Fast Fault Injection and Sensitivity Analysis for Collective Communications

Kun Feng† Manjunath Gorentla Venkata‡ Dong Li* Xian-He Sun†
Oak Ridge National Laboratory‡ University of California, Merced* Illinois Institute of Technology†
manjugv@ornl.gov dli35@ucmerced.edu kfeng1@hawk.iit.edu, sun@iit.edu

***Abstract*—The collective communication operations, which are widely used in parallel applications for global communication and synchronization are critical for application's performance and scalability. However, how faulty collective communications impact the application and how errors propagate between the application processes is largely unexplored. One of the critical reasons for this situation is the lack of fast evaluation method to investigate the impacts of faulty collective operations. The traditional random fault injection methods relying on a large amount of fault injection tests to ensure statistical significance require a significant amount of resources and time. These methods result in prohibitive evaluation cost when applied to the collectives.**

In this paper, we introduce a novel tool named *Fast Fault Injection and Sensitivity Analysis Tool* (FastFIT) to conduct fast fault injection and characterize the application sensitivity to faulty collectives. The tool achieves fast exploration by reducing the exploration space and predicting the application sensitivity using *Machine Learning* (ML) techniques. A basis for these techniques are implicit correlations between MPI semantics, application context, critical application features, and application responses to faulty collective communications. The experimental results show that our approach reduces the fault injection points and tests by 97% for representative benchmarks (*NAS Parallel Benchmarks* (NPB)) and a realistic application (*Large-scale Atomic/Molecular Massively Parallel Simulator* (LAMMPS)) on a production supercomputer. Further, we statistically generalize the application sensitivity to faulty collective communications for these workloads, and present correlation between application features and the sensitivity.

I. Introduction

Collective communications such as MPI_Allreduce, MPI_Bcast, and MPI_Alltoall, are widely employed by *Message Passing Interface* (MPI) applications (commonly used class of scientific simulations) for exchanging data and synchronization. They are known to have a significant impact on the application performance and scalability because of their global nature [1], [2]. An error in collective communication can have a cascading effect on the application reliability because of its global nature. Hence, understanding the application sensitivity to faulty collective communication and improving them is key to improving application reliability.

Although there is a strong motivation to improve the reliability of collective communications (especially in MPI community [3]), we still do not have sufficient knowledge of how faulty collective communications impact the applications, and how errors propagate between the processes during execution. The lack of knowledge is a major obstacle to arrive at an effective fault-tolerant mechanism.

One of the critical reasons for this situation is lack of effective method that enables fast evaluation of application sensitivity to faulty collective communications. The random fault injection [4], [5], [6], [7], [8], [9], the most common practice of evaluating application vulnerability is generally slow. This method relies on injecting faults into a large amount of fault injection points to ensure statistical significance, which requires a significant amount of resources and time. Sometimes, it can take days to evaluate the vulnerability of a single program element (e.g., a data structure) for an individual application. Given the global nature and frequent usage of the collectives by applications, a random fault injection method exacerbates the evaluation cost because of large numbers of fault injection points for the collectives. For a *High Performance Computing* (HPC) system, particularly production and leadership supercomputers, it is not practical to make available the hardware resources for resiliency research for an extended period, because of negative productivity implications. Hence, the fast evaluation methods are paramount to study the application sensitivity to faulty collective communications.

In this paper, we present techniques to address these challenges, and employ them to study and characterize application and benchmarks' sensitivity to faulty collectives.

The major contributions of the paper are as follows:

- We introduce a novel and fast fault injection method to study the impact of faulty collective communications on applications. This method significantly reduces the exploration space for fault injection, hence providing a practical and efficient solution for resilience research.
- We reveal implicit correlations between the collective semantics, application context, critical application features, and application response to faulty collective communications. The discovery of these correlations opens new opportunities to enable fast exploration of application sensitivity to faults. To the best of our

knowledge, this is the first exploration of using implicit application information to derive application sensitivity.
- We introduce a supervised learning ML approach based on the implicit correlations between collective operations and application features to predict the fault sensitivity. Besides predicting the fault sensitivity, this approach also reveals the application features affecting the application sensitivity.
- We develop a fault injection tool, FastFIT, to implement our fault injection method. Using FastFIT, we study the sensitivity of representative benchmarks and petascale capable application to faulty collectives. The results show that we can eliminate more than 97% of fault injection points while not sacrificing the quality of sensitivity results.
- We statistically generalize the application sensitivity to faulty collective communications, and provide a correlation between application features and fault sensitivity of applications.

II. Basic Fault Injection Methodology

To inject faults into collective communications, we follow the traditional random fault injection methodology [4], [5], [7], [8] and statistically quantify the application sensitivity. In particular, we inject the faults randomly into the input parameters of the collective interface. For each fault injection test, the injected fault is manifested by one bit flip in one of the input parameters. The input parameters for MPI collectives typically include the send/receive buffer address, number of data elements, data type, communication destination, message tag, and communicator. To emulate the fault occurrence in the data buffer of the collectives, we flip one random bit in the data buffer. We do not inject faults into the data buffer address as the sensitivity to these faults are obvious. These faults can easily cause catastrophic results (e.g., segmentation fault, or application abort).

The application response to a faulty collective communication is classified into six types as summarized in Table I. Except for *INF_LOOP*, we do not consider the performance impact caused by the faulty collective communication. This means that *SUCCESS* does not have any performance indication. Hence, our fault injection study is focused on the impact of faults on application execution correctness without any performance concerns. In addition, except *SUCCESS*, all other five types of responses are regarded as an *error*. In the rest of the paper, the term *error rate* counts the occurrence frequency of these five types of responses. It needs to be noted that we only focus on the soft errors, and we do not focus on fault recovery as it is not within the focus of the paper.

