Building Reliability into MPI Applications for Distributed Systems

James Bickerstaff jjb169@pitt.edu

Jefferson Boothe jdb193@pitt.edu

Abstract

Large-scale supercomputing clusters are key for performing highly computation or data intensive applications within a reasonable amount of time. With such a large quantity of components, system reliability becomes very important as the likelihood of faults is greatly increased. One such method of fault mitigation is adding error correction codes to critical program data to detect and even correct some faults. In this study, we utilize betweenness centrality as a baseline parallel application to present a custom-built fault injector and demonstrate the fault tolerance of our MPI-based communications with the addition of Hamming Codes. We find our implementation successfully masks all faults with medium to low fault rates, with minimal computational overhead. We also see that at the highest fault rates, SDC occurs and the resulting calculations are erroneous.

Keywords

Betweenness Centrality (BC), Hamming Code, Error Correction Code (ECC), Message Passing Interface (MPI), Silent Data Corruption (SDC), Single Error Correction & Double Error Detection (SEC/DED)

I. Introduction

Distributed computing has become an invaluable tool in overcoming large-scale computing problems within a reasonable amount of time. Composed of potentially thousands of nodes with millions of physical cores, it becomes imperative to ensure the overall reliability of the system in order to avoid program interruption and any potential downtime. Information-related failures stand out within these systems as key points of failure, as it has been observed that memory is a leading cause of faults within large-scale computing clusters [1, 2]. Additionally, the process of communicating large amounts of data between nodes for distribution of work is an area that requires fault detection, correction, and tolerance. If communicated data contains errors, the system must have these mechanisms mentioned in place, otherwise the corrupted data may propagate through computation and cause larger failures later on.

The focus of this study is on building information reliability into the Betweenness Centrality (BC) application used within the area of graph processing. This application was chosen due to its high data-intensity, requiring repeated memory fetches with very little spatial or temporal locality. BC also takes a substantial amount of time to complete execution on a traditional personal computer, opening the door for speedup on parallel computing clusters. With this parallel computing comes the need for communication of large chunks of data between clusters, which is of primary interest to these studies.

Detection and correction of such data errors is done through the use of a Single Error Correction and Double Error Detection (SEC/DEC) Hamming code. This choice is made due to the frequency at which single-bit errors occur, allowing machines to simply correct the received data rather than requesting it to be sent again, which costs valuable runtime. Double-bit errors, though less frequent, need to be detected as well. Otherwise, Silent Data Corruption (SDC) may occur, leading to false results without raising an alarm to the users. To observe the effects and capabilities of the SEC/DED code created, a fault injector has also been designed for this research. This injector determines bit-flips based on varying probabilities, measuring the number of errors corrected and detected, as well as those that go unnoticed.

II. Related Work

To first illustrate the need for building reliability into a distributed system, studies of supercomputing clusters and their root causes of failure will be discussed. The work done within resource [1] showcases the distribution of failures experienced within Los Alamos National Laboratory (LANL) clusters between the years 1996 and 2005. As displayed in Figure 1, hardware accounts for a majority of the root causes of failure within the LANL systems, accounting for 64% of the experienced failures. Expanding upon this, it was found that 30.1% of hardware failures were caused by the memory DIMMs [1]. Despite this, most of these memory related failures were found to be correctable given the proper fault tolerant systems, as error correcting codes were often outpaced by the amount of bit-flips occurring [1]. Similarly, a study of the Sunway BlueLight system discovered that 91% of system faults were caused by DRAM single-bit and multi-bit errors (SBEs and MBEs) [2]. These case studies display the need for improved fault detection and correction techniques within systems, as a majority of memory faults can be found and fixed during runtime. Those that cannot be corrected should at least be flagged accordingly, so that a system may avoid complete failure and loss of progress.

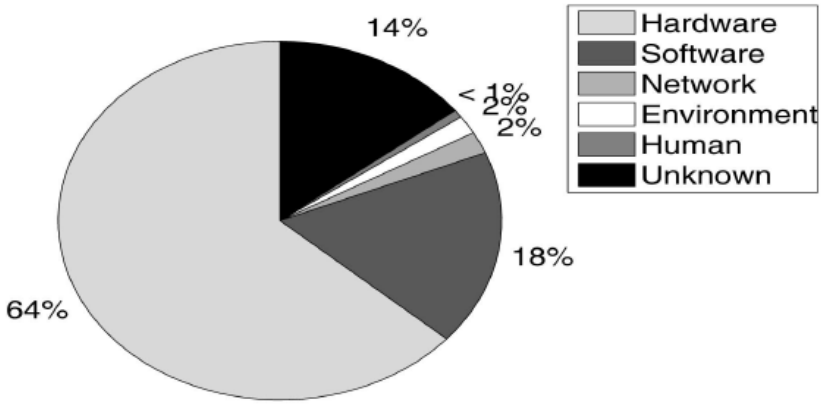


Figure 1. Breakdown of failures by root causes experienced by LANL systems [1]

