valuable computing resources. To address the above problems with *hang detection*, this paper presents *ParaStack*, an extremely lightweight tool to detect hangs in a timely manner with high accuracy, negligible overhead with great scalability, and without requiring the user to select a timeout value. For a detected hang, it provides direction for further analysis by telling users whether the hang is the result of an error in the computation phase or the communication phase. For a computation-error induced hang, our tool pinpoints the faulty process by excluding hundreds and thousands of other processes. We have adapted *ParaStack* to work with the *Torque* and *Slurm* parallel batch schedulers and validated its functionality and performance on *Tianhe-2* and *Stampede* that are respectively the world's current 2^{nd} and 12^{th} fastest supercomputers. Experimental results demonstrate that *ParaStack* detects hangs in a timely manner at negligible overhead with over 99% accuracy. No false alarm is observed in correct runs taking 66 hours at scale of 256 processes and 39.7 hours at scale of 1024 processes. *ParaStack* accurately reports the faulty process for computation-error induced hangs.

1 INTRODUCTION

Program hang, the phenomenon of unresponsiveness [34], is a common yet difficult type of bug in parallel programs. In large scale MPI programs, errors causing a program hang can arise in either the computation phase or the MPI communication phase. Hang causing errors in the computation phase include infinite loop [31] within an MPI process, local deadlock within a process due to incorrect thread-level synchronization [28], soft error in one MPI process that causes the process to hang, and unknown errors in either software or hardware that cause a single computing node to freeze. Errors in MPI communication phase that can give rise to a program hang include MPI communication deadlocks/failures.

SC17, Denver, CO, USA

DOI: 10.1145/3126908.3126938

It is widely accepted that some errors manifest more frequently at large scale both in terms of the number of parallel processes and problem size as testing is usually performed at small scale to manage cost and some errors are scale and input dependent [12, 29, 36, 39, 40]. Due to communication, an error triggered in one process (*faulty process*) gradually spreads to others, finally leading to a global hang. Although a hang may be caused by a single faulty process, this process is hard to locate as it is not easily distinguishable from other processes whose execution has stalled. Thus, the problem of *hang diagnosing*, i.e. locating faulty processes, has received much attention [11, 27, 28, 31].

Typically *hang diagnosing* is preceded by the *hang detection* step and this problem has not been adequately addressed. Much work has been done on communication-deadlock — a special case of hang — using methods like time-out [17, 25, 32], communication dependency analysis [20, 38], and formal verification [37]. These tools either use an imprecise timeout mechanism or precise but centralized technique that limits scalability. MUST [22, 23] claims to be a scalable tool for detecting MPI deadlocks at large scale, but its overhead is still non-trivial as it ultimately checks MPI semantics across all processes. These non-timeout methods do not address the full scope of hang detection, as they do not consider computation-error induced hangs. In terms of hang detection, ad hoc *timeout* mechanism [2, 27, 28, 31] is the mainstream; however, it is difficult to set an appropriate threshold even for users that have good knowledge of an application. This is because the optimal timeout not only varies across applications, but also with input characteristics and the underlying computing platform. Choosing a timeout that is too small leads to high false alarm rates and too large timeouts lead to long detection delays. The user may favor selecting a very large timeout to achieve high accuracy while sacrificing delay in detecting a hang. For example, IO-Watchdog [2] monitors writing activities and detects hangs based on a user specified timeout with 1 hour as the default. Up to 1 hour on every processing core will be wasted if the user uses the default timeout setting. Thus, a lightweight hang detection tool with high accuracy is urgently needed for programs encountering non-deterministic hangs or sporadically triggered hangs (e.g., hangs that manifest rarely and on certain inputs). It can be deployed to automatically terminate erroneous runs to avoid wasting computing resources without adversely effecting the performance of correct runs.

To address the above need for *hang detection*, this paper presents *ParaStack*, an extremely lightweight tool to detect hangs in a timely manner with high accuracy, negligible overhead with great scalability, and without requiring the user to select a timeout value. Due to its lightweight nature, *ParaStack* can be deployed in production

runs without adversely affecting application performance when no hang arises. It handles communication-error-induced hangs and hangs brought about by a minority of processes encountering a computation error. For a detected hang, *ParaStack* provides direction for further analysis by telling whether the hang is the result of an error in the computation phase or the communication phase. For a computation-error induced hang, it pinpoints faulty processes.

ParaStack is a parallel tool based on stack trace that judges a hang by detecting dynamic manifestation of following pattern of behavior – **persistent existence of very few processes outside of MPI calls**. This simple, yet novel, approach is based upon the following observation. Since processes iterate between computation and communication phases, a persistent dynamic variation of the *count* of processes outside of MPI calls indicates a healthy running state while a continuous small count of processes outside MPI calls strongly indicates the onset of a hang. Based on execution history, *ParaStack* builds a runtime model of *count* that is robust even with limited history information and uses it to evaluate the likelihood of continuously observing a small count. A hang is verified if the likelihood of persistent small count is significantly high. Upon detecting a hang, *ParaStack* reports the process in computation phase, if any, as faulty.

The above execution behavior based model is capable of detecting hangs for different target programs, with different input characteristics and sizes, and running on different computing platforms without any assistance from the programmer alike. *ParaStack* reports hang very accurately and in a timely manner. By monitoring only a constant number of processes, *ParaStack* introduces negligible overhead and thus provides good scalability. Finally, it helps in identifying the cause of the hang. If a hang is caused by a faulty process with an error, all the other concurrent processes get stuck inside MPI communication calls. If the error is inside communication phase, the faulty process will also stay in communication; otherwise, it will stay in computation phase. Simply checking whether there are processes outside of communication can tell the type of hang, communication-error or computation-error induced, as well as the faulty processes for a computation-error induced hang. The main contributions of *ParaStack* are:

- *ParaStack* introduces highly efficient non-timeout mechanism to detect hangs in a timely manner with high accuracy, negligible overhead, and great scalability. Thus it avoids the difficulty of setting the timeout value.
- *ParaStack* is a lightweight tool that can be used to monitor the healthiness of *production runs* in the commonly used batch execution mode for supercomputers. When there is a hang, by terminating the execution before the allocated time expires, *ParaStack* can save, on average, 50% of the allocated supercomputer time.
- *ParaStack* sheds light on the roadmap for a detected hang's further analysis by telling whether it was caused by an error in computation or communication phase. In addition, it pinpoints faulty processes for a computation-error induced hang.
- *ParaStack* is integrated into two parallel job schedulers *Torque* and *Slurm* and we validated its performance on the world's current 2nd and 12th fastest supercomputers—*Tianhe-2* and *Stampede*. For a significance level of 0.1%, experiments demonstrate that *ParaStack* detects hangs in a timely manner at negligible overhead with over 99% accuracy. No false alarm was observed in correct runs taking about 66 hours in total at the scale of 256 processes and 39.7 hours at the scale of 1024 processes. In addition, *ParaStack* accurately identifies the faulty process for computation-error induced hangs.

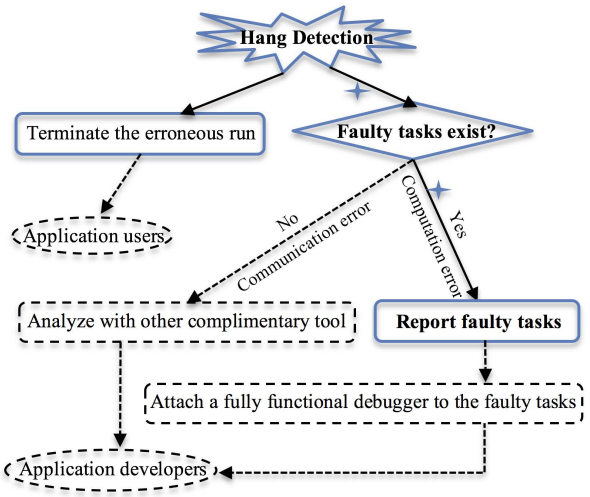

Figure 1: ParaStack workflow – steps with solid border are performed by ParaStack and those shown with dashed border require a complimentary tool.

2 THE CASE FOR PARASTACK

Hang detection is of value to application *users* and *developers* alike. Application *users* usually do not have the knowledge to debug the application. In batch mode, when a hang is encountered, the application simply wastes the remainder of the allocated computing time. This problem is further exacerbated by the fact that users commonly request a bigger time slot than what is really needed to ensure their job can complete. If users are unaware of a hang occurrence, they may rerun the application with even a much bigger time allocation, which will lead to even more waste. By attaching a hang detection capability to a batch job scheduler with negligible overhead, ParaStack can help by terminating the jobs and reporting the information to users when it detects a hang. Thus the unnecessary waste of computing resources is avoided.

Application *developers* need to detect a hang first and then debug based on the information given by *ParaStack*. First, knowing whether the hang-inducing error is in computation or communication sheds light on the direction for further analysis. To debug hangs due to communication error, such as global deadlock, is hard and it usually requires comparatively more heavyweight progress dependency analysis, communication dependency analysis or stack trace analysis. Since stack-trace analysis based tools such as STAT [12] do not require runtime information, they can be applied immediately after *ParaStack* reports a hang. In addition, the faulty process can be identified easily for a computation error induced hang, which benefits developers significantly by reducing from hundreds and thousands of suspicious processes to only one or a few.