A typical deployment of scientific applications in extreme scale systems can involve a huge number of MPI processes. It is quite common that the application has many MPI call sites, and each site is invoked multiple times during the application execution for collective communication and synchronization. Each invocation of an MPI collective call site is a potential *fault injection point*. Given a large amount of MPI processes, MPI call sites and invocations, we have to explore a huge fault injection space to study the application sensitivity to faulty collectives. For example, for a small-scale LAMMPS deployment (molecular dynamics simulation) [10] with 1024 processes and an input problem of rhodopsin protein simulation, there are 618,496 fault injection points. For each fault injection point, we need to perform a large number of fault injection tests to ensure statistical significance. In our case, we use at least 100 fault injection tests at each fault injection point. This translates to at least 61,849,600 fault injection tests for LAMMPS. Hence, reducing fault injection points can significantly accelerate the application sensitivity evaluation.

Table I: Application response to the fault injection in collective communications

Abbreviation	Notes
SUCCESS	The program exits without error and generates the same result as the execution without fault injection
APP_DETECTED	The program exits with error reported by the program itself
MPI_ERR	The program exits with error reported by the MPI environment
SEG_FAULT	The program exits with segmentation fault error
WRONG_ANS	The program exits but generates results different from those of the execution without fault injection
INF_LOOP	The program does not exit and is killed because of timeout

III. Fast Fault Injection and Sensitivity Analysis

The goal of FastFIT is to reduce the fault injection points for fault injection study, in addition to aiding application sensitivity study. Towards this goal, we use the following three methods to prune fault injection points and enable fast sensitivity analysis.

A. Semantic Driven Fault Injection

This method leverages MPI collective operations' semantics to reduce potential fault injection points. An MPI collective operation is a group communication operation, with all MPI processes of the communicator participating in the communication. The communication pattern of each of this process is not necessarily the same. In the case of rooted collectives such as MPI_Bcast, MPI_Reduce, and MPI_Scatter, one process acts as a root process and other processes act as non-root process. The root process is either source (MPI_Bcast, MPI_Scatter) or destination (MPI_Reduce) of the data. It sends the data to all non-root processes and waits for an acknowledgment from the non-root processes, or receives the data from all non-root processes. The root process has a different communication pattern compared to the non-root MPI process, while all non-root MPI processes have the same communication pattern. Leveraging this semantics for MPI_Reduce, MPI_Scatter,

and MPI_Bcast, we inject faults into root process and one representative non-root process for each participating communicator. This observation eliminates the need for injecting faults in non-root processes, except one. In the case of non-rooted operations such as MPI_Allreduce, MPI_Allgather, MPI_Alltoall/v/w, all processes have the same communication pattern. So, we inject faults only into one representative process for each participating communicator.

Employing the above methodology to study the application sensitivity to faulty collectives is not sufficient. The application can still respond differently even if the communication patterns of MPI processes are the same, because the MPI processes can have differences in computation patterns leading to the collective communication. To address this issue, we collect application function call graphs and communication traces for each MPI process throughout the application, and then compare their similarity. If two MPI processes have the same call graphs and traces, then they are empirically treated as equivalent. Among a group of MPI processes, only if they have the same computation and communication pattern, we choose one of them as a representative for the sensitivity study.

Figures 1 and 2 justify the effectiveness of our methodology with two NAS parallel benchmarks, LU and FT. LU and FT are configured with a problem class of B and 32 MPI processes. In each run, one bitwise fault is injected into an input parameter of MPI collective. We repeat the same procedure for all parameters of all MPI collectives. For each fault injection point, we perform this random fault injection tests for 100 times. The location of the faulty bit is selected randomly and uniformly in memory of the injected parameter. 100 random fault injection tests are sufficient to cover as many cases as it might appear.

Figure 1 shows the fault injection results for an MPI_Allreduce in LU. For MPI_Allreduce in LU, all MPI processes are regarded as equivalent, and we randomly choose two MPI processes (labeled as rand1 and rand2 in the figure). The figure shows that the application displays very similar sensitivity between the two MPI processes. Figure 2 shows the sensitivity for MPI_Reduce in FT, which shows a different behavior compared to MPI_Allreduce. For MPI_Reduce, we choose the root process and a random non-root process. The figure reveals a different sensitivity between these two MPI processes.

B. Application Context Driven Fault Injection

This method leverages the application context information at fault injection points to further reduce the fault injection points. The method is based on the following observation. A specific MPI collective call site can be repeatedly invoked, and the call stacks of different invocations can vary greatly. However, the invocations with the same call stack, once corrupted, can result in the similar application response. The call stack here refers to the active functions of an application

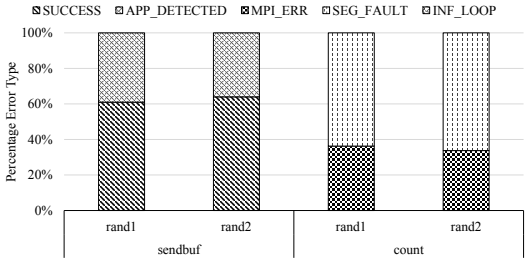


Figure 1: Results of injecting faults into two "equivalent" MPI processes of an MPI_Allreduce collective in LU.

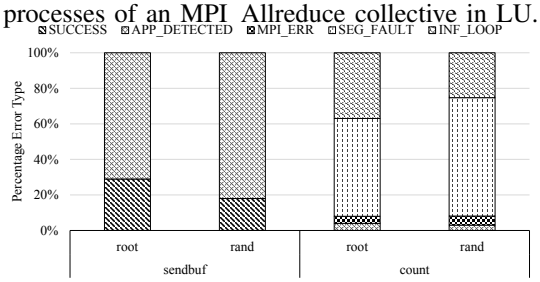


Figure 2: Results of injecting faults into the root and a non-root MPI process of an MPI_Reduce in FT kernel

at a fault injection point. The *same call stack* means that the active functions are the same and called in the same order, but their function parameters may not necessarily the same.