Due to the importance and popularity of MPI within supercomputing clusters, there have been many studies observing the reliability of apps that make use of this communication standard. The experiments conducted by Lu and Reed within [3] seek to examine the impact of transient errors within MPI applications. This was done in two ways: injecting faults into the system memory (registers and application memory regions), and injecting faults into the MPI messages transmitted. The memory faults were limited to injection within the address space of MPI processes- including the text, stack, and heap [3]. Once the injector was triggered, it would obtain the memory space in use from malloc calls, and randomly flip a bit within this space. When injecting faults within MPI messages, the researchers looked at the three layers of how the interface is implemented: the API, the ADI (Abstract Device Interface), and the Channel. This stack is illustrated within Figure 2 below. The Channel was chosen as it is the interface between MPI and the underlying network-specific communication software [3]. This allows for direct modification of the incoming transmissions before it is interpreted by the ADI and API. A message's payload is overwritten once the number of received messages surpasses the value of a randomly generated threshold derived from the communication patterns of previous iterations [3]. It was concluded within their studies that registers and messages are most vulnerable to errors produced by single bit-flips. Faults would manifest and produce errors an average of 34.7% of the time, and this resulted in an incorrect output being produced 28 to 71% of the time [3]. It should be noted that the error data gathered was obtained from three different applications, utilizing MPI to different extents.

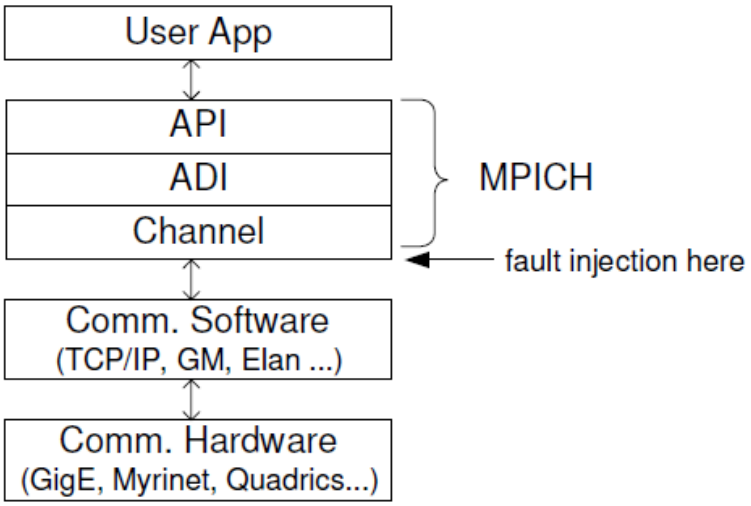


Figure 2. MPI Message Stack and Fault Injection Location [3]

A study conducted by Bautista-Gomez and Capello [4] also takes aim at transient errors within supercomputing systems, although they propose multiple strategies to alleviate their impact as well. The application in use for this study is a turbulent flow simulation based on Navier-Stokes equations. The primary contribution from this research are the four methods of silent data corruption (SDC) detection that are developed and used in conjunction with one another to maximize coverage. Once detected, the method of resolving these corruptions, however, was left largely to the application so that their detectors could be application-agnostic. The first detector is titled the “Plain Detector,” which observes the given dataset input and generates a distribution of the expected data within. If any value is observed to be outside this expected range, it will be flagged as a fault and measures will be taken to correct it [4]. The second detector is titled the “Spatial Detector,” and monitors the spatial distribution of the dataset. The expected space variations of a dataset are computed, and once again, if an observed value is outside this range, the system will take corrective action [4]. The third detector is the “Temporal Detector,” and this tracks the temporal evolution of a given dataset. If any value within the dataset changes to outside of the predicted distribution, then it will be flagged as a data corruption [4]. The final detector combines the spatial and temporal calculations, creating the “Spatiotemporal Detector.” This detector looks to observe when a dataset increases or decreases its level of activity and by approximately what range [4]. These detectors are all combined through a fuzzy logic module, allowing over 85% of corruptions to be detected, while incurring less than 1% computational overhead [4].

A third study conducted by researchers at the California Institute of Technology [5] sought to create an accurate transient fault simulator that emulates radiation-induced bit-flips and faults. It is designed to operate on any number of MPI threads, while providing control over the rate at which faults occur, and how many bits they impact [5].

This study also focuses on injecting faults directly into registers and memory, aiming to uniformly and randomly distribute the faults into portions of the system that are under more use at the time of injection [5]. Results from the testing phase displayed great success for uniformly distributing the amount of injected faults to both registers and memory addresses. This study is important to observe as they depict what may be seen as a “guideline” for how to construct a balanced fault injection system, which provides the opportunity to design and incorporate advanced fault detection and correction mechanisms for experimentation as well.

III. Background

Within this section, the concept of Parity and Hamming Codes in general will be discussed. Additionally, the BC operation will be explained in detail, depicting how the algorithm works, its parallelization, as well as its importance within the field of graph processing. Brief overview of the MPI implementation used is also provided.

