Journal Title Here, 2022, 1–6
doi: DOI HERE
Advance Access Publication Date: Day Month Year
Paper

FrameD: Framework for DNA-based Data Storage Design, Verification, and Validation

Kevin D. Volkel,[1,*] Kevin N. Lin,[2] Paul W. Hook,[3] Winston Timp,[4] Albert J. Keung[5] and James M. Tuck[6,*]

[1,6]Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina, USA, [2,5]Department of Chemical and Biomolecular Engineering, North Carolina State University, Raleigh, North Carolina, USA and [3,4]Department of Biomedical Engineering, Johns Hopkins University, Baltimore, Maryland, USA

FOR PUBLISHER ONLY Received on Date Month Year; revised on Date Month Year; accepted on Date Month Year

Abstract

Motivation: DNA-based data storage is a quickly growing field that hopes to harness the massive theoretical information density of DNA molecules to produce a competitive next-generation storage medium suitable for archival data. In recent years, many DNA-based storage system designs have been proposed. Given that no common infrastructure exists for simulating these storage systems, comparing many different designs along with many different error models is increasingly difficult. To address this challenge we introduce FrameD, a simulation infrastructure for DNA storage systems that leverages the underlying modularity of DNA storage system designs to provide a framework to express different designs while being able to reuse common components.

Results: We demonstrate the utility of FrameD and the need for a common simulation platform using a case study. Our case study compares designs that utilize strand copies differently, some that align strand copies using Multiple Sequence Alignment (MSA) algorithms and others that do not. We found that the choice to include MSA in the pipeline is dependent on the error rate and the type of errors being injected and is not always beneficial. In addition to supporting a wide range of designs, FrameD provides the user with transparent parallelism to deal with a large number of reads from sequencing and the need for many fault injection iterations. We believe that FrameD fills a void in the tools publicly available to the DNA storage community by providing a modular and extensible framework with support for massive parallelism. As a result, it will help accelerate the design process of future DNA-based storage systems.

Availability and implementation: The source code for FrameD along with the data generated during the demonstration of FrameD is available in a public Github repository at `https://github.com/dna-storage/framed` (10.5281/zenodo.7757762)

Contact: kvolkel@ncsu.edu or jtuck@ncsu.edu

Key words: DNA data storage, fault injection, storage systems, error correction, simulation

Introduction

The world is generating data faster and in larger quantities than ever before, raising concerns that traditional storage technologies will not scale to keep up with demand. In the search for new technologies, DNA has gained broad interest due to its superior density and longevity compared to magnetic tape and hard disk drives. Since the early work of Church et al. (2012) and Goldman et al. (2013) demonstrating the ability to store information in DNA strands using modern DNA technology, there have been a range of studies answering important questions such as data addressability (Tomek et al., 2019; Bornholt et al., 2016; Tomek et al., 2021; Lin et al., 2020; Organick et al., 2018, 2020), synthesis efficiency (Antkowiak et al., 2020; Choi et al., 2019; Anavy et al., 2019), DNA reusability (Tomek et al., 2019; Lin et al., 2020), error rates associated with a variety of techniques (Organick et al., 2018; Tomek et al., 2019; Grass et al., 2015; Matange et al., 2021), and the density that can be achieved in DNA molecules (Goldman et al., 2013; Erlich and Zielinski, 2017; Choi et al., 2019; Anavy et al., 2019; Antkowiak et al., 2020). This vast knowledge base of DNA-based data storage comes with an equally

© The Author 2022. Published by Oxford University Press. All rights reserved. For permissions, please e-mail: journals.permissions@oup.com

expansive space of possible implementation approaches for which little if any consensus has been reached. Compounding the problem of choosing any one approach is the fact that there is a lack of common infrastructure that enables the comparison of these designs in a fair and reproducible manner.

To address the growing need for tools to analyze and compare DNA storage systems, we present FrameD, a software framework for designing, verifying, and validating DNA storage system designs. FrameD is not a library of every conceivable error correction algorithm, instead, it provides a fault-injection-based test bed in which DNA storage systems can be evaluated. Constructing FrameD requires several considerations. One being FrameD's flexibility in terms of what DNA storage systems can be represented. To address this issue, we use current literature to inform the construction of a model that can be used as a basis in which a range of DNA storage systems can be expressed. With this model, we are able to implement an execution back-end that executes a set of encoding steps that adhere to the model's interfaces. Thus, for an encoding to be used in FrameD, a user need only follow the interface specification. This execution model back-end also provides transparent support for necessary bookkeeping steps like DNA strand indexing and dropout inference, allowing the user to focus on the details of their algorithms.

Another issue that needs to be considered when simulating DNA storage systems is computational scale issues that arise from several sources. One source is the size of the possible parameter space of interest with regards to an encoding/decoding algorithm, as exploring combinations of parameters can easily lead to exponential growth in the number of experiments. Another source of computational scale arises from the necessity to perform fault injection experiments 1000's of times to achieve narrow confidence intervals on key outcomes such as strand and file decode rates. Compounding each source of computational overhead is the scale of sequencing data that needs to be processed. To support scalability, FrameD utilizes batch jobs to parallelize individual configuration simulations and MPI to parallelize units of work within those batch jobs like strand decoding and fault injection iterations. FrameD implements the parallelization support transparently such that users do not need to manage parallelization communication. Instead, the user just specifies their configurations and the computational resources to allocate to each.

We demonstrate the utility of FrameD by performing a comparison between three designs across two error models representing different sequencing technology. We evaluate 240 total configurations, generating a total of 654 million fault-injected DNA strands, and analyze the read and write density trade-off between the three designs. Our results show that the optimal design approach depends on the designer's read and write cost targets and the target sequencing technology, and bolsters our claim that a common simulation infrastructure is needed.

The Case for DNA Storage Simulation Infrastructure

Before discussing details of FrameD, we present a study of current literature to further motivate the need for a DNA simulation infrastructure and to understand the basic components that such an infrastructure will need to support. For our review, we choose 13 previous works that implement end-to-end DNA storage systems. We selected these works because they are representative of different approaches that have been taken since the revival of DNA data storage started by Goldman *et al.'s* work. Thus, we should be able to make conclusions about consistent approaches taken in DNA storage design, while also being able to account for the inclusion of novel techniques from each individual work. Detailed organization of these works is presented in Supplementary Table 1.