Our observation is clearly pronounced in our wide fault injection tests. Figure 3 shows the results for some of these tests. We choose one MPI_Allreduce call site in an MPI process of LAMMPS. This call site is invoked *107* times, and we choose 100 of them which have the same call stack. For each fault injection test, we inject one bit fault into the data buffer. We perform 100 fault injection tests for one invocation and 10,000 overall for 100 invocations. The figure shows the error rate distribution for those 100 invocations. We can see that the error rate for most of these 100 invocations is focused in a limited range - 25%-35%. This demonstrates that for 100 invocations the application sensitivity is similar. If we use a Gaussian function to emulate the distribution of error rate, we find that the distribution follows a Gaussian distribution with an average error rate of 29.58 with a relatively low standard deviation 7.69. This further justifies the feasibility of our approach.

The rationale behind our methodology and above test

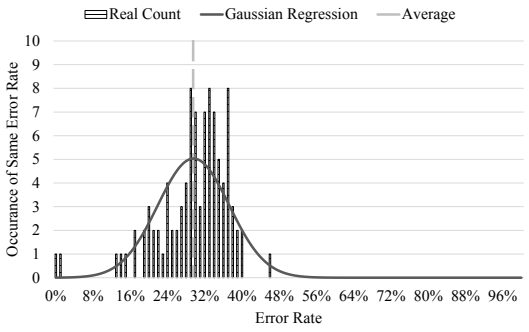


Figure 3: Error rate distribution for 100 invocations of MPI_Allreduce with the same call stack in LAMMPS

results is that the same call stack between different invocations may indicate the same application context. The fault occurred in any two application contexts that are similar often results in the same application response. Based on this methodology, we can significantly reduce the fault injection points. In particular, we can just choose one representative invocation to represent all other invocations that share the same call stack. Our results show that we can reduce 87.6% of fault injection points for LAMMPS by just leveraging the application context information.

C. Machine Learning Driven Fault Injection

We employ a ML based approach to predict the application sensitivity to specific fault injection points. This approach is based on our observation that the application features at the fault injection point have implicit correlations with application sensitivity. The rest of the section provides details of the methodology and the ML algorithm. A thorough study to reveal the correlations between application features and application sensitivity will be shown in Section V-D. The prediction accuracy will also be shown in that section.

Our ML approach works as follows. We use *Random Forest* (an ML algorithm) to train and establish the prediction model. The prediction model is constructed by performing only a limited number of fault injection tests. During the tests, we collect the application features and application responses, and feed them into the random forest algorithm to build the model. Then, with few more tests, we verify the model prediction accuracy by comparing the model prediction results and measured application responses. If the model prediction accuracy does not achieve an expected target threshold, we repeat this process until we reach the target threshold. Once the model is established, for a specific fault injection point, we collect a set of application features at the fault injection point as input and predict the application response using the prediction model.

Application features : We use the following application features to train the model and make the prediction.

1) MPI Collective Type (Type): The communication pattern impacts the application sensitivity to faults. For example, root versus non-root, data source versus data destination, and so on.

2) Execution Phase (Phase): The execution of an application usually consists of multiple phases, such as input, initialization, compute and finalization. Depending on the location of fault injection, the application can display different sensitivity. This correlation between the execution phase and application sensitivity is also revealed in a previous work [7].

3) Error handling code (ErrHal): The collective communication is widely employed for error handling, especially to examine if some critical variables have a consistent value across all MPI ranks. For example, LAMMPS uses MPI_Allreduce frequently to check if errors happen between the MPI processes. In fact, over 40.32% of all MPI_Allreduce calls in LAMMPS are used for error handling. The faulty collective communication in the error handling code has a drastically different impact on the application sensitivity than those in the regular application code.

4) Number of invocations (nInv): The number of invocations of a particular MPI collective indicates the importance of that collective communication. A large number could mean that the application is highly sensitive to the correctness of this collective communication.

5) Average call stack depth (StackDep): The call stack depth at an MPI communication call site is defined by the depth of nested functions starting from the main function. When the call stack goes deeper, the application runs into a more complicated stage which frequently causes higher sensitivity. Hence, the call stack depth can work as an indicator to application sensitivity.

6) Number of different call stacks (nDiffStack): An MPI communication call site can be invoked many times, and each site can have a different call stack. The number of different call stacks is an indicator of varying application behaviors, hence impacting the application sensitivity.

ML algorithm : The random forest is an ensemble learning algorithm that uses a multitude of decision trees. The decision tree is a predictive model that maps an observation (the application features in our case) about an item (the application sensitivity in our case) to a conclusion about the item's target value (the qualification of application sensitivity in our case). A decision tree can be used to explicitly describe the data and represent decision making. The decision of a random forest is a majority decision based on its decision trees' decisions.

