LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-643269

A Study of Application-Level Recovery Methods for Transient Network Faults

I. Laguna, E. A. Leon, M. Schulz, M. Stephenson

September 3, 2013

The International Conference for High Performance Computing, Networking, Storage and Analysis
Denver, CO, United States
November 17, 2013 through November 22, 2013

Disclaimer

A Study of Application-Level Recovery Methods for Transient Network Faults *

Ignacio Laguna
Lawrence Livermore National Laboratory
ilaguna@llnl.gov

Edgar A. León
Lawrence Livermore National Laboratory
leon@llnl.gov

Martin Schulz
Lawrence Livermore National Laboratory
schulzm@llnl.gov

Mark Stephenson
IBM Research Austin
mstephen@us.ibm.com

ABSTRACT

With the increasing number of components in HPC systems, transient faults will become commonplace. Today, network transient faults, such as lost or corrupted network packets, are addressed by middleware libraries at the cost of high memory usage and packet retransmissions. These costs, however, can be eliminated using application-level fault tolerance. In this paper, we propose recovery methods for transient network faults at the application level. These methods reconstruct missing or corrupted data via interpolation. We derive a realistic fault model using network fault rates from a production HPC cluster and use it to demonstrate the effectiveness of our reconstruction methods in an FFT kernel. We found that the normalized root-mean-square error for FFT computations can be as low as 0.1% and, thus, demonstrates that network faults can be handled at the application level with low perturbation in applications that have smoothness in their computed data.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: [Reliability, Testing, and Fault-Tolerance]

Keywords

Resilience; application-level fault recovery; network faults.

1. INTRODUCTION

As high-performance computing (HPC) systems grow in scale and complexity, fault tolerant techniques become crucial to make effective use of these systems. Due to increasing number of components and lower voltages in microprocessor chips, *transient faults* will become commonplace in exascale systems, resulting in reduced system reliability compared to today's petascale systems [1]. An important source of transient faults is the interconnection network. Sources of network errors include circuit faults of network interface controllers and switches, and cable faults due to external noise and electromagnetic interference.

According to the June 2013 edition of the Top500 list, 41% of the systems employ the InfiniBand interconnect. InfiniBand provides several transport services, with different reliability guarantees, including *reliable connection* (RC) and *unreliable datagram* (UD). RC is a connection-oriented transport providing reliable and in-order message delivery. Because a connection between each pair of communicating processes or nodes is required, this transport is not scalable [2, 3, 4]. In contrast, UD provides scalability at the cost of reliability. UD is a connection-less and unreliable transport—messages may be lost during transmission but a single endpoint can communicate with any other endpoint in the system. UD-based MPI implementations can reduce memory usage up to 30 times compared to RC-based implementations [2, 5]. Even though a UD-based MPI implementation provides significant scalability advantages over an RC-based implementation, there is an added cost at the MPI layer due to message retransmissions when network errors occur (MPI is a reliable transport). Providing reliability may significantly increase the overhead of the communication stack [6].

We argue that, for some applications, loosing a number of network packets (application messages are comprised of one or more packets) may not cause a significant deviation in their final outcome. In such scenarios, an application may decide whether a piece of information is critical or not, and whether it should be recovered by retransmission. Applications that can benefit from this approach include those with underlying smoothness in their computation so that their outcomes are not too sensitive to lost data. Examples of these applications include image processing applications (e.g., fast Fourier transform applications) and probabilistic computational methods (e.g., Monte Carlo methods).

In this paper, we propose and evaluate novel recovery methods to cope with network transient faults. We eliminate the need of message retransmissions by providing recovery at the application level—corrupted or lost data is *reconstructed* from non-corrupted data by interpolation. Our recovery model improves the scalability of MPI implemen-

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344 (LLNL-CONF-643269).

ScalA '13 November 17 - 21 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2530268.2530271.

tations by allowing them to use UD transport without incurring retransmissions of faulty messages. We implement our application-level reconstruction policies in a fast Fourier transform (FFT) kernel and evaluate their effectiveness by calculating the normalized root-mean-square (RMS) error.

Our results indicate that application-level recovery can be effective in reconstructing corrupted data with a small error—the normalized RMS error of a faulty FFT run compared to a normal run can be as low as 0.1%. We also show that a simple data reconstruction policy, such as filling missing data with zeros, works surprisingly well, and that an FFT computation is more susceptible from packet errors when the errors occur in complex inputs than when they occur in real inputs.

Our fault injection model is based on real traces of network errors of a large-scale cluster at Lawrence Livermore National Laboratory (LLNL). Our 7-month traces allow us to better understand real-world distributions of transient faults. We found that 93% of InfiniBand ports have a low probability of experiencing transient faults in one-hour periods. However, a few ports (0.29%) experience high error rates of up to a hundred errors per hour. To the best of our knowledge, this is the first study of real-world InfiniBand transient faults in large HPC clusters. Previous studies analyzed network recovery models under theoretical error rates [7].