A. Parity Codes

The key method used for generating check bits for error detection and correction within this research is parity. Parity codes are represented as a series of extra bits appended to the data that are generated by performing an XOR operation on the data itself [6]. Figure 3 showcases an example of how an even parity bit may be generated within hardware. When there is an even number of 1’s within the data, a parity bit of 0 will be produced. If there is an odd number of 1’s within the data, a parity bit of 1 will be produced. This simple concept is expanded upon to produce a code capable of SEC and DED, making use of multiple parity bits to cover different portions of a data string. Multiple parity bits are utilized to create the necessary codes for detecting which bits are in error, and is elaborated on in part B below.

Another key concept utilized to realize SEC/DED capabilities is *overlapping parity*. Overlapping parity ensures that each bit of data is “covered” by more than one parity bit [6]. By checking each data bit with multiple parity bits, it is possible for the Error Correction Code (ECC) to not only find the exact data bits that may be in error when particular bit-patterns are produced, but the parity bits may check for errors amongst themselves as well.

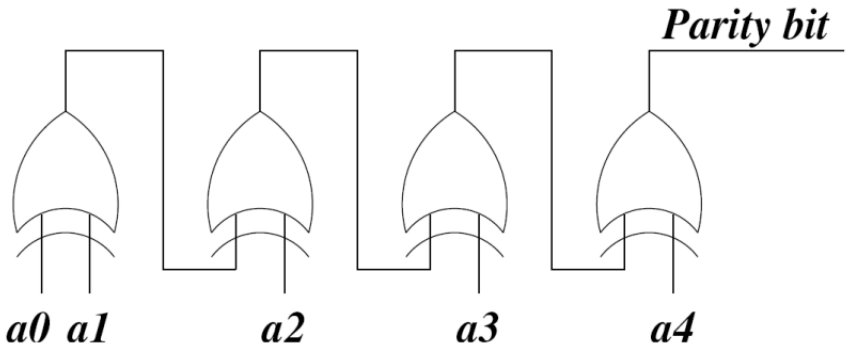


Figure 3. Even parity bit generator [6]

B. Hamming Codes

Hamming codes are built upon the concept of *Hamming distance*, which may be defined as the number of bit positions in which two codewords differ [6]. In order for a code to correct up to *k* bits, the distance between valid codewords must be 2*k*+1, which is key for implementing SEC within a system, as codewords need to be carefully chosen with a distance of 3. To expand this code for DED capabilities, an additional bit must be added to the codewords, trading complexity for this added capability. By using these codewords, when one or two bit-flips occur in any location of the data or check bits themselves, they may be detected and fixed if possible. This detection and correction is made possible through the use of *syndromes*, which are unique bit-patterns created by performing an XOR operation on the received check bits against the recalculated ones [6]. The number of syndromes possible is determined by the size of the code and data words. A code is considered "perfect" if the number of code bits (r), plus the number of data bits (d), plus one is equal to 2^r (2^r=d+r+1). This is because there are d+r possible bits that may be in error, plus 1 for the error-free state [6]. Thus, the total number of possible states for the codeword is equal to the number of possible combinations of the code bits. If this is the case, there is a 1:1 relationship for relating syndromes to error bits. If a code is not perfect, however, it must meet the general case to be valid ($2^r \geq d+r+1$). In this situation, there are more syndromes than possible bits, and the designer must select which ones to use for their code. In the event of a single error, a syndrome will point out exactly which bit received is in error, allowing for a system to correct it by simply flipping its value. A double error is denoted in different ways, depending on the implementation of a code. If a perfect code is utilized, then a double error will be detected whenever the extra parity bit added for DED is equal to 0, while any other check bit is equal to 1. In the event of a non-perfect code, a double error will be detected if a syndrome is produced that does not match any of those chosen for single error correction. Regardless of how DED works, in the event a double-bit error is observed, the system must request a retransmission of the data that is in error and once again check for validity upon reception.

C. Betweenness Centrality (BC)

For this research, the data of interest consists of graphs and arrays. Graphs are a non-linear data type consisting of vertices and edges, commonly used due to their application in many real-world scenarios. For example, a graph can be used to represent systems such as an energy grid, social networks, or road map. Having a digital, numeric representation of such systems allows for different information to be gathered and/or calculated to gain insights into the system overall. There are numerous ways to represent graphs in computer science, and in this study the adjacency list method was utilized. This requires storing a separate list of edges from each node. Compared to alternatives, such as the adjacency matrix, this can save considerable memory footprint, especially in large, sparse graphs which are of frequent interest in HPC analyses.

One particular graph measurement of interest in network theory is Betweenness Centrality (BC). Formalized in 1977 by Freeman, BC provides a metric to compare how central a node in a graph is based on its *betweenness* [7]. Here, betweenness refers to the number of shortest paths between pairs of other points that this node is a member of [7]. This provides key information about the network's behavior. An example given in [7] is that a person strategically placed as the shortest communication path between other pairs in the group would have great responsibility, such that if they were withholding or tampering with information passed along, they would have significant impact on the network.