Figure 4 displays an example of a decision tree. This decision tree is established by our practical fault injection tests and model training procedure. The leaf nodes of the decision tree are the application sensitivity decisions (i.e., low, medium-low, medium-high, and high). These four levels of application sensitivity correspond to four different error rate levels - e.g., Low - 0%~25%, Medium-low - 25%~50%, Medium-high - 50%~75% and High -75%~100%. We do not use a single number to quantify the application sensitivity. Instead, we use the four levels to qualify it for the following two reasons: (1) It is challenging to use a ML approach to accurately predict the error rate because of the statistical nature of these algorithms. Based on our own experience with ML algorithms, the prediction accuracy cannot reliably indicate a specific error rate, but can indicate the error rate level very well. (2) We often take the decision of using a fault-tolerant mechanism based on the qualification of application sensitivity. For example, if an MPI communication is very critical and also results in more than 20% error rate, then we decide to enforce fault-tolerance. Hence, the qualification instead of quantification is more helpful for the decision

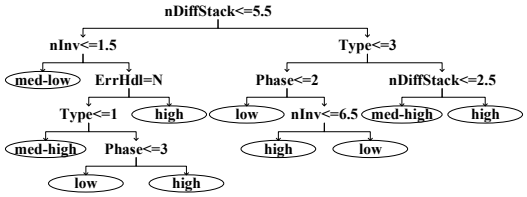


Figure 4: An example of a decision tree

making (i.e., a resilient system design). Note that, although we target at predicting the application responses in terms of four levels of application sensitivity in this subsection, our methodology is applicable to make prediction for any levels of application sensitivity.

The non-leaf nodes of the decision tree represent the application features (i.e., *Type, Phase, ErrHal, nInv, nDiffStack,* and *StackDep*). The application feature must be represented by numerical values to facilitate the tree construction. For the features *Type, Phase,* and *ErroHal*, we assign specific numerical values to each possible type, phase, and error handling code. Based on the numerical values of the non-leaf nodes and given a set of input application features, we decide the application sensitivity. For example, an input with $nDiffStack = 8$, $Type = 2$ and $Phase = 1$ will result in a conclusion of *low* in Figure 4.

Prediction accuracy threshold : As described at the beginning of this section, the training of prediction model is controlled by the prediction accuracy threshold. The fault injection test results are fed into the model for training and verification, until the prediction accuracy reaches the threshold. In the worst case, we could run out of all fault injection points to train the model without reaching the desired prediction accuracy. In this case, our method training and verification is equivalent to the tradition fault injection method. As a matter of fact, in all our fault injection studies, the model can reach our desired prediction accuracy before we run out of the fault injection points.

Furthermore, there is a tradeoff between the prediction accuracy and pruning of fault injection points. A high accuracy threshold means that we need more fault injection results to train the model. On the other hand, a smaller accuracy threshold can lead to faster model training and require fewer fault injection results. The tradeoff relation between accuracy and reduction in fault injection points is shown in Figure 6 and discussed in Section V-D.

It is noticeable that the reduction of fault injection points brought by the ML is a bonus to its key contribution of revealing the implicit correlation between application features and its sensitivity. We only explore the tradeoff when users' threshold is satisfied.

IV. FastFit Implementation

A. Architecture

The application sensitivity study based on FastFIT has three main phases i.e., profiling, learning, and injection. Figure 5 shows various components of FastFIT and their interaction during a typical fault injection test.

During the profiling phase, FastFIT collects the application information required for fault injection and required by *Semantic Driven Fault Injection* (Section III-A) to eliminate unnecessary fault injection points.

The learning and injection phases of FastFIT together constitutes the implementation of *Machine Learning Driven Fault Injection* (Section III-C) method. The feedback loop in Figure 5 shows the coordination. During the injection phase, FastFIT randomly chooses the fault injection points and performs the fault injection. The results are then used in the learning phase to train the prediction model. These two phases are repeated until the prediction model reaches a user-expected accuracy threshold or until it runs out of fault injection points. If the prediction model is accurate enough and we still have untested fault injection points, we will use the prediction model to estimate application sensitivity for those untested fault injection points. More detailed descriptions for these three phases are as follows:

B. Profiling Phase

During the profiling phase, FastFIT profiles the application. The problem used for profiling is same as the problem used for fault injection test. This ensures the application features are consistent between a profiling run and fault injection run of application. The profiling overhead is a one time cost as the collected information can be used for any number of fault injection campaigns. FastFIT collects three different types of profiles.

- Communication Profile: The communication profile information such as collective call sites, collective types, how often the collectives are used, and number of collective invocations are obtained using *mpiP* [11]. This profile is used to identify and potentially eliminate the fault injection points.
- Call Graph Profile: The call graph of the application represents the control path taken by the application. It is obtained using the profiling tools such as vanilla *Callgrind* and *gprof*.
- Call Stack Profile: After collecting the call graph, FastFIT uses the *backtrace()* function available in the standard C library to obtain the call stacks at fault injection points. Based on the similarity of call stacks, it chooses a representative fault injection point.

Using the above profile information, the application code is transformed to enable fault injection at those points, which is performed during the injection phase.

C. Injection Phase

The injection phase includes two modules, *Config Generation* and *Fault Injection*, shown in Figure 5. The two modules are controlled by a set of environment variables listed in

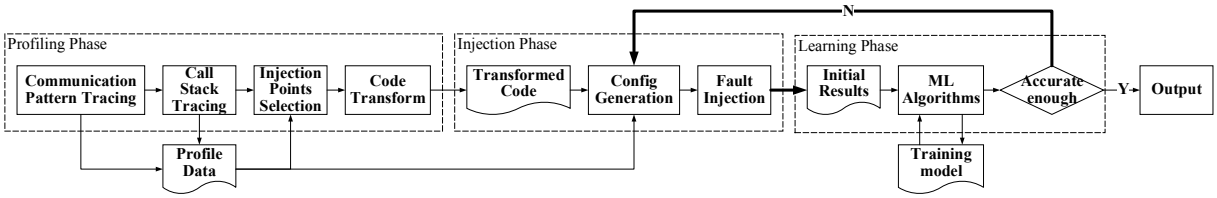


Figure 5: Various components of FastFIT and the interaction of components during a profiling and fault injection campaign

Table II: Configurable parameters for FastFIT

Abbreviation	Width	Notes
NUM_INJ	unlimited	Number of injected faults
INV_ID	3	Id of injected invocation
CALL_ID	3	Id of MPI collective
RANK_ID	unlimited	Id of injected rank
PARAM_ID	1	Id of injected parameter

Table II. At runtime, the *Config Generation* module reads the environment variables and drives the *Fault Injection* module to perform random bit flips on the representative fault injection points.