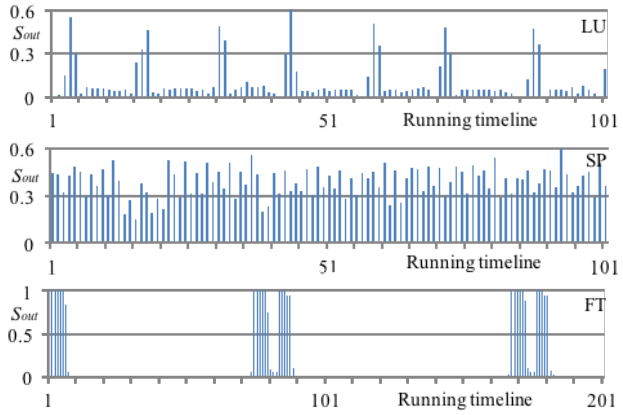

Figure 2: Dynamic variation of S_{out} observed from 3 benchmarks: LU, SP, FT from NPB suite. All are executed with 256 processes at problem size *D*.

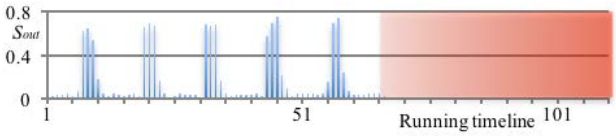

Figure 3: The S_{out} variation of a faulty run of LU, where a fault is injected on the left border of the red region.

The workflow of our tool is depicted in Figure 1. The path marked with blue stars is the main focus of this paper. If a hang happens and no faulty process is reported, we assume implicitly that the hang is caused by communication errors. For *ParaStack* users, debugging of a computation error induced hang has two phases: (1) monitoring the execution to detect hangs and report the faulty process with a lightweight diagnosis tool; and (2) debugging the faulty process with a fully functional debugger. *ParaStack* is an extremely lightweight tool for the first phase.

3 LIGHTWEIGHT HANG DETECTION

We begin by presenting the key observation that distinguishes the runtime behavior of a correctly functioning MPI program from one that is experiencing a hang. An MPI program typically consists of a trivial setup phase followed by a time-consuming loop-based solver phase where the latter is more error-prone. The solver loop can be viewed as consisting of a mix of computation code and communication code where the latter belongs to the MPI library. Depending upon the code being executed by a process, we classify the runtime state of the process as: ***IN_MPI*** if it is executing code in an MPI call; or ***OUT_MPI*** if it is executing non-MPI code. At any point in time, each process can be only in one state. Further ***OUT_MPI* significance**, denoted as S_{out}, is defined as the fraction of an application's parallel processes that are in state *OUT_MPI* at a given time. Next we argue that S_{out} can be used to distinguish between *healthy state* and *hang state*.

– *Healthy runtime state* is characterized by S_{out}'s periodic pattern. Parallel processes run in and out of MPI functions repeatedly in a healthy run. Thus, a healthy process frequently switches between states ***IN_MPI*** and ***OUT_MPI***. Because of the loop structure, parallel processes are expected to show a repetitive pattern in terms of how they flip state from one to the other. Thus, S_{out} is expected to vary over time in a periodic pattern. Figure 2 shows the periodic variation of S_{out} in healthy executions of 3 benchmarks: LU, SP, and FT from NPB suite [5]. This is obtained by repeatedly checking S_{out} at fixed time interval of 1 millisecond. We see that the length of the period varies as it is influenced by factors such as problem size and application type.

– *Hang runtime state* is characterized by a persistently low S_{out}. If a hang is caused by a *computation error*, the majority of processes in state *IN_MPI* should form a tight communication dependency on the faulty processes and the faulty processes in state *OUT_MPI* should be in the minority. If a hang is caused by a *communication error*, all processes should be in state *IN_MPI* and thus S_{out} should be 0 persistently. Figure 3 plots S_{out} during a run of LU benchmark during which a hang is encountered – the dynamic variation ceases and S_{out} is very low after the hang's occurrence. *Thus, the health of an application can be judged by looking for consecutive observations of very low S_{out}.*

Depending upon whether the utilized function is blocking or not, we classify MPI communication styles into 3 types: *blocking* style, i.e. a blocking communication function; *half-blocking*, i.e. a non-blocking communication followed by a blocking function like MPI_Wait to wait for its completion; and *non-blocking* style where a non-blocking communication is performed followed by a check for completion using a busy waiting loop using non-blocking message checking function like MPI_Test. Our characterization of runtime state is able to detect hangs for programs only using the first two styles of communication. For a program that uses a mix of different communication styles, the lesser the use of third communication style the more useful is our approach. For example, HPL uses a mixed style with a small portion of the program in the third style, processes can get stuck at multiple sites upon a program hang and thus a significant fraction of processes would ultimately stay in blocking functions and thus be in *IN_MPI* while some may flip states forever in busy-waiting loops. Our observation is still useful in this case.

Putting S_{out} into Practice. Precisely tracking S_{out} will require monitoring the runtime state of all processes continuously and this will lead to high overhead. To achieve our objective of developing a lightweight hang detection method, we neither monitor all processes nor do we monitor their states continuously. In particular, we determine the state of *constant number of processes* (say C), at *fixed time intervals* (say I), and compute S'_{out} that denotes the fraction of C processes that are in state *OUT_MPI*. A hang is reported if S'_{out} is observed to be persistently low, i.e., no bigger than a threshold t, for K consecutive intervals.

Now the next challenge is determine a selection of the values for C, I, K and t. To simply the discussion, we fix C at 10 processes – this choice is out of performance considerations and its justification is given later in Section 3.3, and fix t at 0 as it is rare that the faulty process happens to be among the randomly selected C processes in a small number of runs. Let us first consider a simple scheme in which the hang detection algorithm a priori fixes the values of both I and K. In fact this scheme is similar in spirit to the commonly used fixed timeout methods [2, 27, 28, 31] that avoid the complexities of chosing the timeout value.

Table 1: Adjusting the timeout method to various benchmarks, platforms and input sizes at scale 256 based on 10 erroneous runs per configuration. *Metrics*: *AC* – accuracy; *FP* – false positive rate; *D* – average response delay in seconds, i.e. the elapsed time from when the fault is injected to when a hang is detected.

Platform →	Tianhe-2						Tardis								
Benchmark(Input size) →	FT(D)			FT(E)			FT(D)			LU(D)			SP(D)		
Metrics →	*AC*	*FP*	*D*	*AC*	*FP*	*D*	*AC*	*FP*	*D*	*AC*	*FP*	*D*	*AC*	*FP*	*D*
$I_1 = 400ms,\ K_1 = 5\ times$	1.0	0.0	3.3	0.0	1.0	—	0.0	1.0	—	0.0	1.0	—	0.3	0.7	2.0
$I_2 = 400ms,\ K_2 = 10\ times$	1.0	0.0	8.1	1.0	0.0	10.9	0.9	0.1	6.5	1.0	0.0	5.3	1.0	0.0	5.1
$I_3 = 800ms,\ K_3 = 5\ times$	1.0	0.0	7.2	1.0	0.0	11.7	0.8	0.2	7.0	1.0	0.0	3.9	1.0	0.0	3.9
$I_4 = 800ms,\ K_4 = 10\ times$	1.0	0.0	13.2	1.0	0.0	17.4	1.0	0.0	10.2	1.0	0.0	10.7	1.0	0.0	8.6

Next we studied the effectiveness of this simple scheme by studying its precision, i.e. studying: (a) accuracy of catching real hangs; and (b) false positive rate, i.e. detection of hangs when none exist. In this study we used two values for I (400ms and 800ms), two values for K (5 times and 10 times) and then ran experiments for three applications (FT, LU, SP) on two platforms (Tianhe-2 and Tardis). The results obtained are given in Table 1 and by studying them we observe the difficulty of setting the (I, K) parameters for different platforms, different input sizes, and different applications. In particular, we observe the following:

- (Platforms: Tianhe-2 vs. Tardis) Consider the case for FT at input size *D*. For (I_1, K_1) while on Tianhe-2 all actual hangs are correctly reported, on Tardis false hangs are reported during the correct execution phase (i.e., before a hang actually occurs) in all 10 runs.
- (Input sizes for FT: *D* vs. *E*) On Tianhe-2 though (I_1, K_1) has a 100% accuracy for FT at input size *D*, for input size *E* the accuracy drops to 0% and false positive rate goes up to 100%.
- (Target Application: LU and SP vs. FT) For parameter settings (I_2, K_2) and (I_3, K_3), on Tardis though the accuracy for LU and SP is 100%, the accuracy for FT is less and false alarms are reported.