The main contributions of this paper are:

- The proposal and evaluation of novel and accurate application-level recovery methods for transient network faults.
- A characterization of transient network faults from seven-month traces of a large, production HPC cluster.
- An analysis of the impact of errors on the result of an FFT computation based on different numerical inputs (real and complex).

The rest of the paper is organized as follows: Section 2 provides a brief overview of the InfiniBand architecture and describes our analysis of network errors based on real traces; Section 3 describes our FFT application and our proposed recovery methods; Section 4 describes our fault injection strategy and experimental results; we survey the related work in Section 6 and discuss the implications of this work in Section 5; finally, we conclude in Section 7.

2. ANALYSIS OF NETWORK ERRORS

2.1 InfiniBand Architecture Overview

The main components of an InfiniBand cluster are compute nodes, host channel adapters (HCAs), switches, and links. As Figure 1 illustrates, compute nodes send messages over links and switches to other nodes through an HCA. We define a *port* to be a physical connection point of either a switch or an HCA.

As in other networks, an application message is transmitted as multiple data units at the transport and link layers. We define a transport-layer data unit to be a *packet*, and a link-layer data unit to be a *frame*. Thus, an application message could comprise multiple packets, and a packet could comprise multiple frames.

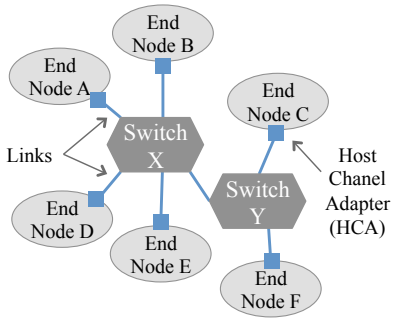


Figure 1: Main components of the InfiniBand architecture.

2.2 Network Errors

We follow conventional definitions of faults and errors. A *fault* is a physical defect, imperfection, or flaw that occurs in the network hardware (e.g., an external interference). An *error*, the manifestation of a fault, causes a deviation from correctness (e.g., a bit change from 0 to 1 in a network packet) [8]. The system logs that we used only capture information about errors. Thus, in the rest of the paper, we refer to faults and errors interchangeably.

We define a *transient fault* to be one that manifests itself one time and disappears after a short period of time. Examples of transient fault manifestations include symbol errors in a frame and dropped packets due to lack of buffers at the receiving port. If we observe errors in a port connected to a node that is down, we treat them as *permanent faults*, and not as transient faults; since the node is down, we will see errors permanently until it is up again.

Network errors manifest themselves at multiple layers of the network stack. For instance, in Figure 1 suppose that node A sends a packet to node F. The packet traverses switch X and switch Y before reaching node F. A local error between node A and switch X may trigger a retransmission of the whole message between A and F.

2.3 Distribution of Errors

2.3.1 Sierra cluster at LLNL

We analyzed error logs from *Sierra*, a production cluster at LLNL, which is interconnected using a two-level fat-tree InfiniBand network. The network is built with 36-port switches on both levels. *Sierra* comprises 1,944 nodes, each with two six-core Xeon processors at 2.8 GHz. We collected network error logs for a period of seven months. Each log records multiple types of errors per InfiniBand port every five minutes. Ports include those in switches and HCAs. *Sierra* has a total of 11,664 ports and 5,832 4X QDR (4x10Gbps) links. Equations 1 and 2 illustrate how we calculate the number of ports and links in *Sierra*; *ASIC* stands for Application-Specific Integrated Circuit (i.e., integrated circuits customized to implement functionality of the network stack).

Table 1: Percentage of total errors per category. T corresponds to transient faults and P to permanent faults.

Error	Percentage	Type	Description
SymbolErrorCounter	73.9505	T	Symbol errors detected in a link.
PortXmitDiscards	21.7302	T, P	Outbound packets discarded because port is down or congested.
VL15Dropped	2.8291	T	VL15 packets dropped due to resource limitations (e.g., lack of buffers)
PortRcvRemotePhysicalErrors	0.6429	P	Packets marked with EBP delimiter. Indicates problem in the fabric.
LinkErrorRecoveryCounter	0.5970	T, P	Link recovery process has completed. Could be caused by node reboots.
LinkDownedCounter	0.1754	T, P	Link error recovery process failed. Could be caused by node reboots.
PortRcvErrors	0.0740	T	Number of packets containing errors due to malformed data or buffer overrun.
PortXmitConstraintErrors	0.0010	T, P	Packets not transmitted because specs failures, e.g., IP version and partition keys.

$$\begin{aligned} 3 \text{ core switches} \times 54\frac{ASICs}{switch} \times 36\frac{ports}{ASIC} + \\ 108 \text{ edge switches} \times 36\frac{ports}{switch} + \\ 1,944 \text{ HCA ports} = 11,664 \text{ ports} \end{aligned} \quad (1)$$

$$\begin{aligned} 1,944 \text{ HCA–edge switch links} + \\ 1,944 \text{ edge switch–core switch links} + \\ 1,944 \text{ core switch internal links} = 5,832 \text{ links} \end{aligned} \quad (2)$$

2.3.2 Error logs

Table 1 shows the major categories of errors and their percentage of occurrence. Based on our definition of faults, we classify them as manifestations of transient or permanent faults. Some errors can be caused by either type. For example, `PortXmitDiscards` errors can be caused because a port is down, a permanent problem, or because of congestion, a transient problem.