D. Message Passing Interface (MPI)

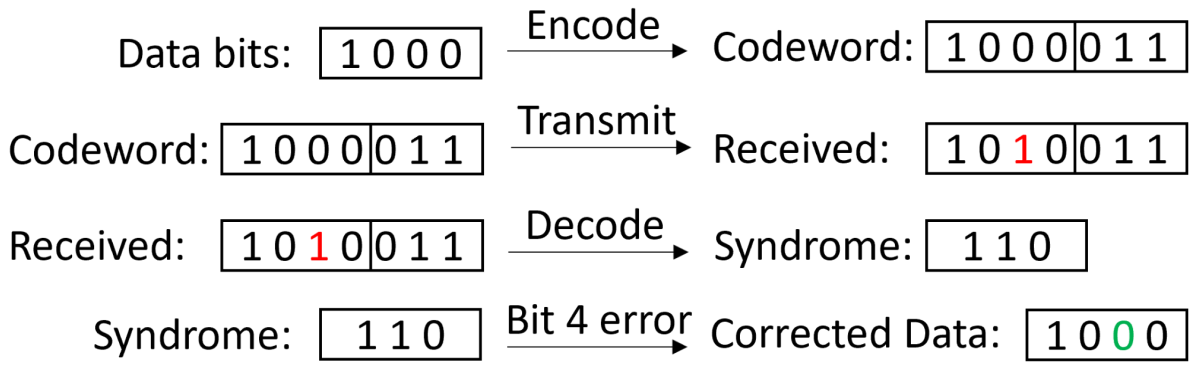
The message passing interface (MPI) describes a standardized messaging interface between distributed memory systems running a parallel program. There are many libraries that implement the MPI standard on various platforms. In this study, Open MPI is used as the communication library to share information amongst processors and finally calculate BC accordingly. Open MPI is an open source implementation of the MPI standard with support for a wide range of interconnects and configurations. Latest Open MPI releases do not offer any support for end-to-end data reliability other than that which is built into the underlying network (i.e. TCP). This makes the BC Open MPI application a strong candidate for fault injection and fault tolerance studies, as the communication is potentially vulnerable to data corruption.

IV. Approach

A. Error Correction Code (ECC)

For this research, 8 check bits are used to cover 64 bits of data, creating a (72,64) Hamming code. This is not a perfect Hamming code, however, as 2^8 (256) is greater than the number of data bits combined with the check bits plus one (73). Thus, a subset of total syndromes had to be chosen to represent bit errors. A parity check matrix containing 8 rows with 64 columns has been created by [8], and the associated syndromes designed for this matrix are utilized as well. Each parity bit accounts for 26 data bits within the string, achieving overlapping parity wherein each data bit is covered by at least 3 parity bits. Despite the SEC and DED capabilities of the code in use for this research, however, data is still susceptible to SDC in the event that 3 or more bit flips occur and produce a syndrome which incorrectly flags a valid bit as being in error. This is elaborated upon more within the Results section.

The process of encoding and decoding data is as follows. To encode, the system provides the 64-bit data string that will be transmitted to the error correction and detection code. Each parity bit is calculated from this string, performing XOR on all data bits that it covers, and these 8 check bits are then appended to the data. Following this, the data is sent to its recipient. Upon receiving the data, the system will once again perform XOR on all data bits covered by each parity bit, this time including the parity bit itself in the operation. This will produce a resulting syndrome for the transmitted data. If the syndrome is all zeroes, then no error is detected. Otherwise, the system will find which bit-error matches the syndrome produced and correct it. In the event the syndrome does not match any of the 72 in use for single bit correction, a double error detection flag is raised to alert the system. Upon detecting a double-bit error, the data is retransmitted and checked once more for validity. An illustration of this process on a simple (7,4) Hamming code is shown in Figure 4. Parity bit 0 covers data bits 0, 1, and 2. Parity bit 1 covers data bits 1, 2, and 3. Parity bit 2 covers data bits 0, 2, and 3. Once a codeword is created, the bits are numbered from least significant (0) to most significant (6), where bits 0-2 are parity bits for the data. In this example, bit 4 (data bit 1) is flipped from 0 to 1 during transmission, causing a nonzero syndrome to be produced upon reception and recalculation of the parity bits. Referring to the syndrome table, it may be seen that this bit-pattern flags bit 4 as the erroneous bit, allowing the system to correct it.