Clearly the above results indicate that fixed settings of (I, K) are not acceptable and thus a more sophisticated strategy must be designed. We observe that no fixed setting of parameters will work for all programs, on all inputs, and different platforms. Therefore the choice of parameters must be made based upon on the runtime characteristics of an application on a given input and platform. We cannot leave this choice to the users as they are likely to resort to guessing the parameter settings and thus will not have any confidence in the results of hang detection.

Therefore the approach we propose is one that automates the selection and tuning of I and K at runtime such that hangs can be reported with high degree of confidence. In fact the approach we propose allows the user to specify the desired degree of confidence and our runtime method ensures that a hang's presence is verified to meet the specified desired degree of confidence. The details of this method are presented next.

3.1 Model Based Hang Detection Scheme

The basic idea behind our approach is as follows. We randomly sample S_{out} at runtime to build and maintain a model and detect hangs by checking S_{out} against the model.

Random sampling of S_{out}. Variation of S_{out} over time is composed of many small cycles, and all cycles exhibit similar trend over time. Suppose the cycle time is C_t. If we randomly take a sample from a time range of NC_t where $N \in \mathbb{N}^+$, no matter how N varies it is clear this randomly observed S_{out} will follow the same distribution denoted as $F(S_{out})$, considering the similarity across cycles. Such random sampling can be achieved by inserting a good *uniformly generated random* time step, denoted as r_{step}, that makes the next sample fall at any point in one or several cycles, between two consecutive samples.

Suppose I is the *maximum time interval*, and $rand(I)$ is a uniform random number generator over $[0, I]$. We make $r_{step} = rand(I) + I/2$ and thus the sampling interval ranges over $[I/2, 3I/2]$ with an average of I. An ideal model can be built either when $I = NC_t$ or when the I is way bigger than C_t so that the sampling is approximately random rather than time-dependent.

Automatically tuning *I*. Hand-tuning I is undesirable and impractical as C_t varies across different applications, input sizes, and underlying computing platforms. Instead, we can achieve approximate random sampling through enlarging I as below. We design an automatic method to enlarge the maximum interval I in the early execution stage by checking the samples' randomness. *If the sampling is statistically found to lack randomness, we double I, as a bigger I leads to better randomness, and then re-evaluate the randomness.* Below details the method we use to check the sampling's randomness.

Runs test [35] is a standard test that checks a randomness hypothesis for a two-valued data sequence. We use this to judge the randomness of a sample sequence. Given a sample sequence of S_{out}, we set the average of samples as *boundary*. Samples bigger than or equal to the boundary are coded as *positive* (+) and samples smaller than that as *negative* (-). A *run* is defined as a series of consecutive positive (or negative) values. Under the assumption that the sequence containing N_1 positives and N_0 negatives is random, the *number of runs*, denoted as R, is a random variable whose distribution is approximately normal for large runs test. Given a significance level 0.05, for small runs test ($N_1 \leq 20, N_0 \leq 20$), we can get a range for the *assumed correct* number of runs via table in [35], i.e. the non-rejection region. If R is beyond this range, we reject the claim that the sequence is random and thus relax I. On the other hand, if either $N_1 \leq 1$ or $N_2 \leq 1$ and thus the non-rejection region is not available, we also assume the sampling is not random to avoid the risk of failing to identifying a non-random sampling process.

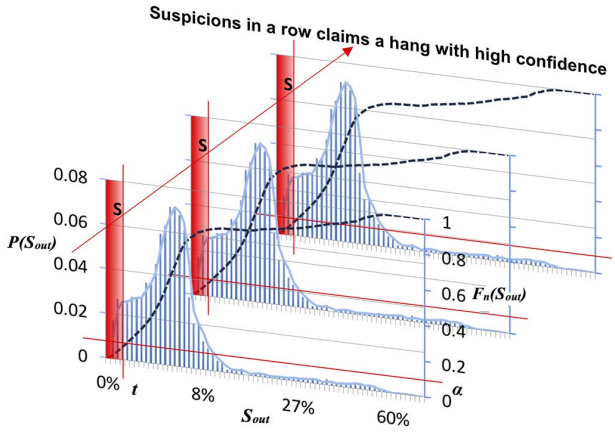

Figure 4: Hang detection. Three panels show the empirical distribution of randomly sampled S_{out} of LU, where the red region shows the suspicion region, the blue curve shows the probability density function $P(S_{out})$, and the dashed black curve shows the cumulative distribution function $F_n(S_{out})$. The red arrow crosses the suspicion region 3 times meaning 3 consecutive observations of suspicion.

For example, consider an MPI program running with 10 processes and thus the possible values of S_{out} are 0.1, 0.2, 0.3, ..., 1.0. There are a sequence 16 samples as follows

$$0.2\ \ 0.1\ \ 0.1\ \ 0.2\ \ \ 0.1\ \ 0.1\ \ 0.0\ \ 0.0$$
$$0.8\ \ 0.9\ \ 1.0\ \ 0.8\ \ \ 0.9\ \ 0.1\ \ 0.9\ \ 0.9,$$

which is equivalent to the two-valued sequence

$$--------++++++-++.$$

Its boundary is 0.44375 with $N_1 = 7$, $N_0 = 9$ and $R = 4$. The assumed correct range is (4, 14) and the number of runs 4 is not inside the range, so we claim the sampling is not random and double I.

Note the model needs to be renewed when I is doubled. The size of old samples collected at average time interval I is 2 times of the size if the old samples were collected at interval $2I$. Therefore, we cut the sample size by half.

Suspicion of hang. As samples accumulate, an empirical cumulative distribution function, denoted as $F_n(S_{out})$, can be built, where n is the number of samples. Given a probability $\hat{p}$, we can obtain $t = F_n^{-1}(\hat{p})$. A *suspicion* is defined as $S_{out} \leq t$, i.e. a very low S_{out}. The observed values can be classified into a pair of opposite random events:

$$\begin{cases} A: \text{Suspicion} & \text{if } S_{out} \leq t, \\ \overline{A}: \text{Non-suspicion} & \text{if } S_{out} > t, \end{cases}$$

Note $\hat{p}$ is selected dynamically to ensure robustness at various sample sizes and will be discussed in Section 3.2.

Significance test of hang. A single suspicion does not justify a hang's occurrence; instead, a continuous detection of suspicions indicates a hang with high confidence. We can quantify the number of suspicions (A) before the first observation of a non-suspicion ($\overline{A}$) as a *geometric distribution*. The probability of $Y = y$ observations of event A before the first observation of $\overline{A}$ can be expressed as follows:

$$P(Y = y) = q^y \cdot (1 - q)$$

where q is an estimation of the *true suspicion probability*, denoted as p, by adapting $\hat{p}$ and will be discussed in Section 3.2. Let us consider the following null and alternate hypothesis:

$$\begin{cases} H_0: & \text{The MPI application is healthy,} \\ H_1: & \text{The MPI application has hung.} \end{cases}$$

Under H_0, the probability to observe at least k consecutive As is

$$\begin{aligned} P_{H_0}(Y \geq k) &= 1 - \textstyle\sum_{y=0}^{k-1} q^y \cdot (1 - q) \\ &= q^k. \end{aligned}$$

Given the confidence level $1 - \alpha$, we reject H_0 and accept H_1 if $P_{H_0}(Y \geq k) \leq \alpha$, i.e., as below:

$$\begin{aligned} & q^k \leq \alpha \\ \Rightarrow\ & k \geq \lceil log_q \alpha \rceil . \end{aligned}$$

Hence a hang would be reported at a confidence level of $1 - \alpha$ if $\lceil log_q \alpha \rceil$ times of consecutive suspicions are encountered as depicted in Figure 4. The theoretical worst case time cost required to detect a hang is $I \cdot \lceil log_q \alpha \rceil$, considering a few *normal suspicions* in the correct phase may appear before a hang really appears.

3.2 Robust Model with a Limited Sample Size

Ideally, we would have $\hat{p} \approx p$ if the sample size is large enough. We can just apply $q = \hat{p}$ to the model. However, the problem is that the sample size can not be large enough as the sample size always grows from 0, and the assignment $q = \hat{p}$ thus would only make a bad hang detection model. To overcome this difficulty, we introduce a method for a achieving a credible q for each level of sample size.

Since we only care about *suspicion* versus *non-suspicion*, the sampling can be viewed as a Bernoulli process, i.e. $X_i \overset{i.i.d.}{\sim} Ber(p)$, where X_i is the i-th sample. By the rule of thumb [6], when $n\hat{p} > 5$ and $n(1 - \hat{p}) > 5$, $\hat{p}$ follows

$$\hat{p} = \frac{\sum_{i=1}^{n} X_i}{n} \overset{\circ}{\sim} N(p, \frac{p(1-p)}{n}).$$

Its 95% confidence interval is $\hat{p} \pm 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$. If we estimate p with an error no bigger than e, i.e. $\hat{p} \in [p - e, p + e]$, at 95% confidence, we have $1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \leq e$. The minimal sample size to justify $\hat{p}$ is

$$\hat{n} = max\{\frac{5}{\hat{p}}, \frac{5}{1-\hat{p}}, \frac{3.8416}{e^2}\hat{p}(1-\hat{p})\},$$

where $0 < \hat{p} < 1$. Because $\hat{p}$ and $1-\hat{p}$ are exchangeable and $\frac{5}{\hat{p}} > \frac{5}{1-\hat{p}}$ in $(0, 0.5]$, we only study

$$\hat{n} = f_{max}(\hat{p}) = max\{\frac{5}{\hat{p}}, \frac{3.8416}{e^2}\hat{p}(1-\hat{p})\}, \text{ where } \hat{p} \in (0, 0.5].$$

where $\hat{p}\ in (0, 0.5]$ and f_{max} is the function that gets the maximum between the given two terms.