Based on Table 1, the most common transient error in *Sierra* is `SymbolErrorCounter` or symbol error (73.95% of the total). According to the InfiniBand Architecture Specification (IBAS), symbol errors are minor link errors in a physical lane and a small number of them is acceptable. The IBAS indicates that the maximum bit error rate should be 10^{-12} or 144 errors per hour for a 4X QDR link (see Equation 3). Ports that experience higher rates could suffer from sa bad port or cable.

$$\begin{aligned} 10^{-12}\frac{errors}{bit} \times 40 \cdot 10^{9}\frac{bits}{sec} \times 60\frac{sec}{min} \times 60\frac{min}{hour} = \\ 144 \text{ errors per hour} \end{aligned} \quad (3)$$

The second most common error is `PortXmitDiscards`; it occurred 21.73% of the time. Since it can originate from permanent faults, we do not study it in detail. The rest of the errors occur very infrequently. Thus, we focus our analysis on symbol errors for the rest of the paper and we use them as a model for our fault injection strategy.

Figure 2 shows the errors-per-hour distribution of transient (symbol) errors across the ports of *Sierra*. We exclude symbol errors from down ports since we do not consider them transient, but permanent faults. Ports (on the X-axis) are sorted by the error rate from high to low and labeled accordingly. The figure only shows rates for the first 2,000 ports. Ports in the range of 1,651–11,642, which correspond to 85.8% of the ports, did not show any errors in the 7-month period. This means that jobs using these ports did not experience transient faults of this type and, therefore, did not need to pay the overhead of transient-fault recovery techniques. We also observe that only two ports (0.017%) showed an error rate higher than the IBAS maximum bit error rate.

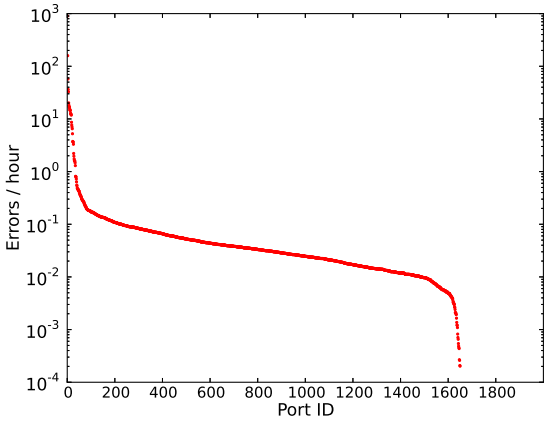


Figure 2: Error rate per port in Sierra. Although the cluster has a total of 11,664 ports, we only show the rates for the first 2,000 ports. Ports in the range of 1,651–11,642 did not experience any error in the 7-month period.

For the rest of the analysis, we focus on errors occurring during one hour window assuming that our application of interest, such as the FFT, runs within that amount of time. Under this scenario, most of the ports experience a low probability of a transient fault; the error rate of 99.7% of the ports is less or equal than 1 error/hour. We observe that there is a small set of ports (only 34) that experience an error rate higher than 1 error/hour. Thus, in the rest of the paper, we study recovery techniques for error rates in the range of one to ten errors/hour to consider the worst case scenario. Only a few ports (around 17 ports) had error rates in the order of hundreds per hour; based on experience we assume that these *bad* ports can be quickly detected and fixed by system administrators.

3. APPLICATION-LEVEL RECOVERY

3.1 Application

3.1.1 Fast Fourier Transform

The fast Fourier transform (FFT), one of the most widely used computational kernels in the world, computes the discrete Fourier transform (DFT) in one or more dimensions. It is used in a diverse set of fields, from computing convolutions to solving partial differential equations. The FFT

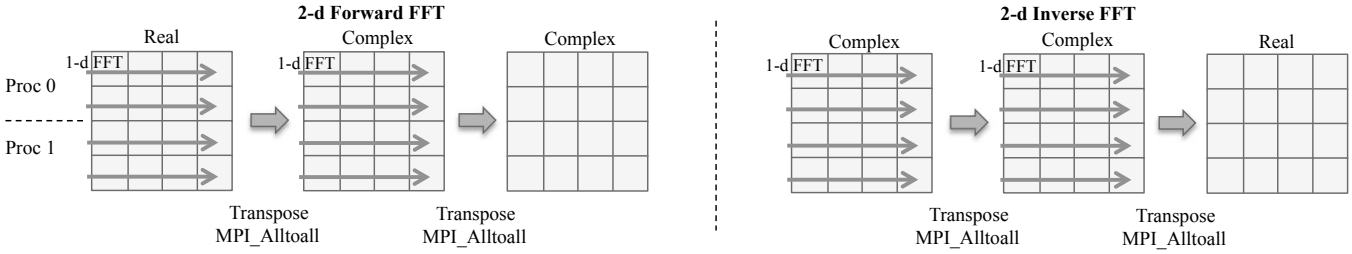


Figure 3: Overview of our 2D FFT implementation. We compute 1D FFTs in columns and rows. The last transpose (in both the forward and inverse FFT) brings back the input matrix to it original form.