State	Erroneous parity check(s)	Syndrome
No errors	None	000
Bit 0 (p_0) error	p_0	001
Bit 1 (p_1) error	p_1	010
Bit 2 (p_2) error	p_2	100
Bit 3 (a_0) error	p_0, p_1	011
Bit 4 (a_1) error	p_0, p_2	101
Bit 5 (a_2) error	p_1, p_2	110
Bit 6 (a_3) error	p_0, p_1, p_2	111

Figure 4. (7,4) SEC Example, Syndrome Table from [6]

B. Parallel Betweenness Centrality (BC)

In this study, we focused on the algorithm described in [9] by Brandes for the central application. Calculating BC requires obtaining the shortest path between every vertex pair and comparing the number of such shortest paths each vertex belongs to. Brandes' algorithm uses clever recursive properties of the shortest path accumulation to significantly reduce runtime complexity on large graphs compared to previous methods [9]. The algorithm spans the graph from every start node, calculating shortest paths to all other vertices and utilizing recursive properties of the paths to reduce the necessary computations [9]. Pseudocode depicting this process may be seen in Figure 5. The parallelisation of this algorithm used in this paper was achieved by having the start nodes distributed amongst all processing elements. The resulting shortest path data is then shared amongst the processors using Open MPI, allowing betweenness to be calculated. Lastly, the main node normalizes the results between 0 and 1, as is standard [9]. BC was selected as any fault in the information flow of the program that led to an error could impact the BC calculations for every single node, rendering all obtained results erroneous. Due to the interwoven nature of shortest path data in determining BC, it is imperative that no data errors occur at any point to ensure accurate results.

Algorithm 1: Betweenness centrality in unweighted graphs

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V;   σ[s] ← 1;
    d[t] ← −1, t ∈ V;   d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + (σ[v]/σ[w]) · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

Figure 5. Brandes' Algorithm Pseudocode [9]

C. Fault Tolerant Wrapper and Fault Injector Design

For this study, a custom fault injector was created for the parallel BC application. The fault injector was designed as a wrapper of the base application, intercepting all communication calls and determining when and where to inject faults before allowing the transmission to complete. Whenever an MPI call occurs, the wrapper utilizes a uniquely seeded random number generator to calculate when faults occur based on a known bit error rate, provided in configuration settings. In the base application, the most common communication is the sharing of a large array of floating point numbers. Upon

intercepting a call, the wrapper first encodes the entire array using the previously discussed error correction code, resulting in 1 additional byte of check bits per 8 bytes of data in the original array. Next, the entire message content is looped through one bit at a time, each time comparing a randomly generated floating point number between 0 and 1 with the bit error rate. If the generated number is less than the bit error rate, the bit is flipped. This updated message containing any faults is then transmitted using the true Open MPI functions. Receiving begins with the actual Open MPI function, and then the wrapper handles decoding before returning to the main application. Upon retrieval, the message is parsed in 72 bit chunks, the size of a codeword, using the previously described error correction code to correct single-bit errors and detect double-bit errors. Whenever a good chunk is parsed, its contents are saved and the chunk number is tracked. This is because when a double-bit error is detected, the entire chunk is deemed corrupt and must be retransmitted. In such a case, the receiver asks the sender for a retransmission, then the chunks already received correctly are skipped while parsing. This process is repeated until all chunks have been received without known errors at least once.

From the base application's point of view, a communication call is made on the floating point array and the encoding, fault injection, decoding, and retransmissions are all abstracted away and invisible. The fault injector can be configured to have different values for the bit error chance, allowing for a sweep of test environments to be analyzed without any modifications to the program. Based on the bit error chance, f, and each bit error being independently determined, one can obtain that for a message with $(d + r)$ bits, the probability of no bit flips occurring is $(1-f)^{d+r}$. Since the coding scheme is capable of correcting single bit errors, it is sometimes more interesting to determine the probability of more than one bit error occurring in a field of $(d + r)$ bits. This probability, Φ, can be obtained by the equation $\Phi(d,r) = 1 - (1-f)^{d+r} - (d+r)f(1-f)^{d+r-1}$. This shows the direct impact of bit error rate on the probability a chunk is corrupted and must be retransmitted.

V. Experimental Results

A. Testbed and Dataset

For this research, the University of Pittsburgh Center for Research Computing (CRC) was leveraged, making use of their available MPI clusters [10]. These nodes are equipped with dual socket Intel Xeon Gold 6342 CPUs, and connected via HDR200 InfiniBand interconnects. The primary graph processed for this research is the "General Relativity and Quantum Cosmology collaboration network (GR-QC)" obtained from the Stanford Network Analysis Platform (SNAP) datasets [11]. This is an undirected, real-world graph, and consists of 5,242 vertices with 14,496 edges connecting them together (28,992 once doubled for analysis). The data represented by this graph is

relationships between authors who submitted to the “General Relativity and Quantum Cosmology” category of arXiv. Other graph datasets of similar scale were tested with this application, although the results remained much the same. Thus, the following charts contain data measured from only the GR-QC dataset.

B. Results

Testing of the application consisted of a sweep of multiple bit error rates, each a factor of 10 larger than the previous, as well as utilizing a range of processing elements in powers of 2 starting with 2 and scaling up to 32 nodes. Additionally, multiple trials were performed for each configuration to ensure consistency in results, averaging the results to produce values seen within the charts.