Given a tolerance error e, our goal is to get an acceptable $\hat{p}$ that can be justified by the smallest sample size n, where the *smallest* ensures the model as soon as possible even with a small sample size. We provide 4 acceptable tolerance levels, 0.3, 0.2, 0.1 and 0.05, to study the relation among suspicion probability, tolerance error and

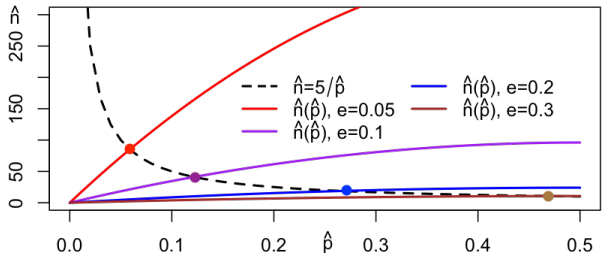

Figure 5: Relation among sample size, suspicion probability and tolerance error, where $\hat{n}(\hat{p}) = \frac{3.8416}{e^2}\hat{p}(1-\hat{p})$.

sample size as shown in Figure 5. Given e, let's denote the minimal $\hat{n}$ as $\hat{n}_m$ and the $\hat{p}$ that minimizes $\hat{n}$ as $\hat{p}_m$. With e equaling 0.3, 0.2, 0.1 and 0.05, we get $(\hat{p}_m, \hat{n}_m)$ respectively as $(0.47, 11)$, $(0.27, 19)$, $(0.12, 42)$ and $(0.06, 86)$. These points specify a path demanding the least sample size to step from a larger tolerance error to a smaller one, i.e. from 0.3 to 0.05. At 95% confidence, we estimate p as below:

$$\begin{cases} p \in [0.17, 0.77] & \text{when } 11 \le n < 19, \\ p \in [0.07, 0.47] & \text{when } 19 \le n < 42, \\ p \in [0.02, 0.22] & \text{when } 42 \le n < 86, \\ p \in [0.01, 0.11] & \text{when } n \ge 86. \end{cases}$$

It coincides with our intuition that a smaller $\hat{p}$ with a smaller e must be justified by a larger n.

However, the model is discrete and very likely such $\hat{p}_m$ does not exist. We thus need to find the sub-optimal $\hat{p}$, denoted as $\hat{p}_{m'}$, around $\hat{p} = \hat{p}_m$, which ensures a sub-minimum $\hat{n}$, denoted as $n_{m'}$. With $t_1 = max\{X\}$, where $F_n(X) < \hat{p}_m$, and $t_2 = min\{X\}$, where $F_n(X) \ge \hat{p}_m$, we have

$$\hat{n}_{m'} = min\{f_{max}(F_n(t_1)), f_{max}(F_n(t_2))\},$$

upon which $\hat{p}_{m'}$ is known. With e equal to 0.3, 0.2, 0.1, 0.05, we can respectively obtain $(\hat{p}_{m'}, \hat{n}_{m'})$ as $(\hat{p}_{m',0.3}, \hat{n}_{m',0.3})$, $(\hat{p}_{m',0.2}, \hat{n}_{m',0.2})$, $(\hat{p}_{m',0.1}, \hat{n}_{m',0.1})$, $(\hat{p}_{m',0.05}, \hat{n}_{m',0.05})$, where $\hat{n}_{m',0.3} < \hat{n}_{m',0.2} < \hat{n}_{m',0.1} < \hat{n}_{m',0.05}$. Therefore, we would have a $\hat{p}$ with a known maximal error at 95% confidence for each level of sample size:

$$\begin{cases} \hat{p}_{m',0.3} \in [p-0.3, p+0.3] & \text{when } n \in [\hat{n}_{m',0.3}, \hat{n}_{m',0.2}), \\ \hat{p}_{m',0.2} \in [p-0.2, p+0.2] & \text{when } n \in [\hat{n}_{m',0.2}, \hat{n}_{m',0.1}), \\ \hat{p}_{m',0.1} \in [p-0.1, p+0.1] & \text{when } n \in [\hat{n}_{m',0.1}, \hat{n}_{m',0.05}), \\ \hat{p}_{m',0.05} \in [p-0.05, p+0.05] & \text{when } n \ge \hat{n}_{m',0.05}. \end{cases}$$

Robust model. The value of $(\hat{p}_{m'}, \hat{n}_{m'})$ is continuously updated as the sample size increases. At each sample size level, a suspicion is defined by the obtained credible $\hat{p}_{m'}$. Because of the maximum error e, we might underestimate p ($\hat{p}_{m'} < p$) and undermine the hang detection accuracy. To avoid this, we make $q = \hat{p}_{m'} + e$. Because the confidence of $\hat{p}_{m'} \in [p-e, p+e]$ is 95%, we claim $q \ge p$ with 97.5% confidence.

Before the hang detection is performed, ParaStack needs to accumulate at least $\hat{n}_{m',0.03}$ random samples to build a model. The model building time is thus $\hat{n}_{m',0.03} \cdot I$. Since different applications may have different appropriate values of I that assure randomness, the model building time also varies from one application to another.

3.3 Lightweight Design Details

One monitor per node can be launched to examine the runtime state of all processes on a local node. But checking the call stack of all processes to sample S_{out} can slowdown the target application's execution. Hence, following lightweight strategy is introduced.

***CROUT_MPI* significance**. We monitor only a constant number, say C, of processes instead of all. Accordingly, we define *CROUT_MPI Significance*, denoted as S_{crout}, as the fraction of processes at OUT_MPI in a *R*andomly selected C processes. The hang detection scheme is still valid by checking S_{crout} as the idea of looking for a rare event remains unchanged.

Since only C processes need to be checked and some of them might coexist on the same node, they *at most* occupy C compute nodes, each of which requires one monitor actively checking the selected processes. We thus say monitors on these nodes are *active* and the others are *idle*. This design makes our tool extremely lightweight because: (1) only C processes' states need to be checked at a time cost of several microseconds per check; (2) communication is only required in a very limited scope of no more than C *active* monitors; and (3) the already trivial time cost can be possibly overlapped by target applications' idle time.

Parameter Setting. (1) The lightweight design requires an appropriate setting of C and I. A larger C leads to more overhead, and a smaller initial value of I also does so though it will be enlarged at runtime. In addition, a small value for C like 2 or 3 can flatten the variation of S_{crout} and thus diminishes the flexibility of adjusting $\hat{p}$ that ensures model robustness. Therefore, we set C to 10 first and then find I with initial value of 400 milliseconds to satisfy the above requirements. (2) We perform the *runs test* every 16 samples until randomness is ensured as that is large enough for runs test and small enough to ensure ParaStack has the smallest sample size required to check hangs. (3) We set $\alpha = 0.1\%$, which is statistically highly significant and implies 99.9% confidence. Note this is the only parameter that is tailored by the users.

Prevention of a corner case failure. Rarely a corner case arises due to the dynamic adjustment of what defines a suspicion to ensure model's robustness. When a suspicion is defined as $S_{crout} = 0$ and one faulty process, whose state is *OUT_MPI* after a hang happens, is one of the C processes being monitored, i.e. $S_{crout} \neq 0$, neither suspicions nor hangs will be observed. This corner case failure can be avoided by monitoring two *disjoint* random process sets, since the faulty process cannot be present in both sets. Of course more sets are required to be resilient for the case containing multiple faulty processes. *ParaStack* alternates between the two sets using each for a fixed number of observations. Since ideally $q \le 0.77$ and $log_{0.77}0.001 = 26.5$, the maximal times of suspicions required to verify a hang is 27. ParaStack alternates between two sets every 30 times to ensure it has enough time to find hangs while monitoring the process set with $S_{crout} = 0$ before switching to the other one with $S_{crout} \neq 0$.