converts time (or space) to frequency and vice versa. In its one-dimensional (1D) form, FFTs take an array of N real or complex numbers and produce an array of size N. Similarly, multi-dimensional FFTs take an N-dimensional array and produce an N-dimensional array. Multi-dimensional FFTs are very important in practice. For example, 3D FFTs are used in large-scale molecular dynamics codes [9], whereas 2D FFTs are used to manipulate large high-resolution radar images [10]. In our work, we use a 2D FFT image processing application to validate the effectiveness of network-fault recovery methods.

3.1.2 Our Parallel FFT

We implement a parallel 2D FFT based on the Cooley-Tukey algorithm [11]. The input is an array of $N_x \times N_y$ elements. Elements are stored in row-major order and distributed along the y dimension. Thus, each parallel process owns N_x/P y-pencils, where P is the number of processes. Figure 3 shows the procedure for a 4×4 FFT on two parallel processes (0 and 1) for both the forward and the inverse FFTs. Each process holds two 4-element y-pencils and performs 1D FFTs in each dimension using the `FFTW` library [12]. We perform transposes using all-to-all communication. Note that the forward FFT input is real, whereas its output is complex—although forward FFTs accept complex data as input, our input is an image, which contains real data. In contrast, the inverse FFT takes complex data as input and produces real data as output.

We implemented our FFT application in C++ with MPI. Complex numbers are packed (in an MPI buffer) as two `MPI_DOUBLE` numbers; one for the real part and another for the imaginary part. Since real elements only occupy one number, their imaginary parts are filled with zeros.

3.2 Recovery Methods

3.2.1 Assumptions

We assume that the MPI library divides the application messages into multiple packets—as most MPI libraries do—and then transmits them using UD transport to ensure scalability. We assume that the MPI library detects network errors at a packet granularity and, instead of retransmitting a corrupted or lost packet, the library notifies the application about the corrupted elements in an application message. These corrupted elements correspond to the corrupted packets. As previous work demonstrates [2], sliding window protocols can implement this detection functionality efficiently at the MPI library.

Note that our *application*-level recovery methods can be applied not only to end-user applications but also to different layers of the software stack, including numerical and communication libraries.

3.2.2 Methods

We describe our interpolation-based recovery methods (see Figure 4):

1. *Zero padding*: Fill corrupted elements with zeros. The overhead of this policy is $O(c)$, where c is the number of corrupted elements.

2. *Average*: Fill corrupted elements with the average of two adjacent non-corrupted elements. One non-corrupted element comes from the right and the other comes from the left of the corrupted element. The overhead of this policy is also $O(c)$.

3. *Regression*: First, we fit a curve $F(x) = y$, where x corresponds to elements and y corresponds to values. We use n non-corrupted elements from the right and n from the left of the corrupted element to fit the curve. Then, we fill corrupted elements by evaluating $F(x)$. We use a kernel-recursive least squares algorithm and the `dlib` machine learning toolkit [13] to train and test $F(x)$. The overhead of this policy is $O(c \times n)$.

3.2.3 Interpolation of Complex Numbers

Interpolation methods, such as averages and regression, do not work readily on complex data— most regression techniques are designed for real data and assume 1D input. To address this, we perform interpolation of real numbers independently from imaginary numbers. From a given buffer of complex numbers $\{(r_1+i_1), (r_2+i_2), \ldots, (r_j+i_j), \ldots\}$, where r_j is a real part and i_j is an imaginary part, we create two buffers; one buffer contains real numbers $R = \{r_1, r_2, ...\}$ and another buffer contains imaginary numbers $I = \{i_1, i_2, \ldots\}$. Then, we perform interpolation in R independently from I.

Another solution is to convert complex numbers to magnitudes and phases using, for example, the Euclidean distance, and then perform interpolation. We plan to investigate this approach in future work.