The most desirable results from the error correction code are accurately detecting no-bit errors and performing single-error corrections. In both of these situations, a system continues execution without the need for retransmission of data, in the worst case performing a rapid XOR check and correcting only a single bit as needed. Occurrences of no-bit errors were not tracked throughout this study, however. Figure 6 displays the number of single-error corrections performed across the different values of bit error chance *f* tested. The chart is of logarithmic scale, and when plotted like this, showcases a steady increase in SECs needed across executions as the error rate per bit increases. It should also be noted that for measuring SECs performed on 2 nodes with 1e-6 error chance, the resulting value is 0.4, which is why it dips below the other bars. As expected, the number of single-bit errors encountered also increased with the number of nodes in use, as there are more communications required to scatter and gather resulting data. An interesting take away from these results are how quickly the number of bit-flips increases with each magnitude of *f*. As this study was done through software simulation, there were no issues correcting them. Within hardware designs such as those used for memory DIMMs, however, high speed circuits capable of servicing large numbers of requests rapidly, while also ensuring the validity of data are much more difficult to create. This introduces a situation in which the tradeoff of ensuring data correctness versus performance will need to be taken into account.

Double-error detection, while not as desirable as SEC, provides the system a way to continue execution even in the event of multiple bit-errors. Whereas, if only SEC were possible, silent data corruptions (SDC) or larger-scale system failure may occur if multiple bits are in error. The primary reason for DED being less desirable than SEC is that it requires retransmission of the resulting data between nodes. This costs valuable runtime in which additional work may have been completed. Figure 7 displays the average number of DEDs encountered during execution of the BC application. It’s very promising to see that no double-bit errors were encountered until *f* was equal to 1e-4, as this showcases the importance of reliable circuitry that provides intrinsic fault tolerance to data errors. In the event of less reliable hardware, or more noisy environments,

however, the number of double bit errors increases rapidly from *f* equaling 1e-4 to 1e-2. Thus, additional circuitry and protection of data is still necessary to minimize the number of uncorrectable errors encountered. This is especially important within mission-critical environments such as space or medical devices, as a system failure or SDC may drastically impact decision making of the system.

Figure 8 shows the required retransmissions due to double errors detected in the decoding process for various node quantities and bit error rates. Immediately, it is clear that the lowest bit error rates that resulted in no double errors did not need any retransmissions. As the bit rate increases and the quantity of double errors grows, more retransmissions are needed to obtain errorless data. It can also be seen that increasing the quantity of nodes directly increases the retransmission amounts. This can be explained as the number of communication calls scales with the number of nodes. With more communication calls, there is more opportunity for double errors to occur and likewise retransmission. Figure 9 reframes the retransmission data as a percentage increase in total communication calls over an error-free execution. Again, only the highest bit error rates result in an increase in communication calls, but the variations by node count are flattened. Thus, one can conclude that the bit error rate has a direct correlation with the percentage of communication calls containing uncorrectable double-bit errors.

The least desirable, and most dangerous possible outcome from this error correction code is silent data corruptions (SDCs). As stated within the Approach section above, this occurs when three or more bit-errors occur, causing the system to generate a valid syndrome that flags a correct bit as being in error. Because of this, the system thinks it has corrected a single-bit error, continuing execution without any interruption. Once again, this is particularly dangerous within situations where an output is used to determine the next plan of action for a system such as a space shuttle. To track if SDC occurred within this study, the result of BC was compared to a “golden” output once complete, and if the data did not match, then it was concluded that the data had been corrupted. The results of tracking SDCs are presented in Table 1, displaying interesting results. There is no corruption occurring within the results until 32 nodes are used with a bit-error chance of 1e-3. This is more than likely due to the much larger amount of communication needed between 32 nodes. Once the bit-error chance reaches 1e-2, however, the results of execution become corrupted within every node configuration. Thus, if the system has hardware that keeps *f* at a value of 1e-4 or less, then SDCs are much less likely to occur. It should also be noted that when an SDC occurred, as stated previously, the system would treat it as a single-bit error and correct it, adding a tally to the SEC counts displayed within Figure 6. Due to their very infrequent occurrence, however, they are heavily outweighed by the number of actual SECs performed. An additional mitigation that may be made to avoid SDCs such as these occurring, in this application and others like it, is to create a layer of software fault tolerance on top of the

ECC. This additional layer may be in the form of a wrapper that verifies inputs and outputs against an expected range as set to and customized for the application in use. If something like this were in place for this application, it is possible that many SDCs may be detected and flagged for retransmission, provided a significant bit value is changed.