Transient slowdowns at large scale. As noted in recent works, on large scale systems, the system noise can sometimes lead to a substantial transient slowdown of an application [24, 33]. We also occasionally encountered transient slowdowns on Tianhe-2 – typically in less than 4 runs out of a total of 50 runs. It is important not

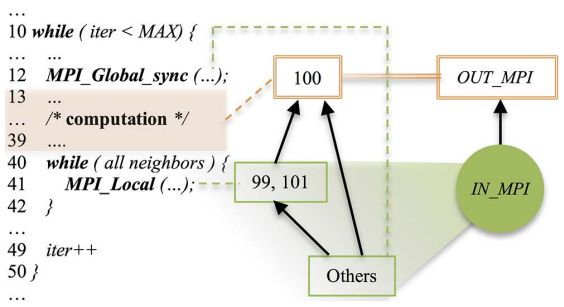

Figure 6: Faulty process identification for *computation-error induced hangs*. On the left is an MPI program skeleton, which hangs due to a computation error in process 100. Traditionally, the faulty process can be detected based on the progress dependency graph as shown in the middle. Our technique greatly simplifies the idea by just checking runtime states as shown on the right.

to confuse a transient slowdown with a hang. We observe that a transient slowdown is distinguishable from a hang because, unlike a hang, it is characterized by the presence of a few processes stepping through the code slowly. This transient-slowdown-specific effect can be identified if any of the following is true: (1) at least one process passes through different MPI functions; (2) at least one process steps in and out of MPI functions other than MPI_Iprobe, MPI_Test, MPI_Testany MPI_Testsome and MPI_Testall, i.e. a process running in a busy-waiting loop stepping across non-MPI code and a function like MPI_Test is treated as staying in the MPI function. Thus we check if such slowdown-specific effect exists based on two stack traces of each target process upon a hang report from the model-based mechanism. If it exists, we report a transient slowdown rather than a hang and resume monitoring; otherwise, we report a hang to users.

4 IDENTIFYING FAULTY PROCESS

Once a hang is detected, *ParaStack* reports the processes in *OUT_MPI* as faulty. The reported processes are claimed to contain the root cause of a hang that results from a computation error. If no process is reported, we claim the hang is a result of a communication error. Next we focus on locating the faulty process for a computation error induced hang.

On the left in Figure 6 we show the solver code skeleton of a typical MPI program that is expected to run at large scale. In addition to the computation, all processes perform both local communication and synchronization-like global communication. Synchronization-like global communication stands for a communication type that works like a synchronization across all processes such that no process can finish before all enter into the function call. For example, *MPI_Allgather* falls into this category, but *MPI_Gather* does not. In an erroneous run, process 100 fails to make progress due to a computation error, so its immediate neighbors, processes 99 and 101, wait for it at the local communication, which in turns causes all the others to hang at the global communication. Process 100 thus should be blamed as the *faulty process* for this hang. Locating this faulty process would take programmers a giant step closer to the root cause considering hundreds and thousands of suspicious processes are eliminated. Traditional progress-dependency-analysis methods [27, 28, 31] are very effective in aiding the identification of faulty process for general hangs but they involves complexities like recording control-flow information and progress comparison as shown in the middle of Figure 6. But for *computation-error induced hangs* these complexities are not necessary. ParaStack instead is inherently simple for this case.

Identification. Across *all* processes, we identify processes in state *OUT_MPI* as the faulty ones for a computation-error induced hang since all the other concurrent ones in *IN_MPI* would wait for the faulty processes. This can be achieved by simply glancing at the state of each process. As shown on the right in Figure 6, process 100 is easily located as it is the only one staying in state *OUT_MPI*.

Busy waiting loop based non-blocking communication. If a hang occurs in an application with busy waiting loops, in addition to persistently finding faulty processes in *OUT_MPI*, we may also occasionally find a few non-faulty in *OUT_MPI*. For example, HPL has its own implementation of collective communication based on busy waiting loops, which can make a few non-faulty processes step back and forth in a track trace rooted at such HPL communication functions when a hang appears. This can mislead ParaStack into believing that such non-faulty processes are faulty. To avoid this, we check every process's state several times and then select the ones that are in *OUT_MPI* persistently.

5 IMPLEMENTATION

ParaStack is implemented in C and conforms to the MPI-1 standard, by which we can ensure the maximum stability by only using a few old, yet good, widely-tested MPI functions while avoiding newly-proposed more error-prone functions. It was tested on Linux/Unix systems and integrated with popular batch job schedulers *Slurm* and *Torque*. It is easily usable by MPI applications using mainstream MPI libraries like MVAPICH, MPICH, and OpenMPI.

Job submission. We provide batch job submission command for Slurm and Torque. It processes users' allocation request, and executes the application and ParaStack concurrently. It ensures only one monitor per node is launched.

Mapping between MPI rank and process ID. ParaStack finds all the processes belonging to the target job by its command name using the common Linux/Unix command *ps*. Users submit a job by specifying the number of nodes and processes per node. Under this setting, the MPI rank assignment mechanism implies two rules: (1) MPI rank increases as process id increases on the same node; and (2) MPI rank increases as node id, in a ordered node list, increases. Suppose the number of target processes per node is *ppn* and a monitor's id is i. The MPI processes from rank $i*ppn$ to $(i+1)*ppn-1$ shares the same node with Monitor i that does the local mapping by simply sorting process ids.

Hang detection. (1) ParaStack suspends and resumes a processes' execution using *ptrace*, and resolves the call-chain using *libunwind*. (2) To obtain the runtime state, we examine stack frames to check if they *start* with *'mpi'*, *'MPI'*, *'pmpi'*, or *'PMPI'* until the backtrace finishes or such relation is found. If found, the state is *IN_MPI*; otherwise it is *OUT_MPI*. This works as mainstream MPI libraries use the above naming rule and users rarely use function

Table 2: Default input sizes used by each application at various running scales. Inputs *D* and *E* are the two largest inputs that come with the benchmarks. The input size for HPL specifies the width of a square matrix and the input size for HPCG specifies the local domain dimension.

Scale	256	1024	4096	8192	16384
BT, CG LU, SP	*D*	*E*		—	—
FT	*D,E*	*E*	—	—	—
MG	*E*	—	—	—	—
HPL	$8*10^4$	$2*10^5$	$2.5*10^5$	$3*10^5$	$3.5*10^5$
HPCG	*64 * 64 * 64*		—	—	—

names starting with such strings. (3) *Idle monitors* wait for messages in a busy waiting loop consisting of a hundreds-of-milliseconds-sleep and a nonblocking test to avoid preemption.

6 DISCUSSION

We discuss handling of complex situations by ParaStack.

Multi-threaded MPI program. A hybrid parallel program, using MPI+OpenMP or MPI+Pthreads, can have both thread-level and process-level parallelism. (1) For thread level MPI_THREAD_SINGLE and MPI_THREAD_FUNNE- LED, only the master thread communicates. Thus, ParaStack works by simply monitoring the master thread. (2) For more progressive mode, MPI_THREAD_SERIALIZED and MPI_THREAD_MULTIPLE, ParaStack must be adapted by redefining the runtime state of a process as: if at least one thread from a process is in MPI communication, we say this process is in *IN_MPI*; otherwise, it is *OUT_MPI*. Hence, a hang can still be captured by the fact that too few *processes* are in *OUT_MPI* persistently.

Applications with multiple phases. An application may alternate among several phases with differing behaviors leading to imprecision in the S_{crout} model. However, ParaStack can be easily adapted by constructing separate models for different phases if the application is instrumented to inform ParaStack of phase changes during execution. ParaStack can build separate models by sampling each of the phases and using them for respective phases.

Applications with load imbalance. ParaStack is developed to detect hangs for applications with good load balance and is not suitable for applications with severe load imbalance. For applications with *severe load imbalance*, near the end of execution, a few heavy-workload processes may be running. Thus our model based mechanism can fail. We ignore this situation because applications with severe load imbalance should not be deployed at large scale so as to avoid computing resources waste. For moderate load imbalance, we can apply the technique of detecting transient slowdowns as the load imbalance is also characterized by the effect that a few processes are still running slowly.

7 EXPERIMENTAL EVALUATION

Computing platforms. We evaluate ParaStack on three platforms: *Tardis*, *Tianhe-2* and *Stampede*. Tardis is a 16-node cluster, with each node having 2 AMD Opteron 6272 processors (with *32 cores* in all) and 64GB memory. Tianhe-2 is the *2nd* fastest supercomputer in the world [1], with each node having 2 E5-2692 processors (with *24 cores* in all) and 64GB memory. Stampede is the *12th* fastest supercomputer in the world [1], with each node having 2 Xeon E5-2680 processors (*16 cores* in all) and 32GB memory. Infiniband is used for all. We allocate respectively 8 nodes—256 (8*32) processes—on Tardis, 64 nodes—1,024 (64*16) processes—on Tianhe-2, and up to 1024 nodes—16,384 (1024*16) processes—on Stampede.

Table 3: For an execution of HPL on a 15000*15000 matrix, the clean run on average takes 185.05 seconds. O_t is the total stack trace overhead due to n stack trace operations.

Time interval	10 ms	100 ms
O_t	50.88s	7.52s
n	18220	1870

Applications and input sizes. We use six NAS Parallel Benchmarks (NPB: BT, CG, FT, MG, LU and SP) [5], High Performance Linpack (HPL) [7], and High Performance Conjugate Gradient Benchmark (HPCG) [21] for evaluation. The execution of these widely used benchmarks consists of a trivial setup phase and a time-consuming iterative solver phase. Though HPCG has multiple phases, all phases are iterative. As ParaStack is developed to monitor long-running runs, we use large available input sizes indicated in Table 2 by default unless otherwise specified. In our evaluation, we ignore MG due to its short execution time on both Stampede and Tianhe-2, and ignore FT on Stampede as it crashes at large scale due to memory limit. We did not inject errors in HPCG on Stampede and Tianhe-2 as it has multiple iterative steps and our random error injection technique is not readily applicable.