4. EVALUATION

4.1 Fault Injection

Our fault injector is part of the GREMLINs infrastructure [14], a toolkit that emulates the behavior of future machines by injecting perturbations in current machines. We developed a network-error gremlin that injects network errors from MPI call wrappers, which are implemented via

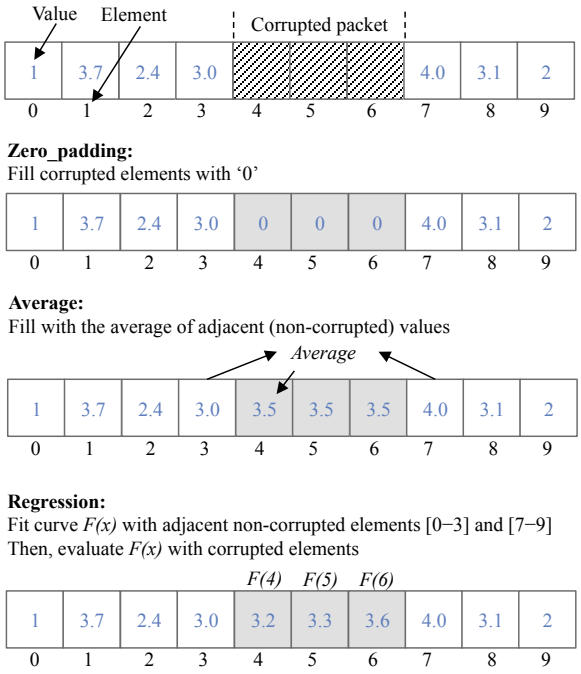


Figure 4: Recovery policies for a corrupted three-element packet on a message of ten elements.

the MPI profiling interface. In particular, we inject errors into `MPI_Alltoall` during the FFT global transposes. The injector determines the number of packets that comprise a message and selects those that would be affected by network errors based on our failure model (see Section 4.4).

The injector give us the ability to select when, where, and to what granularity to inject an erroneous packet. Specifically we can select: (1) the affected MPI *process*; (2) the MPI *routine* and *instance* (there are two instances of `MPI_Alltoall` in the forward FFT and two more instances in the inverse FFT); (3) the *packet size*; and (4) the *number of erroneous packets* in a run.

Note that by choosing the `MPI_Alltoall` instance, we can control injecting an error on packets that carry input composed of real or complex numbers. The all-to-all operations of the forward FFT and the first operation of the inverse FFT carry complex numbers, while the last all-to-all operation of the inverse FFT carry real numbers.

4.2 Randomizing Elements in a Message

A packet contains multiple message elements, so losing a packet results in losing a region of contiguous elements; interpolation methods may not be able to reconstruct a region of elements accurately if the variations within the region cannot be estimated from neighboring elements.

To illustrate this problem, consider loosing a packet of ten elements, $e_i, e_{i+1}, \ldots, e_{i+9}$. We can approximate the element values in the boundaries of the packet (i.e., e_i and e_{i+9}) by using the adjacent elements (i.e., e_{i-1} and e_{i+10} respectively). However, it is difficult is estimate values for the middle elements, such as e_{i+5}, since its adjacent elements, e_{i+4} and e_{i+6}, are also unknown.

Because, for a 2D FFT, input data are packed into an MPI buffer before the all-to-all operation, elements are rearranged locally in multiple blocks, one for each process. This helps alleviate losing contiguous application elements, but even with this rearrangement a block could span several packets containing multiple message elements.

To further reduce the possibility of losing contiguous data, we randomize the order of the elements in a message before the data are split into packets and sent over the network. When a packet is received, we use the same randomization function to appropriately re-order the received elements into the application's buffer. We implemented this mechanism in our GREMLIN emulation framework. Figure 5 illustrates this procedure.

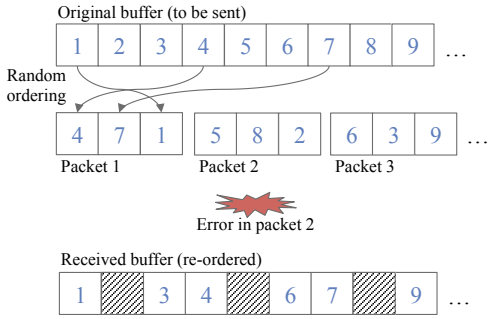


Figure 5: Randomizing the order of the elements in a message before network transmission. In this example, a packet comprises three elements. When a packet error occurs, we avoid losing contiguous data.

4.3 Evaluation Metric

We use the normalized root-mean-square (RMS) function to calculate the error of our recovery policies:

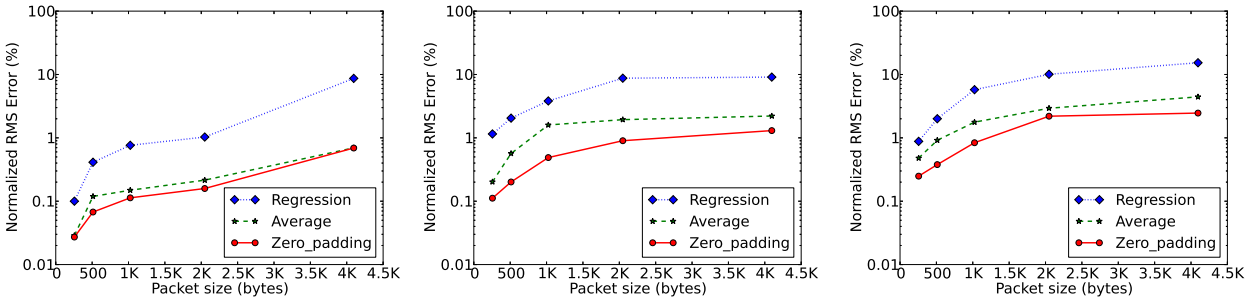
$$Error = \frac{\sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}}}{y_{max} - y_{min}}, \quad (4)$$

where y is an element of the FFT input, $\hat{y}_i$ is an element the FFT output, and N is the number of elements. The denominator corresponds to the range of values of the input data. Formula (4) can be expressed as a percentage (multiply by 100), where lower values indicate less residual variance based on the magnitude of the input. In our FFT implementation, the output data are equal to the input data, thus a zero error percentage means that our recovery methods are perfect.