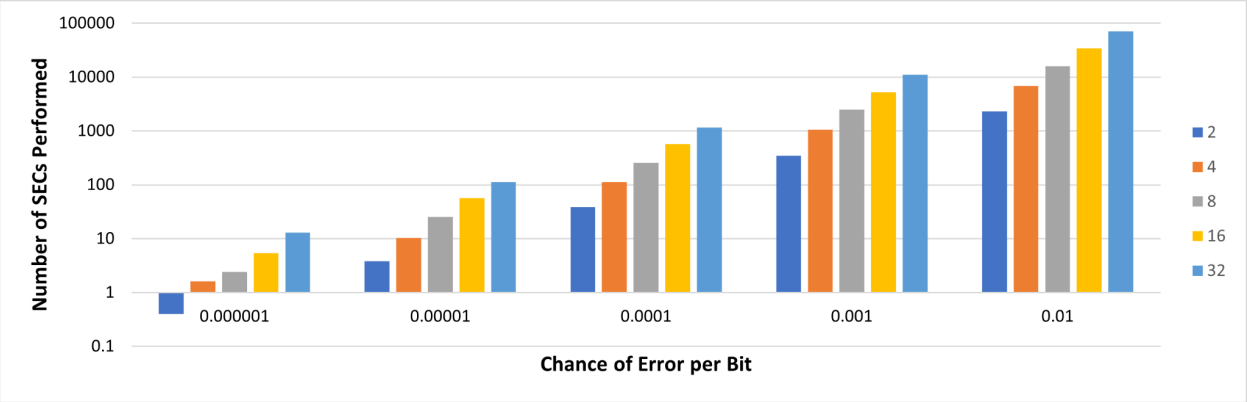


Figure 6. Number of SECs performed across multiple values of *f*

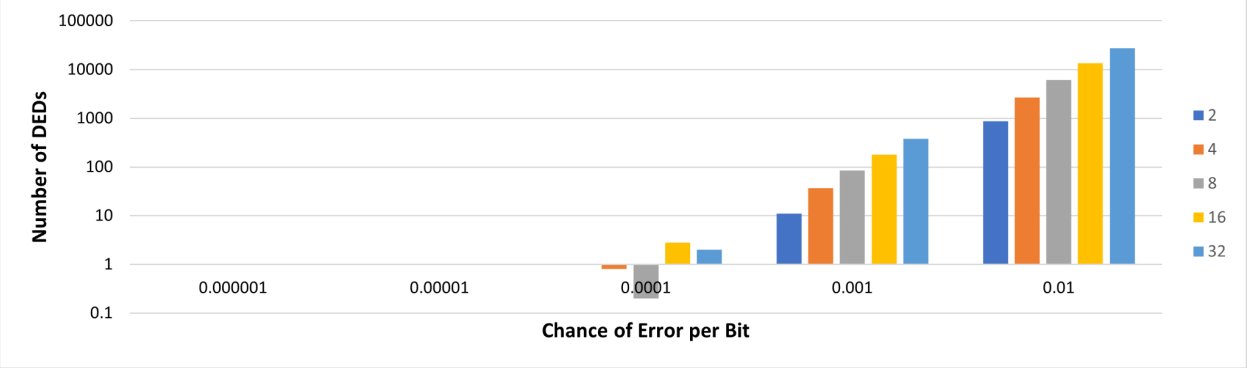


Figure 7. Number of DEDs performed across multiple values of *f*

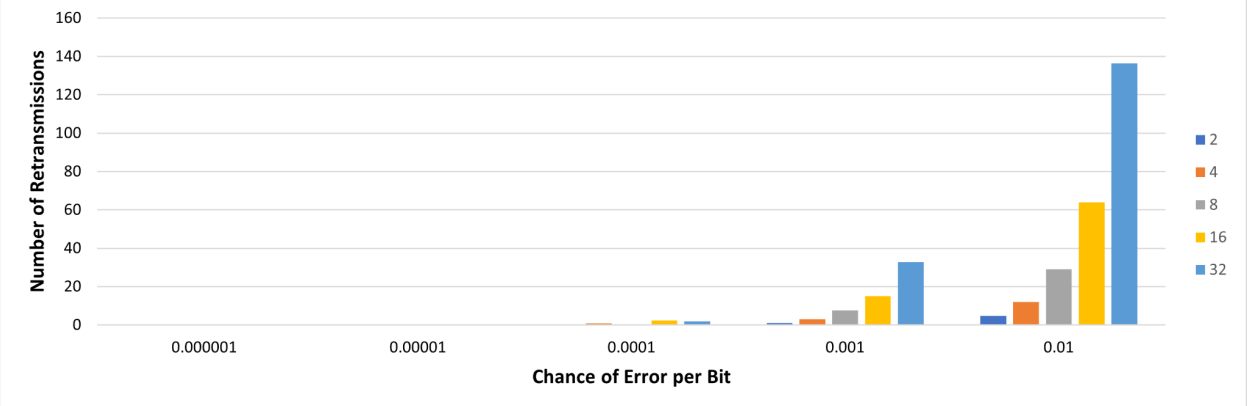


Figure 8. Number of retransmissions performed across multiple values of *f*

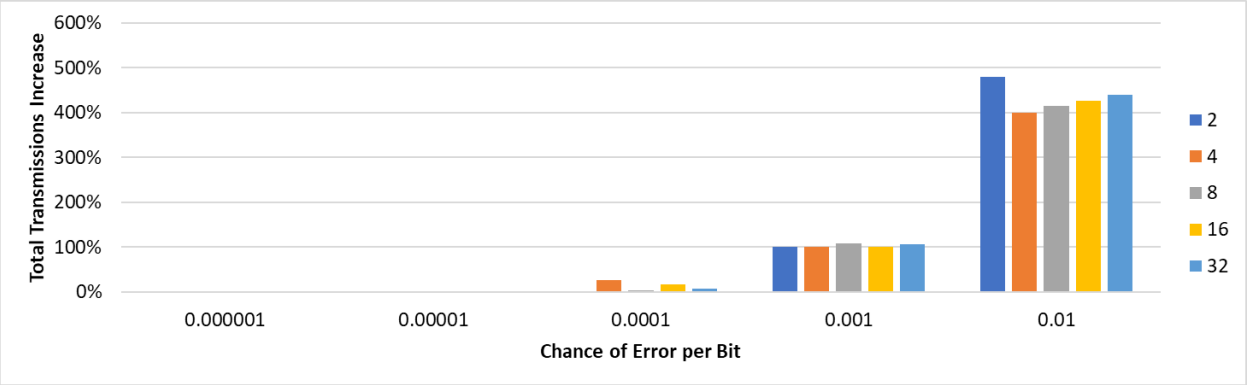


Figure 9. Percent increase in total transmissions performed across multiple values of *f*

Table I: SDC During Execution

Nodes	*f*	Result Corrupted?
2	1e-6	No
4	1e-6	No
8	1e-6	No
16	1e-6	No
32	1e-6	No
2	1e-5	No
4	1e-5	No
8	1e-5	No
16	1e-5	No
32	1e-5	No
2	1e-4	No
4	1e-4	No
8	1e-4	No
16	1e-4	No
32	1e-4	No
2	1e-3	No
4	1e-3	No
8	1e-3	No
16	1e-3	No
32	1e-3	*Yes*
2	1e-2	*Yes*
4	1e-2	*Yes*
8	1e-2	*Yes*
16	1e-2	*Yes*
32	1e-2	*Yes*

VI. Conclusions

In this paper we integrate information fault tolerance within MPI calls to provide ways to correct and detect errors that may occur within transmissions. Betweenness Centrality was targeted due to its large computational runtime as graphs scale in size, and due to the many communications needed to scatter information and reduce values to produce results. Thus, ECC would be invaluable so that the large runtime is not wasted due to corrupt results. It has been found that using a (72,64) Hamming code with SEC and DED capabilities provides tolerance for errors within the system that would otherwise lead to corruption of data if it is not in place. A significant amount of single-bit errors were corrected, and double-bit errors detected as the bit-error rate increased in magnitude. Silent data corruptions were still present at the highest fault rates, however, and would require more investigation or expansion of Hamming Code capabilities to account for. Future work for this research includes testing larger-scaled graphs, with millions of vertices and edges, and observing the number of bit-errors corrected and detected, as well as the presence of SDCs. Increasing the processing element count is also an avenue of interest, as it has been observed that the increase in the node count increases the amount of transmissions and quantity of data shared, proportionally increasing the number of faults and potential to corrupt the final results.