Fault injection. On Tardis, to simulate a hang, we suspend the execution of a randomly selected process by injecting a long sleep in a random invocation of a random user function as faults are more likely to be in the application than in well-known libraries [31]. We use *gprof* to collect all the *user functions*. *Dyninst* [8] is used to statically inject errors, i.e. long enough sleep calls, in application binaries. We discard the cases where error appears in the first 20 seconds of execution because real-world HPC applications spend the majority of time in the later solver phase and building our model takes around 20 seconds. Note our tool targets hangs in the middle of long runs such as those reported in [3, 4] and the model building time is trivial in comparison to the program execution time. On Stampede and Tianhe-2 we inject errors in the source code and simulate a hang by injecting a long sleep call in a randomly selected iteration of a randomly selected process.

7.1 Hang Detection Evaluation

I. Overhead. To begin with, we measured the overhead of stack trace for a single process running HPL, a highly compute intensive application. We executed 5 clean runs and 5 runs with stack trace using time intervals of 10ms and 100ms. The average cumulative total overhead (O_t) and number of stack trace operations (n) are given in Table 3. As we can see, the overhead is high for interval of 10ms – a 50.88 seconds increase over clean run that takes 185.05 seconds. However, for the interval of 100ms the overhead is low – 7.52 seconds. Thus, for I of 100ms or higher we can expect our tool to have very low overhead.

Table 4: Performance comparison of running applications with ParaStack (I = 100ms), with ParaStack (I = 400ms) and without ParaStack (clean) on Tardis at scale 256. *P*erformance is measured by the delivered *GFLOPS* for HPCG and by the *time cost in seconds* for the others, and *S*tandard deviation of the performance is shown.

Benchmark	BT		CG		FT		LU		MG		SP		HPL		HPCG	
Metric	*P*	*S*	*P*	*S*	*P*	*S*	*P*	*S*	*P*	*S*	*P*	*S*	*P*	*S*	*P*	*S*
clean	336.7	1.0	132.0	1.1	178.8	0.3	247.8	2.9	347.3	0.5	511.1	0.3	277.8	0.8	29.1	0.1
I=100	336.4	0.6	131.6	0.2	179.5	0.2	247.8	0.6	347.0	0.5	510.3	0.4	277.7	0.5	29.1	0.1
I=400	336.8	1.4	132.4	0.6	179.07	0.7	246.6	0.6	347.1	0.3	511.0	0.6	277.2	0.4	29.1	0.1

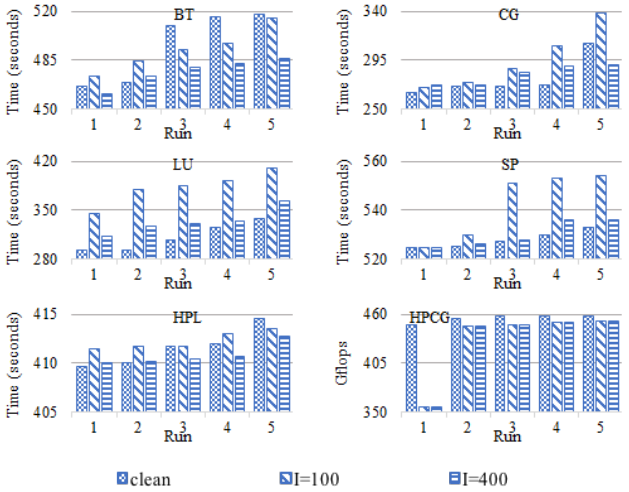

Figure 7: Performance comparison of running applications with ParaStack (I = 100 **ms), with ParaStack (I = 400 **ms**) and without ParaStack (clean) on Stampede at scale 1024 based on 5 runs in each setting. The performance is evaluated as *GFLOPS* for HPCG and as *time cost in seconds* for all the others, and the the 5 runs are ordered by performance.**

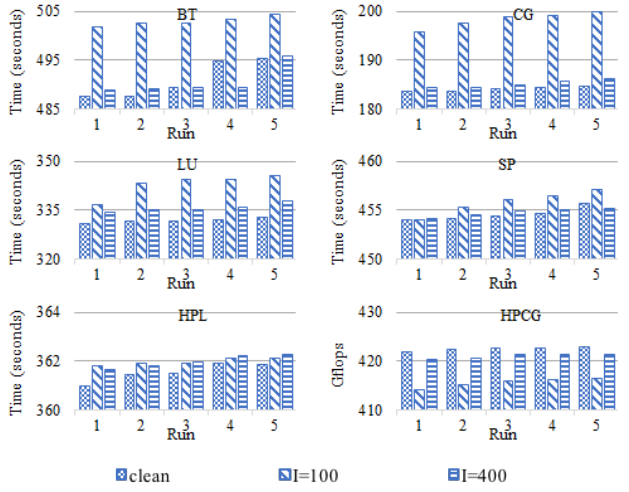

Figure 8: Performance comparison of running applications with ParaStack (I = 100 **ms), with ParaStack (I = 400 **ms**) and without ParaStack (clean) on Tianhe-2 at scale 1024.**

Table 5: ParaStack's Overhead on Tianhe-2 at scale 1024 based on the average of 5 runs.

Benchmark	BT	CG	LU	SP	HPL	HPCG
I=100	2.44%	7.61%	3.35%	0.26%	0.12%	1.64%
I=400	-0.08%	0.55%	1.14%	0.04%	0.12%	0.35%

Now we study the impact of using *ParaStack* on runtimes for all applications under two I settings of 100ms and 400ms at scales of 256 and 1024 processes. Note I does not change in this study – we disable the automatic adjustment of I. Experiment results are based on 5 runs at each setting. Table 4 shows results at scale 256 and it shows that ParaStack has negligible impact on applications' performance in either setting. At scale 1024, we separately present the performance for each of the 5 runs in each setting on Stampede and Tianhe-2 as the performance variations due to system noise are greater than the prior experiment. On Stampede, Figure 7 shows the performance for I = 400ms is often better than that with I = 100ms, and is almost the same as that of clean runs (except for LU). Since Tianhe-2 suffers less system noise due to its lower utilization rate than Stampede, the performance variation on it is less. Hence we can expect Tianhe-2 better captures ParaStack's overhead. On Tianhe-2, Figure 8 clearly shows that I = 400ms is always better than I = 100ms, and introduces a slight overhead compared with the clean runs. Table 5 shows the overhead for each application. The overhead with I = 400ms is at most 1.14%, which is always better than the overhead in the other setting. Hence for the rest of the experiments we use the setting of I = 400ms.

II. False positives were evaluated using 100 correct runs of each application at scale 256 on Tardis taking about 66 hours, 50 correct runs for BT, CG, FT, LU, SP, HPCG, and HPL at scale 1024 on Tianhe-2 taking about 27.9 hours, and 20 correct runs for BT, CG, LU, SP, HPCG, and HPL at scale 1024 on Stampede taking about 11.8 hours. *The false positive rate was observed to be 0% when the theoretical false positive rate is* α = 0.1%. In addition, no false positives were observed even in all erroneous runs performed in experiments presented next.

III. Accuracy refers to the effectiveness of *ParaStack* in detecting hangs in erroneous runs. Let the total number of faulty runs be T and the total number of times that the hang can be detected correctly be T_h. The accuracy is defined as T_h/T. Table 6 shows the accuracy based on 100 erroneous runs at scale 256 on Tardis, 50 erroneous runs at scale 1024 on Tianhe-2 and 20 runs at scale 1024 on Stampede. *ParaStack* misses only 6 times out of 800 runs at scale 256. In these cases, hangs happen very early (even before *ParaStack* has collected enough samples to build an accurate model) and thus

Table 6: Accuracy of hang detection. The rough time cost of a correct run is shown.

Platform	Tardis		Tianhe-2		Stampede	
# runs	100		50		20	
Scale	*256*		*1024*		*1024*	
Metric	Time(s)	AC_h	Time(s)	AC_h	Time(s)	AC_h
BT	336	99%	487	100%	495	100%
CG	132	100%	177	100%	278	100%
FT	179	98%	100	100%	—	—
LU	247	98%	328	98%	311	100%
MG	347	100%	—	—	—	—
SP	511	100%	454	100%	528	100%
HPCG	—	100%	—	—	—	—
HPL	277	99%	362	100%	411	100%

Table 7: Response delay on Tianhe-2: *D* is the average response delay in seconds; *S* is the standard deviation.

Scale↓	Metric↓	BT	CG	FT	LU	SP	HPL
1024	*D*	7.2	18.8	8.8	9.0	4.8	6.8
	S	7.3	14.7	7.3	4.2	2.2	3.3

Table 8: Response delay on Stampede: *D* is the average response delay in seconds and *S* is the standard deviation.