4.4 Experiments and Results

In this section, we evaluate the error of each recovery policy as described in the previous section. We run our FFT application with 1,000 MPI processes on *Sierra* using a $4,267 \times 4,267$ pixels image as input. The image was taken from a NASA Mars exploration rover [15].

In our experiments, an MPI process is selected randomly and packet errors are injected in one of the four `MPI_Alltoall` operations. We perform injections for error rates of 1, 5, and 10 packet errors per run, which emulates the highest error rates observed in our log analysis of *Sierra*. We vary the packet size in each experiment to investigate different MPI implementations that could make use of different packet sizes to transmit messages under UD. We use packets of 256, 512, 1,024, 2,048, and 4,096 bytes of size. Each experiment is run 10 times and we calculate the average error.



(a) 1 packet error/run (b) 5 packet errors/run (c) 10 packet errors/run

Figure 6: Normalized RMS error for the evaluated recovery policies. The injected packet error rate increases across the plots (from left to right). `Zero_padding` achieves the minimum amount of error compared to `Average` and `Regression`.

4.4.1 Normalized RMS Errors

Figure 6 shows the normalized RMS error for each of our recovery policies. Figure 6a shows the results for an error rate of one (packet) error per run. We observe that `zero padding` and `average` incur less error than `regression`—about one order of magnitude difference. Both `zero padding` and `average` keep the error under 1%, even for large a packet size of 4,096 bytes. On the other hand, the error for `regression` could reach up to 10% for large packet sizes.

Figure 6b and 6c show the results for increased error rates of five and ten packet errors per run respectively. We observe that the overall magnitude of the normalized RMS error increases for all the methods. We also observe that, contrary to our expectations, `zero padding` incurs the smallest error; the maximum normalized RMS error for `zero padding` is 2.4% for ten packet errors per run and the maximum packet size of 4,096 bytes.

From these experiments we learned the following: (1) the normalized RMS error is proportional to the packet size and the error rate, which is what we expected because the higher the packet size and error rate, the larger the amount of data that is lost; (2) the normalized RMS error is low, as low as 0.1% for MPI packets of small size, in the order of 256–512 bytes; and (3) filling lost data with zero values works surprisingly well to recover from errors in FFTs. We explain (3) in the next section where we analyze the effect of packet errors on different data inputs (i.e., real and complex).

4.4.2 Errors on Real and Complex Input

We separate packet errors that occur on real inputs from those on complex inputs. Recall that when a packet error occurs in any of the first three transposes of the FFT, it involves complex data, whereas when a packet error occurs in the last transpose, it involves real data.

Figure 7 shows the normalized RMS error for the three recovery techniques from real and complex inputs. The magnitudes of the error from real input are overall lower than the ones from complex input, for all the methods. The reason for this is that real data elements are numerically correlated to their neighbor data elements—at least for the case of an image. Thus, interpolation techniques take advantage of this fact to reconstruct lost data. In contrast, complex numbers, which have two components (real and imaginary) are poorly correlated; elements between corrupted and non-corrupted data elements are not numerically similar.

In Figure 7a, we observe that contrary to the results of Section 4.4.1, `zero padding` incurs higher normalized RMS error than the other methods. While the other methods can take advantage of neighbor non-corrupted data to interpolate real numbers, the `zero padding` method cannot. `Average` is the approach the one with the smallest error. We attribute this result to the fact that `regression` fits its model ($F(x)$) with *noise* data, i.e., data that are unnecessary to make accurate predictions.

Figure 7b shows similar results to those in Section 4.4.1. `Zero padding` has the smallest amount of error. Since there is a low correlation between a complex data element and its adjacent element, interpolation methods, `average` and `regression`, tend to produce noisy (new) elements; it is better to assume that a component of a complex number, either real or imaginary, is simply zero. Since there are more chances of getting a packet error on complex than real numbers, packet errors with complex data dominate the overall normalized RMS error (as show in Section 4.4.1).