D. Learning Phase

During the learning phase, the random forest module uses the fault injection results to train the prediction model. With additional tests, the model's prediction accuracy is verified. If the desired accuracy is not achieved, the injection phase is triggered continuing the cycle until the desired accuracy for results is achieved or all fault injection points are tested. FastFIT is not tied to the random forest algorithm. It can be replaced by other machine learning algorithms, if required.

V. Evaluation

In this section, we evaluate how effectively our methods can reduce the fault injection points, characterize the application sensitivity, and evaluate the prediction accuracy of our methods.

A. Setup

The experiments were conducted on *Titan* housed at *Oak Ridge Leadership Computing Facility* (OLCF). Each compute node of Titan includes one 16-core 2.2GHz AMD Opteron 6274 (Interlagos) processor. It uses a 3D torus network built from Gemini application-specific integrated circuits (ASICs).

We perform fault injection tests and sensitivity analysis with NPB suite, and a realistic scientific application LAMMPS. We use 32 MPI processes in all tests. For experiments with NPB, we use IS, FT, MG and LU with input class B. For LAMMPS, we use the rhodopsin protein simulation as an input problem.

B. Reduction of Fault Injection Points and Fault Injection Tests

By exploring the equivalence between fault injection points and taking advantage of correlations between the application features, we effectively reduce the exploration space for fault injection. Table III summarizes the reduction of fault injection points for NPB and LAMMPS. The evaluation of the prediction accuracy is shown in Section V-D. For NPB, we only apply the *Semantic Driven Fault Injection* and *Application Context Driven Fault Injection* methods, as the number of injection points are already small after applying the two methods.

For LAMMPS, we employ *Semantic Driven Fault Injection* and *Application Context Driven Fault Injection* methods to reduce the fault injection points and conduct the fault injection campaign. Further, we reduce the fault injection tests required for the fault injection campaign by using *Machine Learning Driven Fault Injection*. ML techniques predict the fault impact using the fault model trained by the fault injection campaign results, rather than requiring the need to conduct fault injection experiments on all points. With ML, we reduced the fault injection tests required by half (53.33%). Overall, we reduce the fault injection points by over 97%. The ML algorithm takes only several seconds to learn in all our experiments. Thus, the overhead of learning phase is negligible compared to the time taken by fault injection tests.

As discussed in Section III-C, the prediction accuracy threshold is directly related to the reduction of fault injection points with ML. To investigate this relation, we vary the prediction accuracy threshold during the model training, and calculate the reduction of fault injection points after applying the model. The results for LAMMPS are shown in Figure 6. In the figure, the prediction accuracy threshold is represented by a percentage number, which refers to the percentage of expected correct prediction cases. As shown in the figure, the reduction of fault injection points goes down as we use a higher prediction accuracy threshold. In the best case (the threshold is 45%), we can reduce the fault injection points by more than 80%.

In general, there is a balance between fault injection reduction and prediction accuracy. A higher prediction accuracy threshold, though results in a more accurate prediction model, requires more fault injection tests for model training and has less opportunities to reduce fault injection. We choose 65% in our fault injection campaign as it strikes a good balance between reducing the fault tolerance points and prediction accuracy.

Table III: Reduction ratio after applying the three techniques with FastFIT

	MPI	App	ML	Total
IS	96.88%	90.00%	NA	99.69%
FT	96.31%	95.24%	NA	99.78%
MG	96.09%	90.70%	NA	99.64%
LU	96.35%	40.00%	NA	97.81%
LAMMPS	97.24%	87.58%	53.33%	99.84%

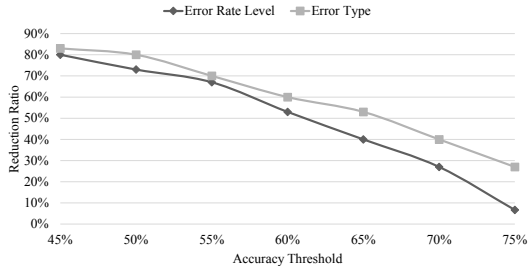


Figure 6: The relationship between prediction accuracy threshold and reduction in fault injection points.

C. Fault Injection Results

In this section, we present fault injection results using FastFIT. The goal is to statistically generalize application sensitivity to faults in collective communications and provide guidance for designing the resilient collectives. During the fault injection tests, we inject faults into the data buffer of collective communications (if there is any data buffer), unless indicated otherwise.

Impacts of Fault Injection on NPB Benchmarks (General Results): Figure 7 shows the fault injection results for all collective communications in four NPB kernel benchmarks. In this experiment, we inject the faults into the kernels, and learn the error type response of the benchmark.

We can see that *INF_LOOP* has the least occurrence in all benchmarks. This is consistent with the results of LAMMPS shown later. Also, a significant portion of errors are *MPI_ERR*, which are the errors reported by the MPI implementation. This is especially true for FT (46% of all injected faults). This indicates that a resilient communication library implementation is very important for achieving application resilience. Further, we can observe that the application only detects a small percentage of faults as indicated by *APP_DETECTED*. This may suggest that NPB's error handling code is not effective at capturing the errors caused by faulty collective operations. Last, we observe that *SEG_FAULT* is a very common application response, only second to *SUCCESS*. The IS (44%), MG (28%), and LU (24%) kernels exhibit this behavior. We expect that a mechanism that effectively responds to *SEG_FAULT* can significantly improve the application resilience.