We find that transformations applied to information can be organized hierarchically in two levels. At the first level we identify two broad categories we refer to as **Single strand** and **Multi-Strand** transformations. **Single Strand** transformations focus on processing information stored in a single molecule of DNA, while the **Multi Strand** processes relate to processing information stored within a group of molecules.

Under the **Single Strand** category, we found 3 typical transformation steps: **Binary Transformation**, **Transcoding**, and **Functional Site Encoding**. Each of these sub-categories modify the data present on a single DNA molecule in their own way. A **Binary Transformation** modifies the raw digital information before it is represented as DNA molecule. Such modifications typically included parity checks (Bornholt et al., 2016), Reed-Solomon codes (Grass et al., 2015; Antkowiak et al., 2020), and base conversions from the typical Base-2 binary representation of digital information to a numerical base that may be more convenient for a certain Reed-Solomon field (Grass et al., 2015; Anavy et al., 2019). The **Transcoding** category consists of processes that represent the digital source information in terms of a DNA molecule. While transcoding can be as simple as a base-conversion to base-4 (Antkowiak et al., 2020), approaches typically consider constraints such as GC balance (Press et al., 2020; Yazdi et al., 2017) and homopolymers (Bornholt et al., 2016; Tomek et al., 2019; Goldman et al., 2013; Organick et al., 2018), yielding a range of options with different error correction and density properties. The final **Single Strand** pass, **Functional Site Encoding**, is not inherently dependent on the raw information stored but instead includes DNA substrings in the stored molecules to facilitate functionality. Functionality encoding includes adding primers for polymerase chain reaction (PCR) random access (Tomek et al., 2019; Bornholt et al., 2016; Organick et al., 2018), T7 promotor sites for RNA transcription (Lin et al., 2020), and restriction sites for DNA fragmentation (Tomek et al., 2021).

Under the **Multi-strand** category, we determined 3 distinct processing steps: **Outer Code**, **Consolidation**, and **Reconstruction**. The **Outer Code** step is similar to the **Binary Transformation** step of the single strand category, except error correction codes like Reed-Solomon are applied using the data of a group of strands so that errors can be corrected using information dispersed across DNA molecules (Press et al., 2020; Tomek et al., 2021; Organick et al., 2018). This error correction technique is crucial for dealing with the occurrence of missing DNA molecules, a common error mode of DNA storage systems (Press et al., 2020; Organick et al., 2018; Bornholt et al., 2016). Another issue that a DNA storage system design must address is the reconstruction of the order of information. Representing arbitrarily large sets of information requires storing subsets of information on individual DNA molecules because synthetic DNA of arbitrary length is not feasible to synthesize. Provided mixtures of DNA molecules are not guaranteed to be sequenced in any particular order, a **Reconstruction** strategy is needed to map a DNA molecule to its place in the complete data set. Because of its optimality regarding density (Heckel et al., 2017), an **indexing** strategy that stores an ordering integer in each strand is a common approach (see Supplementary Table 1). Lastly, because

DNA storage systems typically generate multiple copies of each transcoded DNA molecule by way of synthesis, sequencing, or amplification (Organick et al., 2018; Bornholt et al., 2016; Yazdi et al., 2017; Tomek et al., 2019), a processing step which we call **Consolidation** is required to generate 1 final representative of the information of a stored DNA molecule. This can be as simple as detecting and removing bad strands using error correction until finding a valid strand (Bornholt et al., 2016; Goldman et al., 2013; Tomek et al., 2019; Press et al., 2020), or bioinformatics tools such as multiple sequence alignment (MSA) algorithms can be employed to determine a consensus sequence (Yazdi et al., 2017; Antkowiak et al., 2020; Organick et al., 2018).

From this discussion, we can immediately see that the transformation of information within a DNA storage system can be described by a small set of general categories. This indicates that an infrastructure which provides support for routing this information between these categories can be effective in representing many unique DNA storage systems.

Given this observation, we consider how prior works in DNA storage simulation support general encoding and simulation environments. DNA-Storalator is a DNA storage system simulator that focuses on evaluating clustering and reconstruction algorithms that work to construct a representative DNA strand from a cluster of noisy versions. However, DNA-Storalator does not offer support for the evaluation of Outer, Binary Transformation, or Transcoding codecs Chaykin et al. (2022). Furthermore, the only error models supported are those which are generated by wet-lab experiments Sabary et al. (2021). While these may be accurate for a given DNA storage system implementation, allowing for user-defined error models allows for testing codecs over a wider set of cases. DNAssim offers flexible fault and coverage models, along with outer code evaluation. However, DNAssim does not consider the complex space of inner code design and chooses to only use binary to base-4 conversion as its transcoding method Marelli et al. (2023). Another simulator is DeSP, however it is only a tool that provides error injections and strand distribution changes, and so it is not a framework for evaluating different encoding designs Yuan et al. (2022). In the following sections we discuss how FrameD provides a framework in which a rich space of encodings can be evaluated using flexible error models in a scalable environment.

FrameD

We leverage categorical overlap of designs by understanding that there is a logical ordering in which the information transformations can be applied. This ordering is illustrated in Figure 1, where information flows from left to right. First, information in a file is broken into contiguous pieces called **packets**. **Packets** serve as the scope for the outer encoding, allowing for designers to choose a granularity for the outer error correction algorithm. Before the outer code is applied, the **packet** is broken down into **base-sequences**, contiguous sections of information that will be stored on each DNA molecule. A packet's base-sequences are then processed by the outer code, which generates indices automatically for each base-sequence while also adding additional error correction base-sequences. FrameD defaults to basic incremental integers for indexing, but provides the designer the interface to implement special indexes like the Luby Seed for Fountain codes. Each indexed base-sequence is passed through the **inner-encoder**, a series of steps consisting of the **Single Strand** operations.