All code for this research may be found at: https://github.com/jjb169/ECE2165_Project

VII. References

[1] B. Schroeder and G. A. Gibson, “A large-scale study of failures in high-performance computing systems,” IEEE Transactions on Dependable and Secure Computing, vol. 7, no. 4, pp. 337–350, 2010.

[2] R.-T. Liu and Z.-N. Chen, “A large-scale study of failures on Petascale Supercomputers,” Journal of Computer Science and Technology, vol. 33, no. 1, pp. 24–41, 2018.

[3] Charng-da Lu and D. A. Reed, "Assessing Fault Sensitivity in MPI Applications," SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, 2004, pp. 37-37, doi: 10.1109/SC.2004.12.

[4] Bautista-Gomez, Leonardo, and Franck Cappello. "Detecting silent data corruption for extreme-scale MPI applications." Proceedings of the 22nd European MPI Users' Group Meeting. 2015.

[5] R. R. Some, W. S. Kim, G. Khanoyan, L. Callum, A. Agrawal and J. J. Beahan, "A software-implemented fault injection methodology for design and validation of system

fault tolerance," 2001 International Conference on Dependable Systems and Networks, 2001, pp. 501-506, doi: 10.1109/DSN.2001.941435.

[6] I. Koren and C. M. Krishna, Fault-Tolerant Systems, 2nd ed. Cambridge, MA: Morgan Kaufmann, 2021.

[7] Freeman, Linton, "A set of measures of centrality based on betweenness". Sociometry. 40 (1): 35–41. doi:10.2307/3033543. JSTOR 3033543, 1977.

[8] J. Gaeddert, “Liquid-DSP: Digital Signal Processing Library for software-defined radios,” GitHub. [Online]. Available: https://github.com/jgaeddert/liquid-dsp.

[9] Brandes U. A faster algorithm for betweenness centrality. Journal of mathematical sociology. 2001 Jun 1;25(2):163-77.

[10] “Node configurations,” *University of Pittsburgh Center for Research Computing*. [Online]. Available: https://crc.pitt.edu/resources/cluster-hardware-overview/node-configurations.

[11] J. Leskovec and A. Krevl, “Stanford Large Network Dataset Collection,” *SNAP Datasets: Stanford Large Network Dataset Collection*, Jun-2014. [Online]. Available: https://snap.stanford.edu/data/.