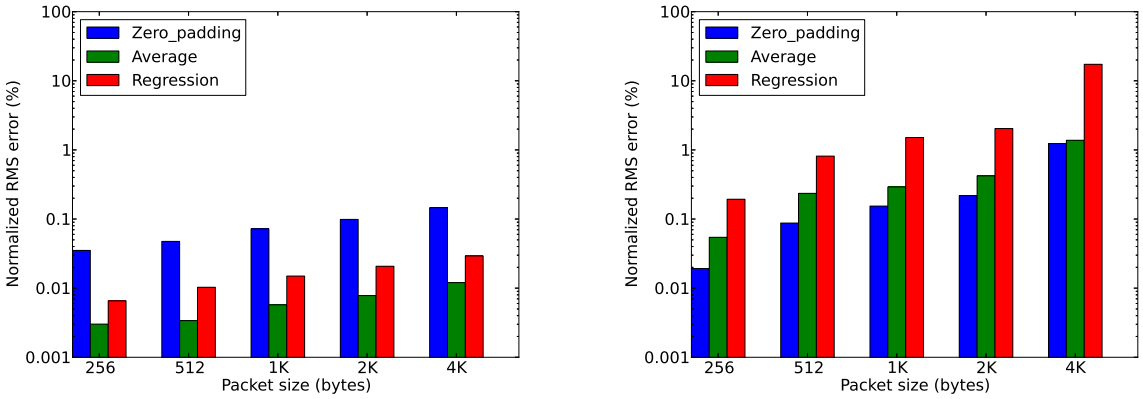
5. DISCUSSION

In this section, we discuss the limitations and practical implications of this work.

First, our recovery techniques assume that the input data show some *numerical locality* (i.e., adjacent data is correlated). We leverage this property to reconstruct corrupted or lost data due to network failures. In our FFT computations that process an image with real input, the normalized RMS error is small under realistic network error rates.

Second, initial measurements indicate that one can implement our recovery policies with low overhead for the `zero padding` and `average` methods, which operate only on one to three elements for each erroneous element. In some cases, we can even eliminate this overhead completely by simply initializing receive buffers appropriately (zeros for the `zero padding` method). We acknowledge that methods such as `regression` may be more costly. Different applications may show different tradeoffs in terms of which policies are more accurate and which can be implemented more efficiently. We anticipate that our recovery methods are at least more scalable than current RC-based MPI implementations since the memory overhead of keeping per-process connections is fully eliminated with our UD-based MPI implementation.

Third, there is more than one metric to measure the deviation of the recovered data versus the uncorrupted (reliable)



(a) Real.

(b) Complex.

Figure 7: Normalized RMS error for real and complex inputs.

data. We used the normalized RMS error (as opposed to the RMS error) because it takes into account the numerical magnitude of the input. A different metric may produce different results in terms of the magnitude, but we expect the trends of the curves to be similar. Ultimately, the choice of an error metric depends on the application and its input.

Finally, recovery methods other than the ones proposed in this work are possible. Our choice of reconstruction policies represent a balanced approach in terms of accuracy and overhead of implementation. For instance, sequential-data machine-learning techniques can be used to improve the `regression` method, but they are computationally expensive, which would hinder the scalability of our approach.

6. RELATED WORK

Algorithm-based fault tolerance (ABFT) methods have been proposed to detect and correct errors in matrix operations (e.g., addition, multiplication, scalar product, and transposition) in parallel systems [16, 17, 18]. The main idea is to compute extra information, such as checksums, that can be used for error detection and error correction. Moreover, their focus is on detection and correction from memory- and CPU-related hardware errors, rather than from network errors. Our proposed recovery policies do not require storing extra information, instead, we allow the application to reconstruct corrupted data from non-corrupted data at the cost of accuracy.

A large amount of work has been done in fault-tolerance for FFTs. Wang and Jha [19] consider an algorithm-based approach to tolerate errors in networks of FFTs, which has lower overhead than previous approaches; it can be implemented in hardware with minimal overhead. Oh et al. [20] reported an algorithm-based approach to detect errors FFT that achieves a high error coverage with low false positive rate by applying linear weight factors to checksums. Fu and Yang [21] proposed an algorithm to survive process failures in a parallel implementation of FFTs. These approaches are complimentary to our proposed methods since they do not focus on network faults and do not take into account MPI scalability (as we do).

From a scalability point of view, the research studies that are closest to our work provide scalable UD-based implementations of MPI [2, 3]. Our methods, in fact, rely on these implementations to achieve scalability. Our work eliminates the need of packet retransmissions by reconstructing missing or corrupted data at the application level.

Finally, a body of work analyzed network error logs from data centers [22], IP backbone networks [23], and wide-area networks [24, 25], but none analyzed errors from InfiniBand networks, which are used frequently in HPC systems. Recently, Koop et al. [26] proposed an MPI design for InfiniBand that can cope with network and HCA errors. Unlike this work, our focus is to investigate the implications of handling network errors at the application level.

7. CONCLUSIONS AND FUTURE WORK

We propose and evaluate application-level recovery methods for transient network faults. Our methods can leverage the scalability advantages of UD-based MPI implementations but without the need of packet retransmissions. Our methods allow applications that are generally resilient to minor errors during computation to recover from packet errors via data interpolation. Our results show that, in an FFT code, normalized RMS errors can be as low as 0.1% for application messages that are transmitted with small packet sizes (i.e., 256–512 bytes). We also show that our recovery methods are more accurate on real inputs than on complex inputs. Lastly, we demonstrate that a simple recovery policy, such as filling lost data with zero values, works surprisingly better than interpolation methods such as regression-based methods. Ultimately, the accuracy of our methods are application and input dependent. The results from our FFT evaluation are promising and encourage us to investigate other algorithms that may be resilient to transient errors.