The decode phase is mostly identical to encoding, except that the transformations made by each pass are reversed. However, decoding must have a mechanism to deal with multiple copies per encoded strand. We found two approaches that can be taken. One is to first cluster the DNA strands input to the decoder based on similarity scores like edit distance or using a MinHash-based approaches (Antkowiak et al., 2020; Organick et al., 2018; Rashtchian et al., 2017). Then MSA algorithms, such as Muscle, can be used to aggregate information across strands and help resolve errors through consensus voting (Antkowiak et al., 2020; Edgar, 2004; Yazdi et al., 2017). To account for these approaches, we provide users the ability to add MSA and clustering steps to the pipeline before the inner encoding is reversed. This process is outlined in the Supplementary Information. Another approach is to consolidate the strands after completing the inner code, throwing out strands that may violate error checks, and coming to a consensus on the digital representation of information. FrameD supports the use of either approach, or even both.

The pipelined approach of FrameD provides DNA storage system designers several benefits. By implementing the routing of information between components, the designer can focus on their algorithms as long as it adheres to the pipeline's information transformation interface. Further, by breaking larger steps, e.g. inner code, down into smaller sub-components, a

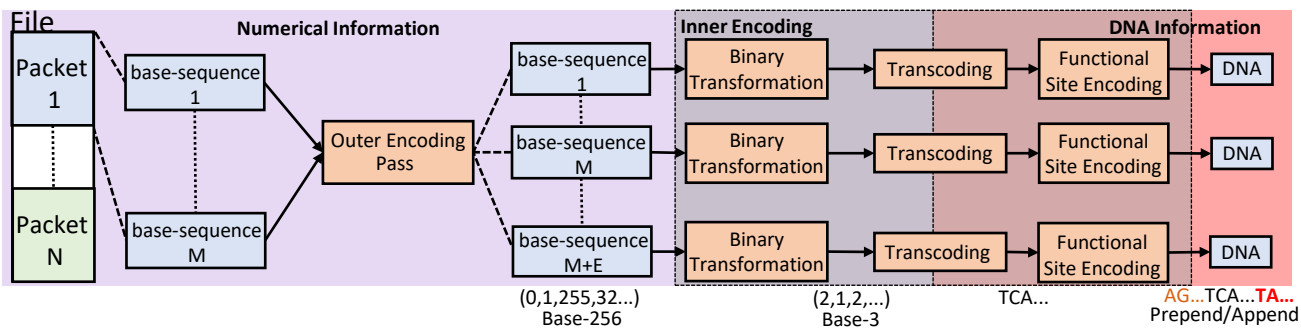

Fig. 1: Model used by FrameD to represent encoding for DNA storage systems. Included in this figure is an example of the state of information throughout the pipeline, where initially data is in its original byte representation, then converted to base-3 and subsequently DNA, and finally stored with prepend/appended DNA strings typically done for PCR primer functionality.

new unique inner code can be constructed by modifying just a single component without needing to re-implement the other algorithms that constitute the inner code. For example, one could change their Binary Transformation pass while keeping the same implementations of the Transcoding and Functional Site Encoding passes. As long as the new component adheres to its prescribed interface given the type of transformation it performs, no other work is required outside of implementing the new algorithm.

Figure 1 shows only 3 components within the inner encoding and 1 outer code component, however, we point out that both can consist of an arbitrary number. That is, Outer Codes, Binary Transformations, and Functional Site Encoding can be cascaded an arbitrary amount. For example, one may have two binary transformations, one to modify the numerical base representation of data and another one to convert to a Reed-Solomon code. We leave out Transcoding from this list, since transcoding binary information to DNA can only occur once. Details on how indexing is supported in cascaded Outer Codes is discussed in the Supplementary Information. FrameD also leverages transformation type interfaces to provide validation checks for the designer to ensure that there is a logical arrangement of components.

Fault Injection Workflow

FrameD includes a simulation tool based on a fault injection methodology for evaluating DNA storage system designs. Fault injection simulation provides several benefits over the analytical model. Fault injection experiments provide the ability to quickly estimate properties of specific steps of the encoding/decoding processes. For example, fault injection experiments can determine a rate at which strands can be successfully decoded using a specified error model. This information can be used to verify analytical results or to derive parameters for a full-scale DNA storage system. Fault injection models are also more flexible, where changing an encoding or error model can make deriving a new analytical model difficult. Lastly, a fault injection framework exercises actual algorithm implementations against strands with errors in them, allowing for benchmarking and debugging that an analytical model cannot provide.

Figure 2 overviews the workflow of using FrameD for fault injection. First, a user develops a JSON configuration file which includes a path to the binary data to be converted to DNA, along with details of three general categories of parameters that control simulation behaviour. **1)** Encoder/decoder parameters that configure the behavior of encoding/decoding components. **2)** Fault distribution parameters that configure the simulated channel's error model. **3)** Copy distribution parameters that configure the model used to represent strand copies that arise in the storage system. Within this configuration file, users can specify a parameter sweep to explore combinations of parameters such as error rate and inner code rates, and the fault injection tool will generate individual simulation jobs (batch jobs) for each unique parameter combination.

During the evaluation of a simulation job, the input binary data is passed through the encoder to generate a library of synthetic DNA strands that subsequently sample the copy and error model distributions. The noisy set of strands passes through decoding, during which an attempt to reconstruct the original file is made, and information on errors and their locations are captured.

The output of fault injection consists of a set of files placed in a unique directory for each unique parameter combination. This set of files specifies the parameters that were used for the experiment. This allows us to keep records of all parameters used for all experiments easily. In addition to these files, a statistics file is output by the simulation, aggregating counters that are used to track events during simulation such as decode failures, location of byte errors within strands, location of base errors within strands, etc. FrameD allows statistics to be largely user-defined so that appropriate statistics can be chosen for a given experiment. [1] In the following section, we outline how this is done in FrameD. Our documentation provides a tutorial on fault injection using FrameD.