Impacts of Collective Communication Diversity on NPB Benchmarks: We further look into the fault injection

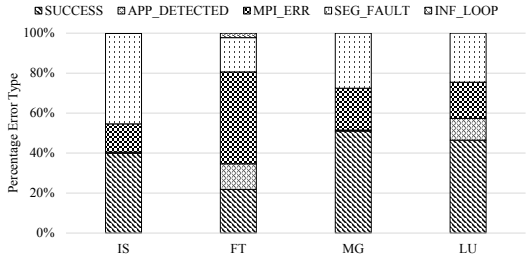


Figure 7: NPB benchmark's response in error types, when faults are injected into NPB's MPI collectives

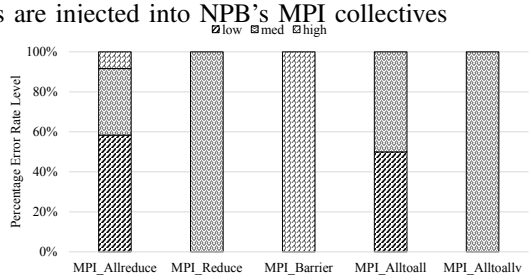


Figure 8: NPB benchmark's response in error rate levels, when faults are injected into NPB's MPI collectives

results and analyze the error rate level distribution for different collective communications. Figure 8 shows the results. There are three error rate levels (i.e., low, med, and high) in the figure. Given an MPI collective communication in a benchmark, the error rate level is defined according to how many instances of this communication causes application error responses (i.e., *APP_DETECTED*, *MPI_ERR*, *SEG_FAULT*, *WRONG_AN*S, and *INF_LOOP*). The low, med, and high error rates correspond to 15%, 15%-85%, and 85% of the communication instances respectively.

The results show that faulty MPI_Reduce and MPI_Barrier have more negative impact on the application than the faults in other collectives. MPI_Alltoallv causes the least damage to the applications. This sensitivity variance indicates that there is a need for adaptive fault-tolerance mechanism rather than a single uniform fault-tolerant mechanism across all collectives.

Impacts of Erroneous Input Parameters of Collective Communications on NPB Benchmarks: The MPI collective communication interface has many input parameters. To study the impact of faults in different parameters of the collective operations, we inject faults into each parameter and measure the application response. Figure 9 shows the results for a specific collective, MPI_Allreduce. MPI_Allreduce has 6 input parameters including *sendbuf, recvbuf, count, datatype, op*, and *comm*. The errors in *recvbuf* have a little impact on the application, as faults are injected before the collective call is enforced, and it is likely that the fault bits are overwritten by the communication library. The application is more sensitive to faults in *sendbuf* compared to faults in *recvbuf*. However, a majority of faults is detected by the application. Furthermore, the faults in other parameters

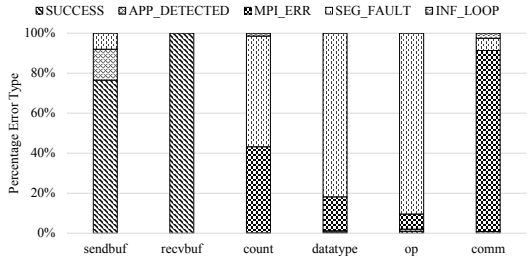


Figure 9: NPB benchmark's response in error types, when faults are injected into the parameters of NPB's MPI collectives.

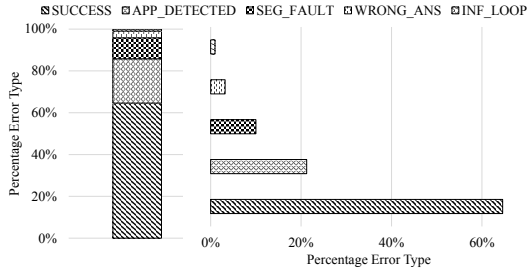


Figure 10: LAMMPS benchmark's response in error types, when faults are injected into LAMMPS' MPI collectives

(*count*, *datatype*, *op* and *comm*) have a high impact on the application as they often cause *SEG_FAULT*. Hence, those parameters demand stronger protection to improve the application resilience.

Impact of Fault Injection on LAMMPS (General Results): Figure 10 shows the fault injection results for all collectives in LAMMPS. LAMMPS frequently uses MPI_Allreduce, MPI_Bcast, MPI_Barrier, and MPI_Allgather. When faults are injected into these collectives, *SUCCESS* is the most common application response. About 65% of fault injection tests does not cause the negative application response. *APP_DETECTED* is the second most common response of the application. This indicates that LAMMPS has more mature error handling code compared to NPB, where lower percentage of errors are *APP_DETECTED*. Though *SEG_FAULT* is not very common (about 10%) in LAMMPS comparing with NPB, it is still very significant. This also demonstrates the necessity for protection against *SEG_FAULT*. *INF_LOOP* has the least occurrence, which is consistent with the NPB results. *WRONG_ANS* is also not a common response in case of LAMMPS. Given the fact that LAMMPS employs a Monte Carlo model which is a statistical-based approach, this may not be a surprising result.