Our future work includes the following. First, we plan to evaluate the accuracy of our proposed techniques on various applications and with different inputs. We intend to classify these applications by their resiliency to network faults. We are also considering implementing our recovery methods in numerical or communication libraries so that application developers can use them without changes to their codes. Second, we plan to evaluate network error distributions of multiple HPC clusters to study factors that contribute to transient network faults. These factors may include vendor

provider, component specific (cables, switches, and HCAs), node placement, and network topology.

Acknowledgments

The authors would like to thank the Livermore Computing division at LLNL for their support and feedback. In particular, we thank Trent D'Hooge, Jim Silva, Adam Moody, Matthew Leininger, and Albert Chu.

8. REFERENCES

[1] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *International Journal of High Performance Computing Applications*, vol. 23, pp. 374–388, Nov. 2009.

[2] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda, "High performance MPI design using unreliable datagram for ultra-scale Infiniband clusters," in *International Conference on Supercomputing*, ICS'07, (Seattle, WA), ACM, June 2007.

[3] A. Friedley, T. Hoefler, M. L. Leininger, and A. Lumsdaine, "Scalable high performance message passing over Infiniband for Open MPI," in *KiCC Workshop*, (Aachen, Germany), Dec. 2007.

[4] M. Koop, J. Sridhar, and D. Panda, "Scalable MPI design over InfiniBand using eXtended reliable connection," in *IEEE International Conference on Cluster Computing*, (Tsukuba, Japan), pp. 203–212, 2008.

[5] M. J. Koop, S. Sur, and D. K. Panda, "Zero-copy protocol for MPI using InfiniBand unreliable datagram," in *IEEE International Conference on Cluster Computing*, (Austin, TX), Sept. 2007.

[6] V. Karamcheti and A. A. Chien, "Software overhead in messaging layers: Where does the time go?," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, (San Jose, CA), ACM, Oct. 1994.

[7] J. Tang and A. Bilas, "Tolerating network failures in system area networks," in *International Conference on Parallel Processing*, ICPP'02, (Vancouver, B.C., Canada), IEEE, Aug. 2002.

[8] A. Avizienis, J.-C. Laprie, B. Randell, *et al.*, *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[9] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, Mar. 1995.

[10] G. A. Mastin, S. J. Plimpton, and D. C. Ghiglia, "A massively parallel digital processor for spotlight synthetic aperture radar," *International Journal of High Performance Computing Applications*, vol. 7, no. 2, pp. 97–112, 1993.

[11] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[12] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. Special issue on Program Generation, Optimization, and Platform Adaptation.

[13] D. E. King, "Dlib-ml: A machine learning toolkit," *Journal of Machine Learning Research*, vol. 10, pp. 1755–1758, 2009.

[14] Lawrence Livermore National Laboratory, "GREMLIN infrastructure." `https://scalability.llnl.gov/gremlins/`.

[15] NASA, "Mars exploration rovers." `http://marsrovers.jpl.nasa.gov/gallery/images.html`.

[16] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[17] Z. Chen and J. Dongarra, "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources," in *International Parallel and Distributed Processing Symposium*, IPDPS'06, (Rhodes Island, Greece), 2006.

[18] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pp. 225–234, 2012.

[19] S.-J. Wang and N. Jha, "Algorithm-based fault tolerance for FFT networks," *IEEE Transactions on Computers*, vol. 43, no. 7, pp. 849–854, 1994.

[20] C. Oh, H. Y. Youn, and V. Raj, "An efficient algorithm-based concurrent error detection for FFT networks," *IEEE Transactions on Computers*, vol. 44, no. 9, pp. 1157–1162, 1995.

[21] H. Fu and X. Yang, "Fault tolerant parallel FFT using parallel failure recovery," in *International Conference on Computational Science and Its Applications*, ICCSA'09, pp. 257–261, 2009.

[22] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM'11, pp. 350–361, 2011.

[23] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot, "Characterization of failures in an operational IP backbone network," *IEEE/ACM Transactions on Networking*, vol. 16, pp. 749–762, Aug. 2008.

[24] C. Labovitz, A. Ahuja, and F. Jahanian, "Experimental study of internet stability and backbone failures," in *International Symposium on Fault-Tolerant Computing*, FTCS '99, (Washington, DC), IEEE, 1999.

[25] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage, "California fault lines: understanding the causes and impact of network failures," in *Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM'10, (New Delhi, India), Aug. 2010.

[26] M. Koop, P. Shamis, I. Rabinovitz, and D. Panda, "Designing high-performance and resilient message passing on InfiniBand," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, IPDPSW'10, 2010.