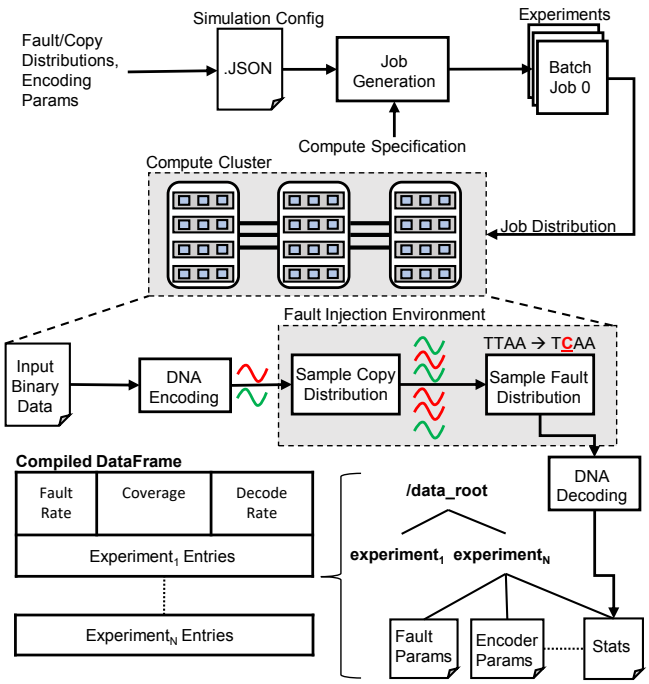

Fig. 2: End to end workflow of performing fault injection simulation using FrameD.

Generating Statistics

The fault injection tool of FrameD captures general system-level information out of the box such as file decode rate and total byte-errors in the final file. Designers can also further leverage FrameD to generate custom statistics of a strand's information at various points of the pipeline. The mechanism to achieve this is called a **probe**. A probe is a special pipeline component that does not modify information, but can interrogate the state of strands as they are encoded/decoded. To calculate error rates, a probe can capture the state of the information as it is encoded. This snapshot of the information can then be used as a comparison point during decoding to calculate statistics. Such statistics can be versatile and cover a wide range of error types including errors within the bases of a strand and the errors within bytes after decoding strands. A detailed example is provided in the Supplementary Information.

[1] We do not list every statistic collected, since they are easy to change.

Analyzing NGS Data

While this paper focuses on the *in-silico* fault simulation tool of FrameD, we recognize that users of this framework would likely want to leverage the probes they created for fault injection simulation to analyze sequencing data from real experiments. In addition to the fault injection tool, we provide a tool that evaluates FrameD pipelines against NGS data. This additional tool allows developers the opportunity to determine which encoded DNA strand that a sequencing read originated from. This is something that is not known *a priori* and has to be computed since determining errors and their positions requires baseline information to compare against. Computing this mapping can be as simple as pairing a decoded index with the sequencing read identifier, or by computing the best alignments to a known set of strands Sabary et al. (2021). Our codebase includes a mapping probe to perform this analysis along with detailed documentation on how to leverage it within FrameD for NGS data analysis.

Handling Computational Scale

When performing fault injection simulations, and decoding strands from real sequencing data, computational scalability quickly becomes an issue. FrameD enables scalability by identifying parallel units of work within the general flow of information in the framework. This allows any design to leverage paralellization transparently since FrameD can handle all communication of data. FrameD identifies parallelizaiton at three levels shown in Figure 3.

In the first level, FrameD creates and submits individual batch jobs that can be processed by HPC workload managers like Slurm or LSF. Within each batch job, FrameD can be be configured to allocate compute resources in the form of MPI ranks to both fault injection iterations (second level) and work done during fault injection simulations like decoding individual strands, packet outer codes, and sequence alignment (third level). Allocation is up to the user. A user may allocate more MPI ranks to fault injection iterations if individual decode tasks are small, or the user may allocate most ranks to decoding to deal with sequencing data or to benchmark their pipeline. We utilize MPI at these levels due to its ease of communication and scalability. By targeting these clear large-grain units of work we can reach large numbers of computational cores while also keeping those resources busy.

We recognize that there may be other user-defined opportunities for parallelization. For example, consolidating DNA strands using a clustering approach may benefit from MPI-based parallelization (Rashtchian et al., 2017). Because communication patterns are specific to such algorithms, we provide the user with the MPI communicator that is allocated to the given fault injection iteration. This allows the user to implement their own custom parallelization if so desired. See Supplemental Figure 4 for FrameD's communication patterns.

While FrameD aids in scaling the number of strands and fault injection experiments performed, we point out that the rate in which information will be decoded/encoded will be greatly influenced by the chosen algorithms and their implementation details. A major implementation detail impacting performance is the chosen language, and while FrameD is fully implemented in Python, the implementation of a component can be in any language as long as it has an interface to Python. We find this to be an acceptable strategy since the infrastructure of FrameD is only focused on moving information between components that perform a bulk of the computation, which can be handled by a higher performing language.

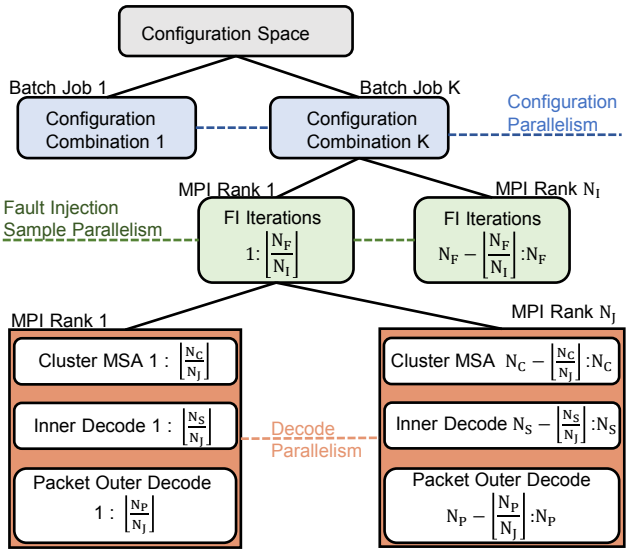

Fig. 3: Hierarchical parallelism leveraged by FrameD. Items grouped in the same level, e.g. MSA and inner decode, represent work that can be done in parallel using the same MPI ranks. N_F, N_C, N_S, N_P represent the total number of fault injection iterations, clusters, strands, and packets.

Choosing Designs with FrameD

To demonstrate FrameD we perform an analysis that serves as an example of how a DNA storage system designer may use the framework to determine the most cost-effective approach from a set of choices. In our example, we consider three different pipelines and two different error channel models, which are detailed in Table 1. Our encoder configurations are based on approaches taken in existing literature, and each leverages sequencing depth and resolves errors within a strand differently. The **RS** pipeline is representative of approaches that utilize conventional Reed-Solomon inner-error correction that deal with errors within strands post clustering and MSA (Antkowiak et al., 2020). Pipeline **HEDGES** contrasts with **RS** by directly resolving base errors within a strand with a convolutional code (HEDGEs) without applying MSA first (Press et al., 2020). Lastly, we consider an approach, **HEDGES-MSA**, that combines information aggregation of MSA with the convolutional inner encoding to remove remaining errors. For all MSA operations, we utilize Muscle (Edgar, 2004). Provided each approach leverages error correction and sequencing depth differently, we are interested in studying whether or not there are benefits as the sequencing depth and error rate of the system evolve.