Impact of Collective Communication Diversity on LAMMPS: Figure 11 shows the error rate level distribution for all collectives in LAMMPS. Similar to NPB, the faulty MPI_Barrier has a lethal effect on LAMMPS: the faults injected in MPI_Barrier result in large percentages of *high* and *med* error rates. However, the error rate levels for other

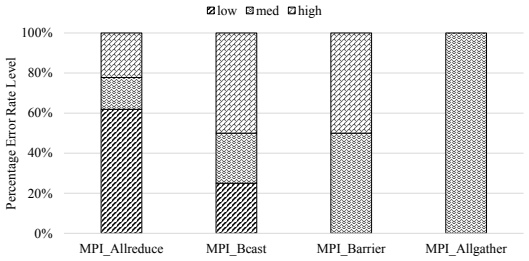


Figure 11: LAMMPS benchmark's response in error rate levels, when faults are injected into LAMMPS' MPI collectives

collectives are not skewed towards one direction. Also, we observe that MPI_Allreduce has a *low* error rate. This is a surprise, given that MPI_Allreduce is frequently used (more than 84% of collectives) by LAMMPS. This low error rate for MPI_Allreduce is also pronounced in NPB results, where more than 63% of collectives are MPI_Allreduce.

D. Accuracy Evaluation of Machine Learning-Based Sensitivity Prediction

To evaluate the effectiveness of our ML approach, we train our ML model using some fault injection tests as a training set. We randomly divide the training set into two class - one class for training and the other for testing. We repeat the random division of the training set for five times to get the average results.

We employ our model to predict error type (*SUCCESS*, *APP_DETECTED*, *SEG_FAULT*, *WRONG_ANS*, *INF_LOOP*) and error rate level (e.g., low, med, or high). Figure 12, shows the prediction accuracy of ML for predicting the error type. In the figure, we can observe the prediction accuracy for error type *SUCCESS*, *APP_DETECTED*, *SEG_FAULT*, and *WRONG_ANS* is 86%, 80%, 47%, and 75%, respectively. The model predicts *SUCCESS*, *APP_DETECTED*, and *WRONG_ANS* with high accuracy. However, it has a low prediction accuracy for error type *SEG_FAULT*. We suspect that *SEG_FAULT* has weak correlation with the chosen application features. To improve prediction accuracy, we have to include more application features in the ML model.

Figure 13 shows the prediction accuracy for the prediction of 2 or 3 error rate levels. We divide the error rate range evenly into 2 or 3 levels for the two cases respectively. For the prediction with 2 error rate levels the ML model correctly classifies over 80% of fault injection points as either highly sensitive or not highly sensitive. For the prediction with 3 error rate levels, the ML model can correctly predict over 76% of fault injection points as low sensitive and over 66% as high sensitivity. Although we only show prediction results for 2 or 3 error rate levels, our model can support prediction for any number of error rate levels.

Quantifying Sensitivity of Application Features:

This section justifies that there are implicit correlations between application features and application sensitivity.

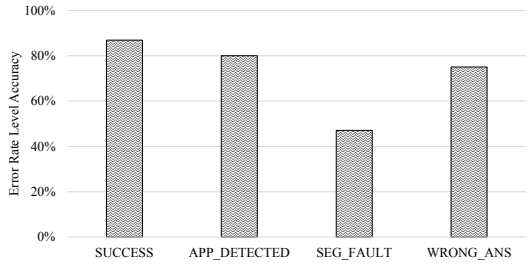
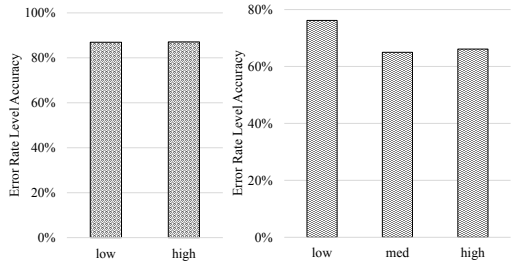


Figure 12: Error type prediction accuracy



(a) Two levels (b) Three levels

Figure 13: Error rate level prediction accuracy

Those correlations lay foundation for our machine learning-based prediction. We quantify the correlation based on a well established equation (Equation 1). This equation calculates the correlations between each of application feature (see Section III-C) and error rate level.

$$Correlation(X,Y) = \frac{1}{2}(\frac{\sum(x-\bar{x})(y-\bar{y})}{\sqrt{\sum(x-\bar{x})^2(y-\bar{y})^2}} + 1) \quad (1)$$

The variables X and Y represent quantified application feature and error rate level respectively. x and y are the examples in series X and Y, and $\bar{x}$ and $\bar{y}$ are the averages of X and Y. A correlation value close to 1 means that application sensitivity and the corresponding application feature vary similarly. Hence the application feature has a strong indication to the application sensitivity. A value close to 0 means that they vary in an opposite way. A value of 0.5 means that the application feature does not affect the application sensitivity.

Table IV shows the correlation results. We can see that the features such as application phases (e.g., initial phase and input phase) and error handling code have a strong correlation with the application sensitivity. This result makes sense. In particular, the initial phase and input phase determines the application input problems and configurations, hence they are vital to the correctness of application execution. The error handling code also has a direct impact on application tolerance to the errors, hence they are also well correlated with application correctness. Although the other application features are less correlated, we include them for training the model as including them can maximize the prediction accuracy. Note that including them does not increase the model training cost, because all of those features can be collected at the profiling phase together.

VI. Evaluation Summary

FastFIT significantly reduces the number of fault injection points. For NPB and LAMMPS, it reduces the fault points by 99.23% and 99.84%, respectively as shown in Table III. We use three salient techniques in FastFIT, and verify their effectiveness in Figures 1 to 3, and Figures 12 and 13, and Table IV.