Scale↓	BT		CG		LU		SP		HPL	
	D	*S*	*D*	*S*	*D*	*S*	*D*	*S*	*D*	*S*
1024	7.1	4.5	7.6	4.5	7.8	5.9	4.1	1.2	5.0	2.5
4096	5.4	3.6	24.1	13.1	4.3	1.3	3.7	2.0	5.6	4.7

I is continuously enlarged and the probability of $S_{crout} = 0$ is increased. Hence there is not enough time to verify the hang before the allocated time slot expires. One hang in LU is also missed at scale 1024 on Tianhe-2; for all other runs at scale 1024, the accuracy is 100% on Stampede and Tianhe-2.

Due to the high cost, a limited number of experiments was conducted. At *scale 4096*, we studied ParaStack's accuracy based on 10 erroneous runs for BT, CG, LU, SP, and HPL. For BT, LU, and HPL, $AC_h = 1$; for CG and SP, AC_h equals 0.8 and 0.9 respectively. Also, as the later two take less time, errors are more likely to happen earlier. At *scale 8192*, the accuracy based on 5 erroneous runs of HPL is $AC_h = 5/5$. At *scale 16384*, the accuracy based on 3 erroneous runs of HPL is $AC_h = 3/3$.

IV. Response delay is the elapsed time from a hang's occurrence to its detection by *ParaStack*. For the erroneous runs where hangs are correctly identified by *ParaStack*, we collected the response delays for all applications. Figure 9 shows the response delay distribution for 100 erroneous runs at scale of 256 on Tardis. Table 7 shows the average response delay and the standard deviation based on 50 erroneous runs at scale of 1024 on Tianhe-2. Table 8 shows the average response delay and the standard deviation based on 20 erroneous runs at scale of 1024 and 10 erroneous runs at scale of 4,096 on Stampede. At *scale 8,192* the response delay for 5 erroneous runs of HPL are 5, 6, 14, 16, and 17 seconds. At

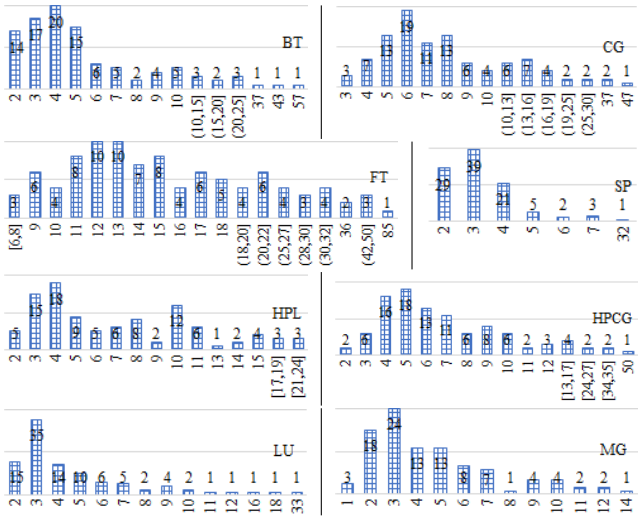

Figure 9: The response delay of hang detection based upon 100 erroneous runs for each application at scale of 256 on Tardis. *The horizontal axis represents response delay in seconds and the vertical axis represents the number of times ParaStack identifies a hang with the corresponding delay.*

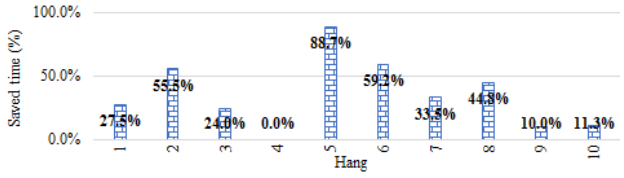

Figure 10: The percentage of time savings ParaStack brings to application users in batch mode based on 10 erroneous runs of HPL with the average percentage equal to 35.5%.

scale 16,384 response delays for 3 erroneous runs are 6, 7, and 10 seconds. *ParaStack* commonly detects a hang with a delay of no more than 1 minute at various scales. We observe that the response delay not only varies across applications, it also differs from one hang to another for a given application. As we know, the response delay in worst case is $I \cdot \lceil log_q \alpha \rceil$. The variation of q depending upon sample size and the adaptation of I leads to the variation in response delay.

V. ParaStack enabled time savings for application users. Supercomputers typically charge users in *Service Units* (*SUs*) [9, 10]. The total number of *SUs* charged for a job is equal to the product of the number of nodes occupied, the number of cores per node, and the elapsed wallclock time of the job. Application users run their job assuming absence of hangs; thus, when running an application in batch mode, the allocated time will be wasted if a hang arises and the user is charged for it. *ParaStack* saves this cost by terminating the application upon a hang. To quantify the time saving, we ran HPL 10 times using a problem size 100,000 with a (uniform) random error injected in the iterative phase. The correct run takes around 518 seconds, so users are inclined to request a larger time slot – conservatively let us assume a 10-minute time slot is requested. The percentage of time ParaStack saves is shown in Figure 10. For the 10

Table 9: ParaStack's generality for variation of platforms, benchmarks and input sizes at scale 256 based on 10 erroneous runs per configuration. Notes: (1) P stands for the default ParaStack with I being initialized as 400ms; $P*$ stands for ParaStack with I being initialized as 10ms. (2) *AC*, accuracy; *FP*, false positive rate; *D*, average response delay.

Platform	Bench.	P			P*		
		AC	*FP*	*D*	*AC*	*FP*	*D*
Tianhe-2	FT(D)	1.0	0.0	4.8	1.0	0.0	3.5
	FT(E)	1.0	0.0	29.4	1.0	0.0	14.9
Tardis	FT(D)	1.0	0.0	14.0	0.9	0.0	25.2
	LU(D)	1.0	0.0	4.5	1.0	0.0	1.1
	SP(D)	1.0	0.0	3.3	1.0	0.0	1.0

Table 10: Evaluation of faulty process identification.

Platform	Tardis		Tianhe-2		Stampede	
Scale	*256*		*1024*		*1024*	
Metric	AC_f	PR_f	AC_f	PR_f	AC_f	PR_f
BT	$^{99}/_{99}$	1.0	$^{50}/_{50}$	1.0	$^{20}/_{20}$	1.0
CG	$^{100}/_{100}$	1.0	$^{50}/_{50}$	1.0	$^{20}/_{20}$	1.0
FT	$^{97}/_{98}$	0.99	$^{50}/_{50}$	1.0	—	—
LU	$^{98}/_{98}$	1.0	$^{49}/_{49}$	1.0	$^{20}/_{20}$	1.0
MG	$^{100}/_{100}$	1.0	—	—	—	—
SP	$^{100}/_{100}$	1.0	$^{50}/_{50}$	1.0	$^{20}/_{20}$	1.0
HPCG	$^{100}/_{100}$	1.0	—	—	—	—
HPL	$^{99}/_{99}$	1.0	$^{50}/_{50}$	1.0	$^{20}/_{20}$	1.0

runs, on average the time saved is 35.5%. With increasing number of tests, the average time saved will approach 50%. Because ParaStack detects a hang soon after its occurrence, if hang is expected to happen randomly during execution, the average time at which the program is terminated is about half of the execution time.

VI. ParaStack vs. timeout. Unlike timeout method, ParaStack can report hang according to the user specified confidence which automatically adjust parameters like sampling interval, what defines a suspicion, how many times of suspicions confirm a hang. Our experimental results in Table 1 already demonstrated the drawbacks of timeout-based mechanism. In contrast, ParaStack's default configuration shows 100% accuracy and 0% false positive rate. Even though we initialize I with a very small value that does not deliver random sampling – say $I = 10$ in comparison to the default value of $I = 400$, ParaStack's effectiveness ($P*$) still compares well with the default (P). This is because ParaStack has capability of adapting I automatically so as to ensure random sampling. In short, the key advantage of *ParaStack* is that it reports hang based on runtime history with high confidence $1-\alpha$ while traditional timeout method is based upon guesses as shown in Table 1.

7.2 Faulty Process Identification

We evaluate the the effectiveness of faulty process identification using two metrics: *faulty process identification accuracy* (AC_f); and *faulty process identification precision* (PR_f). As the faulty process identification is only performed after a hang is detected, this evaluation is based on the same experiment as conducted in the hang detection accuracy evaluation. Recall, T_h denotes the total number of times that the hang is detected correctly. Let the number of times that the faulty process is found be T_f out of T_h times, and let x_i be the number of processes reported as faulty ones in the i-th run. For the i-th run, if the true faulty process is in this report, we say its precision (p_i) is $p_i = 1/x_i$ in this single run; otherwise, $p_i = 0$. The 2 metrics are defined as $AC_f = T_f/T_h$ and $PR_f = \frac{1}{T_h}\sum_{i=1}^{T_h} p_i$.