DNA storage system designers are also faced with various technologies for writing/reading DNA, each with different error characteristics which can impact decoder choice. To demonstrate the evaluation of multiple error channels for the same pipelines, we consider two commonly used error models. One is a simple i.i.d model for insertions, substitutions, and deletions. This model

Table 1. Table of simulated parameters. For **RS** pipelines, the inner configuration tuples (x, y) indicate a $RS(2^8)$ configuration of x redundant bytes per strand and y data bytes per strand. Similarly, for the **HEDGES** configurations, an (x, y) tuple indicates a code rate of x and bytes per strand of y.

Name	Index Length	Binary Transformation	Transcoding	Outer Code	MSA	Depth	Inner Configuration	Error Model
RS	4 bytes	Randomize, $RS(2^8)$	$\Psi(\mathcal{B}_2^1; \mathcal{D}_4^1)$	$RS(2^8)$	Muscle	[3-30]	(1,55), (9,47) (14,42),(28,28)	i.i.d(1,5,10%), DNArSim
HEDGES	4 bytes	N/A	HEDGES	$RS(2^8)$	N/A	N/A	(0.167,4),(0.25,9) (0.5,24),(0.75,39)	i.i.d(1,5,10%), DNArSim
HEDGES-MSA	4 bytes	N/A	HEDGES	$RS(2^8)$	Muscle	[3-30]	(0.167,4),(0.25,9) (0.5,24),(0.75,39)	i.i.d(1,5,10%), DNArSim

is typically used when considering NGS DNA readout (Yuan et al., 2022; Press et al., 2020). However, it has been shown that an i.i.d channel does not well represent nanopore errors due to the lack of support for burst errors (Hamoum et al., 2021). To study whether burst errors may change the choice of decoder, we also consider a publicly available model, DNArSim (`https://github.com/BHam-1/DNArSim`), which describes conditional error probabilities that are derived from real sequencing data.

A key piece of FrameD that enables our analysis is parallelism support which allows us to simulate 240 unique pipeline configurations, totaling over 650M fault-injected DNA strands, in a reasonable time frame on a High-Performance Computing (HPC) cluster. All simulations were completed within 4 days on an HPC cluster with specs outlined in Table 2. To verify that our infrastructure aids in scaling to larger simulations, we perform scalability experiments on one pipeline configuration with the core type fixed to Intel Xeon *Gold6226R*. We picked the **RS (9,47)** pipeline with read depth 25x and fault rate 10%. For this study we simulate the pipeline only 368 times, and for a single core, we measured 20325s to complete the simulation. Because 1 MPI rank is allocated for scheduling, 31 additional compute ranks will populate all cores of a node. In this case the experiment finishes in 802s (25.3x speedup). In the previous cases, we are able to exclusively use a node with no outside job interference, however for multiple node runs we are not due to our cluster being a shared system. For 92 compute ranks, we measured a 65.99x speedup. These speedups indicate that FrameD is scalable.

We note that FrameD only provides the infrastructure for scalability, and that efficiently executing parallel units of work requires understanding the computational resources required needed by each rank. For example, developers should be aware if utilizing all cores of a node exhausts all memory when parallelizing multiple instances of the inner code, and thus allocate ranks to their compute system appropriately. In our studied pipelines we did not observe this need, however.

This experiment also relies on other properties of FrameD. Importantly, the error model modularity allows us to apply multiple error models to each pipeline, and allows us to adopt significant portions of DNArSim with minor modifications to fit FrameD's interface. This demonstrates FrameD's ability to incorporate existing model implementations. Probes also play a major role in this analysis by providing decoding success rates of inner error correction algorithms, a key value when determining storage system cost.

In all experiments, we control for strand length by modulating the number of bits of each base-sequence when we change the density of the encoding. This ensures that all designs fall in a strand length space that is reasonable given the practical limitations of DNA synthesis technologies (Bishop et al., 2017). In our experiments, strand lengths fall in the range of 240-242 bp. We use ideal clustering in our experiments to isolate MSA error correction from approximations made by clustering algorithms (Antkowiak et al., 2020; Rashtchian et al., 2017).

Table 2. Overview of HPC system used for simulations, parallelization parameters, and simulation characteristics. MPI parallelization was utilized for only fault injection iterations.

Node Architecture	2 Intel Gold 6226/6130 192 GB RAM per node
Number of Batch Jobs	240
MPI Ranks to Parallelize Fault Injection Iterations	128
Fault Injection Iterations/Pipeline	1024
Total Strands Generated	654.7 Million
Binary Data/Fault Injection Run	2.78 kB
Minimum Inner Code Samples	51,200

Comparing Pipelines

We compare pipelines by calculating the read and write code rates required for each pipeline to meet a target Mean Time to Failure (MTTF) of 10^9 accesses on 1 MB of data. We refer to these rates in terms of read and write density, each with units bits/base, and calculate them as the ratio of total bits read/written to the total number of bases read/written. Given that synthesis and sequencing cost is proportional to the number of bases, these metrics allow for technology-independent cost comparisons. A higher density is better. Write density is derived from the redundancy allocated to the inner and outer code. While read density considers encoding redundancy, it also considers the number of copies for each strand that were sequenced. We point out that the densities used for comparison are different from those provided in Table 1 for the inner codes. While the rate of the inner code influences the success rate of decoding individual strands for a given channel error, additional outer code error correction is required to develop a robust system to meet reliability targets. Thus, our final code rate is a single metric that factors in both the code rate of the inner codes shown in Table 1, and the code rate necessary for the outer code after measuring strand decode probabilities from fault injection. Detailed calculations can be found in the Supplementary Information section.