Further, we use FastFIT to study the sensitivity of NPB and LAMMPS to faults in collective operations. Some of the important results are summarized as follows:

NPB: The NPB kernels have varying responses to faults in collectives as seen in Figure 7. IS mostly crashes in the presence of faults. However in the case of FT, a high percentage of errors caused by faults are detected by the MPI implementation. For MG and LU, the results show that a high percentage of faults either crash the kernels or the errors caused by faults are detected by the MPI implementation. The difference in response is partially because of difference in how different collective operations respond to faults. Figure 1 and Figure 2 shows the response of MPI_Allreduce and MPI_Reduce to faults. Also, as seen in the Figure 9, where the fault occurs in the collective parameters also impacts the fault response.

LAMMPS: A large percentage of fault seems to have no impact on the application execution. Rest of faults ($\tilde{4}0\%$), LAMMPS either crashes the application, produces wrong result, or results in an application abort because of application detected errors. Some of the faults (21.24%) are detected by the application's error handling code.

ML Techniques: For LAMMPS and NPB, using ML techniques, with high accuracy, FastFIT predicted the error rates and error types, when a fault is injected into the collective operations used by these kernels. The prediction accuracy was summarized in Figure 12 and Figure 13. Further, the ML techniques revealed the correlation between application features and fault sensitivity. The results are summarized in Table IV. It reveals that faults in applications phases and error handling code have a strong impact on fault sensitivity.

VII. Related Work

Random fault injection has been used as an approach to explore the impact of errors to HPC applications. Bronevetsky and de Supinski [4] perform fault injection in random locations of memory stack or heap based on manual instrumentation. Debardeleben et al. [8] attempt to study the applications' sensitivities at the instruction level. They take advantage of the QEMU virtual machine to achieve injection at specific assembly instructions. Naughton et al. [9] propose a fault injection framework targeting at API-level failures for memory (slab errors and page allocation errors) and disk I/O errors. Casas et al. [12] inject faults into instructions' output according to the LLVL typed byte code of algebraic multi-grid solver. Li et al. [7] use a binary instrumentation-based

Table IV: Correlation between application specific features and error rate level

	Init Phase	Input Phase	Compute Phase	End Phase	ErrHdl	Non-ErrHdl	nInv	nDiffGraph	StackDepth
LAMMPS	0.56	0.69	0.3	0.49	0.64	0.36	0.41	0.47	0.37

mechanism to inject faults randomly into data structures at various data segments.

Our work is different from the above work from three perspectives. First, we focus on reducing fault injection points and enabling fast evaluation instead of just focusing on studying application vulnerability. Second, we predict the application sensitivity to faults, reducing the amount of fault injection tests. Also, this approach enables understanding the co-relation between the application features and fault sensitivity. Third, we particularly study collective communications, which is different from those research targets in the previous work. They are particularly challenging because of their global nature.

Sastry et al. [13] and Xu et al. [14] also aim to reduce the fault injection points. They choose a small subset of injection points to inject faults after static and dynamic program analyses. Our method is different from them, since we rely on implicit application information and statistical correlation to prune fault injection points. Our method is general and portable, and does not require extensive compiler support.

VIII. Conclusions

This paper presents FastFIT, a fast fault injection and sensitivity analysis tool, and studies the sensitivity of a petascale capable application and benchmarks to faulty collective communications. We also present a series of methods that explores the use of implicit application information to significantly reduce the potential fault injection points. The experimental results show that FastFIT, which is based on these methods, can reduce more than 97% fault injection points required for a fault injection study for representative benchmarks and a scientific application. Our ML techniques, with high accuracy, predicted the error types and error rates of applications in presence of faulty collectives. Further, using FastFIT, we statistically generalize the application sensitivity to faulty collective communications, and present correlation between faults and application features. Even though these techniques were tested only on the collective operations in this paper, it can be applied to other programming elements of an HPC application, which is a part of our future work.

IX. Acknowledgement

This research used resources of the Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research is also partially supported by UC Merced Startup Fund, and is supported in part by a research grant from Huawei Technologies Co, Ltd., the US National Science Foundation under Grant No. CCF-0937877 and CNS-1162540.

References

[1] R. Rabenseifner, "Automatic Profiling of MPI Applications with Hardware Performance Counters," in *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 1999.

[2] ——, "Optimization of Collective Reduction Operations," in *International Conference on Computational Science 2004 Springer-Verlag LNCS 3036*, 2004.

[3] J. Hursey and R. L. Graham, "Preserving Collective Performance across Process Failure for a Fault Tolerant MPI," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011.

[4] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, 2008.

[5] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the impact of soft errors on iterative methods in scientific computing," in *Proceedings of the International Conference on Supercomputing*, 2011.

[6] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz, "Fault Resilience of the Algebraic Multi-grid Solver," in *Proceedings of the International Conference on Supercomputing*, 2012.

[7] D. Li, J. S. Vetter, and W. Yu, "Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[8] N. DeBardeleben, S. Blanchard, Q. Guan, Z. Zhang, and S. Fu, "Experimental Framework for Injecting Logic Errors in a Virtual Machine to Profile Applications for Soft Error Resilience," in *Workshop on Resilience in High Performance Computing in Clusters, Clouds and Grids*, 2011.

[9] T. Naughton, W. Bland, G. Vallee, C. Engelmann, and S. Scott, "Fault injection framework for system resilience evaluation: fake faults for finding future failures," in *Proceedings of the Workshop on Resiliency in High Performance Computing*, 2009.

[10] "LAMMPS Molecular Dynamics Simulator," http://lammps.sandia.gov/bench.html.

[11] "mpiP," http://mpip.sourceforge.net/.

[12] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz, "Fault resilience of the algebraic multi-grid solver," in *Proceedings of International Conference on Supercomputing*, 2012.

[13] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *ACM SIGARCH Computer Architecture News*, 2012.

[14] Xin Xu and Man-Lap Li, "Understanding soft error propagation using Efficient vulnerability-driven fault injection," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012.