Table 10 gives results based on 100 erroneous runs at scale of 256 on Tardis, 50 erroneous runs at scale of 1024 on Tianhe-2, and 20 erroneous runs at scale 1024 on Stampede. In terms of accuracy, *ParaStack* misses the faulty process once at scale 256. Because this is a rare occurrence, we can handle it by printing debugging information for further analysis. The precision of faulty process identification for FT is approximately 99.0% as *ParaStack* misses the faulty process once out of 98 runs. The precision for all other applications is 100%.

At *scale 4096*, ParaStack's effectiveness based on 10 erroneous runs for BT, CG, LU, and SP is $AC_f = 1.0$, and $PR_f = 100\%$; for HPL, $AC_f = PR_f = 0.9$. At *scale 8192*, *ParaStack*'s effectiveness based on 5 erroneous runs of HPL is $AC_f = 5/5$, and $PR_f = 86.7\%$ as in one run ParaStack identifies 3 processes as faulty which includes the real faulty process while it precisely identifies the real faulty process the other 4 runs. At *scale 16384*, ParaStack's effectiveness based on 3 erroneous runs of HPL is $AC_f = 3/3$, and $PR_f = 100\%$.

8 RELATED WORK

Automatic bug detection for MPI programs. Many runtime approaches have been proposed to detect bugs. Umpire [38], Marmot [25], and Intel Message Checker [18] can detect MPI errors including deadlocks, resource errors, and type mismatches. Approaches for detecting deadlocks can be divided into three categories: timeout-based [17, 25, 32], communication dependency analysis [20, 38], and formal verification [37]. These tools either use an imprecise timeout mechanism or precise but centralized technique that limits scalability. MUST [22, 23] claims to be a scalable tool to detect MPI deadlock at large scale based on distributed wait state tracking, but its overhead is still non-trivial considering it checks MPI semantics across all processes. As deadlock is a special case of hang, the advantage of ParaStack over these non-timeout tools is that it is able to detect deadlock statistically at runtime with *negligible overhead*; the advantage of non-timeout tools is they are precise and potentially gives detailed insights to remove the errors. Also, ParaStack is better than time-out methods as has already been justified. DMTracker [19] finds errors resulting from anomaly in data movement. FlowChecker [15] detects communication-related bugs in MPI libraries. SyncChecker [16] detects synchronization errors between MPI applications and libraries. MC-Checker [14] detects memory consistency in MPI one-sided applications. In contrast, our tool detects errors producing the symptom of program hang due to various reasons.

Problem diagnosis in MPI programs at large scale. Techniques in [13, 26, 30] debug large-scale applications by deriving their normal timing behavior and looking for deviations from it. A few recent efforts focus on hangs and performance slowdowns. STAT [12] divides tasks (process/thread) into behavioral equivalence classes using call stack traces. It is very useful for further analysis once *ParaStack* identifies a hang, especially when hangs

are hard to reproduce. STAT-TO [11] extends STAT by providing temporal ordering among processes that can be used to identify the least-progressed processes; however, it requires expensive static analysis that fails in the absence of loop-ordered-variables. AutomaDed [27, 28] draws probabilistic inference on progress dependency among processes to nominate the least-progressed task, but it fails to handle loops. *Prodometer* [31] performs highly accurate progress analysis in the presence of loops and can precisely pinpoint the faulty processes of a hang. Prodometer's primary goal is to *diagnose* cause of hangs by giving useful progress dependency information. In contrast, *ParaStack*'s main aim is to *detect* hangs with high confidence in production runs. Thus Prodometer's capabilities are complementary to our tool.

9 CONCLUSION

By observing S_{crout}, ParaStack detects hangs with high accuracy, in a timely manner, with negligible overhead and in a scalable way. Based on the concept of *runtime state*, it sheds light on the roadmap for further debugging. It does not require any complex setup and supports mainstream job schedulers – *Slurm* and *Torque*. Its compliance with MPI standard ensures its portability to various hardware and software environments.

ACKNOWLEDGEMENTS

This work is partially supported by the NSF grants CCF-1524852, CCF-1318103, OAC-1305624, CCF-1513201, the SZSTI basic research program JCYJ20150630114942313, and the MOST key project 2017YFB0202100.

REFERENCES

[1] Top500 list. http://www.top500.org/lists/2015/11/.
[2] IO-Watchdog. https://code.google.com/p/io-watchdog/.
[3] Bug occurs after 12 hours. https://github.com/open-mpi/ompi/issues/81/.
[4] Bug occurs after 200 iterations. https://github.com/open-mpi/ompi/issues/99.
[5] NAS parallel benchmarks. https://www.nas.nasa.gov/publications/npb.html.
[6] Probability theory and mathematical statistics: normal approximation to binomial. https://onlinecourses.science.psu.edu/stat414/node/179.
[7] HPL: a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/.
[8] Paradyn project: Dyninst. http://www.paradyn.org/html/manuals.html#dyninst
[9] Ohio Supercomputer Center's charging policy. https://www.osc.edu/supercomputing/software/general#charging
[10] San Diego Supercomputer Center's charging policy. http://www.sdsc.edu/support/user_guides/comet.html#charging
[11] 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 *ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis* (SC), Article No. 44, Nov. 2009.
[12] D.C. Arnold, D.H. Ahn, B.R. de Supinski, G.L. Lee, B.P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *IEEE International Parallel and Distributed Processing Symposium* (IPDPS), pages 1–10, March 2007.
[13] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. Ahn, and M. Schulz. Automaded: Automata-based debugging for dissimilar parallel tasks. In *IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN), pages 231–240, July 2010.
[14] 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 International Conference for High Performance Computing, Networking, Storage and Analysis* (SC), pages 499–510, 2014.
[15] Z. Chen, Q. Gao, W. Zhang, and F. Qin. Flowchecker: Detecting bugs in mpi libraries via message flow checking. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (SC), 2010.
[16] Z. Chen, X. Li, J-Y. Chen, H. Zhong, and F. Qin. Syncchecker: Detecting synchronization errors between mpi applications and libraries. In *IEEE International Parallel and Distributed Processing Symposium* (IPDPS), pages 342–353, May 2012.
[17] J. Coyle, J. Hoekstra, G. R. Luecke, Y. Zou, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.
[18] 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*, pages 78–82, 2005.
[19] Q. Gao, F. Qin, and D. K. Panda. Dmtracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *ACM/IEEE Conference on Supercomputing* (SC), Article No. 15, 2007.
[20] W. Haque. Concurrent deadlock detection in parallel programs. *International Journal of Computers and Applications*, 28(1):19–25, Jan 2006.
[21] M. A Heroux and J. Dongarra. Toward a new metric for ranking high performance computing systems. *TR SAND2013-4744*, June 2013.
[22] 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 *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis* (SC), 2013.
[23] T. Hilbrich, B. R. de Supinski, Martin Schulz, and Matthias S. Müller. A graph based approach for mpi deadlock detection. In *ACM/IEEE International Conference on Supercomputing* (SC), pages 296–305, 2009.
[24] T. Hoefler and R. Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (SC), page 73. ACM, 2015.
[25] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. Marmot: An mpi analysis and checking tool. In *PARCO*, pages 493–500, 2003.
[26] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, B. Rountree. Large scale debugging of parallel tasks with AutomaDeD. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (SC), pages 50:1–50:10, 2011.
[27] 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 *International Conference on Parallel Architectures and Compilation Techniques* (PACT), pages 213–222, 2012.
[28] 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 *IEEE Transactions on Parallel and Distributed Systems* (TPDS), 26(5):1280–1289, 2015.
[29] 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):72–81, 2015.
[30] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller. Problem diagnosis in large-scale computing environments. In *ACM/IEEE Supercomputing Conference* (SC), Nov 2006.
[31] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin. Accurate application progress analysis for large-scale parallel debugging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 193–203, 2014.
[32] P. Ohly and W. Krotz-Vogel. Automated mpi correctness checking: What if there was a magic option? In *LCI HPCC*, 2007.
[33] F. Petrini, D.J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *ACM/IEEE Conference on Supercomputing* (SC), page 55. ACM, 2003.
[34] X. Song, H. Chen, and B. Zang. Why software hangs and what can be done with it. In *IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN), June 2010.
[35] F. S. Swed and C. Eisenhart. Tables for testing randomness of grouping in a sequence of alternatives. *The Annals of Mathematical Statistics*, 14(1):66–87, 03 1943.
[36] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications* (IJHPCA), page 1094342014522573, 2014.
[37] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. Isp: A tool for model checking mpi programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), 2008.
[38] J. S. Vetter and B. R. de Supinski. Dynamic software testing of mpi applications with umpire. In *ACM/IEEE Conference on Supercomputing* (SC), Article No. 51, 2000.
[39] B. Zhou, M. Kulkarni, and S. Bagchi. Vrisha: using scaling properties of parallel programs for bug detection and localization. In *International Symposium on High-Performance Distributed Computing* (HPDC), pages 85–96. ACM, 2011.
[40] B. Zhou, J. Too, M. Kulkarni, and S. Bagchi. Wukong: automatically detecting and localizing bugs that manifest at large system scales. In *International Symposium on High-performance Parallel and Distributed Computing* (HPDC), pages 131–142. ACM, 2013.