With this analysis relying on estimating the decode rate of strands through the inner code, there must be enough inner code

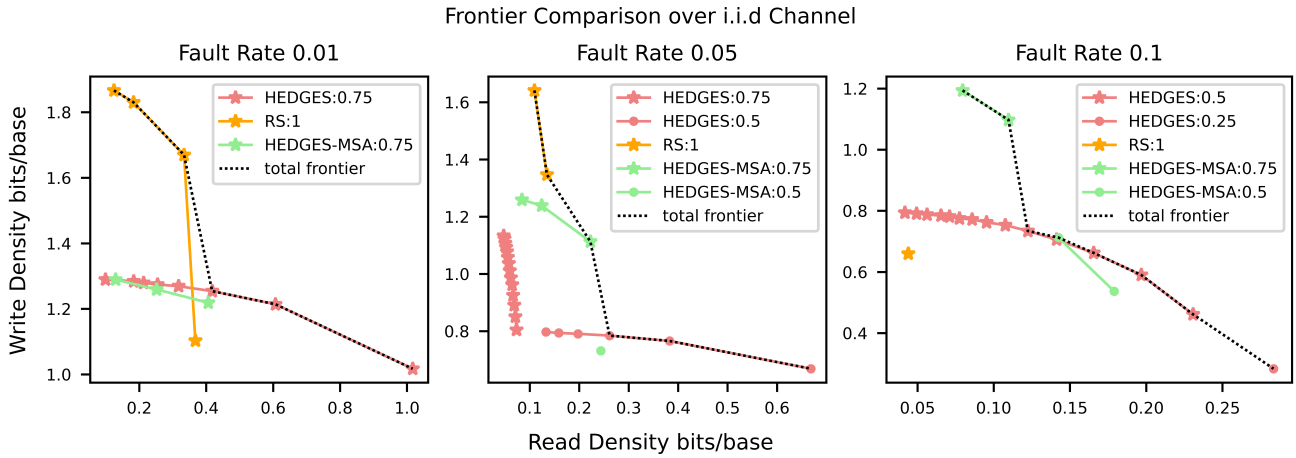

Fig. 4: Optimal density design points for the three studied pipelines across the three studied i.i.d error rates, 1%, 5%, and 10%. The dotted black line represents the total frontier composed of points from all of the studied pipelines. Each pipeline configuration is labeled by its inner-code redundancy.

samples to build reasonable confidence for these rates. Controlling for a consistent strand length across inner encoding densities leads to some pipelines simulating more strands given a constant input binary file. However, we ensure that each inner code is simulated against at least 51k fault-injected strands.

Lastly, our analysis does not consider the impact on the outer code from physical strand dropouts that can arise throughout the DNA storage life cycle (Tomek et al., 2019; Organick et al., 2018; Bornholt et al., 2016). FrameD supports dropout modeling, but our focus in this case study is on the inner code's ability to cope with errors in the channel. Thus, the outer code is entirely provisioned based on how well each inner-code is able to decode strands without error. However, our design choices hold when factoring in dropouts if assuming a fixed dropout rate per pipeline.

Results

To visualize the comparison of pipeline designs across sequencing depths, we plot the Pareto front for each pipeline with respect to read and write density. A Pareto front provides the design points that optimally trade off one cost for another such that a point is included if it provides an improvement in at least one cost. If a configuration offers no benefit, it is left out. Thus, every point in Table 1 will not make it to the frontier. The pipeline frontiers for the i.i.d error model for three error rates are shown in Figure 4. In this Figure, each pipeline has a different color, and a point shape represents a configuration of the inner code of the pipeline. The black line connecting points represents the complete frontier across all pipelines.

For the lowest fault rate (1%), we find that the **RS** pipeline with 1 redundancy byte for error detection provides the best write density. This configuration indicates that MSA is able to resolve a majority of errors. However, at a read density of 0.4 this pipeline's write density drops and gets overtaken in optimality by the **HEDGES** pipeline. This happens because MSA alone is not able to keep up with HEDGEs' error correction at lower read depths, requiring the **RS** pipeline to use considerably more outer encoding overhead. Interestingly, adding MSA to an inner code is not always best as shown by **HEDGES** enveloping **HEDGES-MSA**. The reason stems from the HEDGEs code high decode rate of single strands at this error rate such that it is more likely to decode a strand by applying the code multiple times rather than aggregating the information in MSA.

As the i.i.d rate increases, MSA-based approaches become more prominent. For example, when the error rate is 5%, **HEDGES-MSA** outperforms **HEDGES** for the same inner code configuration. This is because it is now more cost-effective to use sequencing depth to reduce the per-base error rate with MSA, rather than applying HEDGEs individually to each sequenced copy. The same occurs for a 10% error rate. Still, a pattern emerges where a non-optimal **HEDGES-MSA** approach becomes enveloped again by the **HEDGES** pipeline with the same configuration. We conclude from this that the optimality of using HEDGES with or without MSA is highly dependent on the error rate of the storage system, a conclusion a designer will not be able to come to without a simulation framework like FrameD.

A pattern that emerges for **HEDGES** configurations is that when read density is increased by decreasing sequencing depth, at a certain point it no longer becomes cost-effective due to ballooning outer code overhead, making lower density inner codes preferred. This can be seen for a fault rate of 5% between **HEDGES:0.75** and **HEDGES:0.5**. However, this is not the case for **RS**, as no configurations that utilize less dense Reed-Solomon codes for error correction appear in Figure 4. This indicates that Reed-Solomon as an inner code is ineffective against insertions and deletions. We demonstrate this further in the Supplementary Information.

Figure 5 compares the three pipelines for nanopore-based fault injection. In contrast with the i.i.d frontier, the complete frontier consists of points only from **HEDGES-MSA**. The main driving force of this is that nanopore sequencing has a higher frequency of burst errors compared to the i.i.d model. Burst errors generate a decoder mismatch with HEDGEs since this algorithm relies on guessing errors based on an i.i.d error model. Thus, HEDGEs experiences a large decode rate decrease unless MSA is applied before hand to help resolve bursts. Another interesting component of Figure 5 is that there is no configuration that just relies on MSA to resolve errors. These results showcase that with FrameD

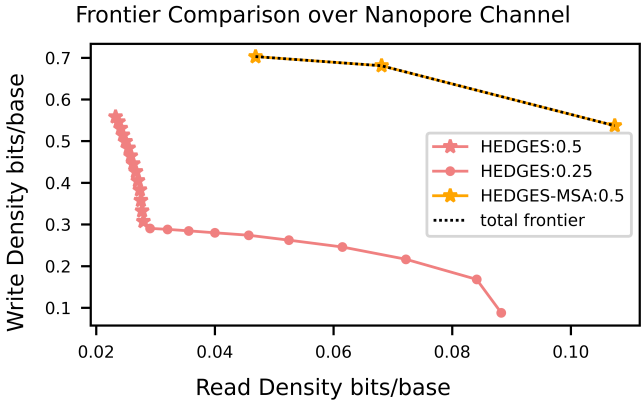

Fig. 5: Optimal frontiers obtained when the error injection model is based on nanopore sequencing technology.

designers are able to define weaknesses in decoding algorithms and determine better pipeline combinations when faced with designing for different sequencing devices.

Conclusion

We have shown that as work continues in the area of DNA storage systems there is an increasing number of unique pipelines that overlap in the components that they use. We introduce FrameD to address the void of tools available to the DNA data storage community, enabling the modularization of common algorithms and integrating fault injection models to provide a basis for fair system comparisons. Because of its foundation in the literature of DNA storage systems, FrameD provides designers with the ability to simulate a wide variety of storage systems. FrameD also provides transparent support for the parallelization of computational units of work such as individual strands and fault injection iterations, enabling the use of scalable high-performance computing systems. These features are demonstrated in our analysis of three pipelines that utilize the same components in different combinations across two error models representing different sequencing devices. In our analysis, the optimal pipeline choice and configuration depends both on the cost targets set by the designer and the target sequencing device. This highlights the basic need for DNA storage designers to have tools that can compare designs across a range of environments.

Funding Information and COI

This work was funded by the National Science Foundation [1901324 to K.V. K.L. A.K. J.T., 2027655 to K.V. K.L. A.K. J.T. P.H. W.T.].]. J.T. and A.K. are co-founders of DNAli Data Technologies. W.T. has two patents (8,748,091 and 8,394,584) licensed to ONT. W.T. has received travel funds to speak at symposia organized by ONT.

References

Anavy, L. et al. (2019) Data storage in DNA with fewer synthesis cycles using composite DNA letters, *Nature Biotechnology*, 37(10) 1229–1236.

Antkowiak, P. L. et al. (2020) Low cost DNA data storage using photolithographic synthesis and advanced information reconstruction and error correction, *Nature Communications*, 11 (1)5345.

Bishop, B., Mccorkle, N., and Zhirnov, V. (2017) Technology Working Group Meeting on Future DNA Synthesis Technologies, page 39.

Bornholt, J. et al. (2016) A DNA-Based Archival Storage System. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 637–649, New York, NY, USA ACM. ISBN 978-1-4503-4091-5.

Chaykin, G. et al. (2022) DNA-Storalator: End-to-End DNA Storage Simulator, *Non-Volatile Memories Workshop 2022*.

Choi, Y. et al. (2019) High information capacity DNA-based data storage with augmented encoding characters using degenerate bases, *Scientific Reports*, 9(1)1–7.

Church, G. M., Gao, Y., and Kosuri, S. (2012) Next-Generation Digital Information Storage in DNA, *Science*, 337(6102)1628–1628.

Edgar, R. C. (2004) MUSCLE: a multiple sequence alignment method with reduced time and space complexity, *BMC Bioinformatics*, 5 (1)113.

Erlich, Y. and Zielinski, D. (2017) DNA Fountain enables a robust and efficient storage architecture, *Science*, 355(6328)950–954.

Goldman, N. et al. (2013) Towards practical, high-capacity, low-maintenance information storage in synthesized DNA, *Nature*, 494 (7435)77–80.

Grass, R. N. et al. (2015) Robust chemical preservation of digital information on DNA in silica with error-correcting codes, *Angewandte Chemie (International Ed. in English)*, 54(8)2552–2555.

Hamoum, B. et al. (2021) Channel Model with Memory for DNA Data Storage with Nanopore Sequencing. In *2021 11th International Symposium on Topics in Coding (ISTC)*, pages 1–5.

Heckel, R. et al. (2017) Fundamental limits of DNA storage systems. In *2017 IEEE International Symposium on Information Theory (ISIT)*, pages 3130–3134. ISSN: 2157-8117.

Lin, K. N. et al. (2020) Dynamic and scalable DNA-based information storage, *Nature Communications*, 11(1)2981.

Marelli, A. et al. (2023) Integrating FPGA Acceleration in the DNAssim Framework for Faster DNA-Based Data Storage Simulations, *Electronics*, 12(12)2621.

Matange, K., Tuck, J. M., and Keung, A. J. (2021) DNA stability: a central design consideration for DNA data storage systems, *Nature Communications*, 12(1)1358.

Organick, L. et al. (2018) Random access in large-scale DNA data storage, *Nature Biotechnology*, 36(3)242–248.

Organick, L. et al. (2020) Probing the physical limits of reliable DNA data retrieval, *Nature Communications*, 11(1)616.

Press, W. H. et al. (2020) HEDGES error-correcting code for DNA storage corrects indels and allows sequence constraints, *Proceedings of the National Academy of Sciences*, 117(31)18489–18496.

Rashtchian, C. et al. (2017) Clustering Billions of Reads for DNA Data Storage. In Guyon, I. et al., editors, *Advances in Neural Information Processing Systems 30*, pages 3360–3371 Curran Associates, Inc.

Sabary, O. et al. (2021) SOLQC: Synthetic Oligo Library Quality Control tool, *Bioinformatics*, 37(5)720–722.

Tomek, K. J. et al. (2019) Driving the Scalability of DNA-Based Information Storage Systems, *ACS Synthetic Biology*, 8(6)1241–1248.

Tomek, K. J. et al. (2021) Promiscuous molecules for smarter file operations in DNA-based data storage, *Nature Communications*, 12(1)3518.

Yazdi, S. M. H. T., Gabrys, R., and Milenkovic, O. (2017) Portable and Error-Free DNA-Based Data Storage, *Scientific Reports*, 7(1) 5011.

Yuan, L. et al. (2022) DeSP: a systematic DNA storage error simulation pipeline, *BMC Bioinformatics*, 23(1)185.

Supplementary Information

FrameD Index Support

As previously explained, FrameD will provide automatic indexing support for outer codes if simple incremental counting indexes are used. To illustrate this, we provide a schematic in Figure 1 of the indexing that FrameD provides. Initially, FrameD breaks down a file into a set of packets. These packets are broken down into sub-packets. The number of sub-packets is determined by the number of base-sequences in the original packet, and how many base-sequences should be in each sub-packet, both of which can be specified by the user. At each sub-packet level a user specifies an outer encoding pass. The outer encoding pass provides protection to the initial sub-packets consisting of data by generating new sub-packets called **ECC Sub-Packets**. The manner in which ECC Sub-Packets are constructed is determined by the user's outer code such as XOR or Reed-Solomon. The process of sub-packeting is supported to occur an arbitrary number of times, where are each sub-packeting level more error correction is introduced to protect the smaller sub-packets. Once ECC sub-packets are generated, they are treated as any other sub-packet from the same level with respect to the following sub-packeting steps. Eventually after splitting sub-packets from each level, there will be a point reached

Table 1. Table indicating the approach that current DNA storage systems can take in order to utilize FrameD. To condense explanation, we use notations to generally represent the steps of information conversion that prior works take. We allow $\mathcal{B}_y^x$ to represent a set of length x base y integers, e.g $B_2^3 \subseteq \{0,1\}^3$. To differentiate DNA information from digital domain information, we let $\mathcal{D}_y^x$ to represent length x DNA strings of base y. Allowing for bases $y \neq 4$ is necessary of composite DNA works (Choi et al., 2019; Anavy et al., 2019). We abbreviate Reed Solomon codes over field F with $RS(F)$, and represent base changes of information as $\Psi(\mathcal{B}_{y_0}^{x_0}; \mathcal{B}_{y_1}^{x_1})$, where $\Psi(\cdot;\cdot)$ is a bijection map between the two sets of integers. To allow for variable length base changes used by Huffman codes, we can let x be a sequence of integers. For example, $\mathcal{B}_2^{\{3,4\}} \subseteq \{0,1\}^3 \cup \{0,1\}^4$ is a set consisting of both length 3 and 4 base-2 integers. $\mathbf{P}$ and $\mathbf{I}$ indicate transformations applied to Payload/Index respectively.

	Single Strand			Multi Strand		
Storage System	Binary Transformation	Transcoding	Functional Site Encoding	Outer Code	Consolidation	Reconstruction
Bornholt et al. (2016)	Parity Check, $\Psi(\mathcal{B}_2^8; \mathcal{B}_3^{\{5,6\}})$	$\Psi(\mathcal{B}_3^1; \mathcal{D}_4^1)$ Rotating Code	Prepend & Append Primers	Overlap Repetition, XOR	Detect & Remove	Index
Goldman et al. (2013)	Parity Check, $\Psi(\mathcal{B}_2^8; \mathcal{B}_3^{\{5,6\}})$	$\Psi(\mathcal{B}_3^1; \mathcal{D}_4^1)$ Rotating Code	N/A	Overlap Repetition	Detect & Remove	Index
Tomek et al. (2019)	$\Psi(\mathcal{B}_2^8; \mathcal{B}_3^{\{5,6\}})$	$\Psi(\mathcal{B}_3^1; \mathcal{D}_4^1)$ Rotating Code	Prepend & Append Primers	Overlap Repetition, XOR	Detect & Remove	Index
Lin et al. (2020)	N/A	Byte-to-DNA Map	Prepend & Append Primers, T7 Promoter	N/A	N/A	Index
Grass et al. (2015)	$\Psi(\mathcal{B}_{2^8}^2; \mathcal{B}_{47}^3)$, $RS(47)$	$\Psi(\mathcal{B}_{47}^1; \mathcal{D}_4^3)$	Prepend & Append Primers	$\Psi(\mathcal{B}_{2^8}^2; \mathcal{B}_{47}^3)$, $RS(47^{39})$	N/A	Index
Press et al. (2020)	N/A	HEDGES Convolutional Code	Prepend & Append Primers	$RS(2^8)$	Detect & Remove	Index
Erlich and Zielinski (2017)	$RS(2^8)$	$\Psi(\mathcal{B}_2^1; \mathcal{D}_4^1)$	Prepend & Append Primers	Fountain Code	Detection & Remove	Luby Seed
Antkowiak et al. (2020)	Randomize, $RS(2^6)$	$\Psi(\mathcal{B}_2^1; \mathcal{D}_4^1)$	N/A	$RS(2^{14})$	DNA Cluster and MSA	Index
Choi et al. (2019)	N/A	$\mathbf{P}(\Psi(\mathcal{B}_2^7; \mathcal{D}_6^3)$, Nuc. Deduction), $\mathbf{I}(\Psi(\mathcal{B}_{48}^1; \mathcal{D}_4^3))$	Prepend & Append Primers	$RS(2^7)$	N/A	Index
Anavy et al. (2019)	$\mathbf{P}(\Psi(\mathcal{B}_2^5; \mathcal{B}_6^2)$, $RS(7^3))$ $\mathbf{I}(RS(2^4))$	$\mathbf{P}(\Psi(\mathcal{B}_6^2; \mathcal{D}_6^2)$, Nuc. Deduction), $\mathbf{I}(\Psi(\mathcal{B}_2^2; \mathcal{D}_4^1))$	Prepend & Append Primers	Fountain	N/A	Luby Seed
Yazdi et al. (2017)	N/A	$\Psi(\mathcal{B}_2^{14}; \mathcal{D}_4^8)$ GC-balance Constrained Code with homopolymer checks	N/A	N/A	DNA Cluster and MSA	Index
Organick et al. (2018)	Randomize, $\Psi(\mathcal{B}_2^6; \mathcal{B}_3^4)$	$\Psi(\mathcal{B}_3^1; \mathcal{D}_4^1)$ Rotating Code	Prepend & Append Primers	$RS(2^{16})$	DNA Cluster and MSA	Index
Tomek et al. (2021)	$RS(2^8)$	$\Psi(\mathcal{B}_2^8; \mathcal{D}_4^8)$	Prepend & Append Primers, Restriction Enzymes	$RS(2^8)$	Detect & Remove	Index

where sub-packets are individual base-sequences. While ECC base-sequences can be added at this final level, no more sub-packeting is possible.

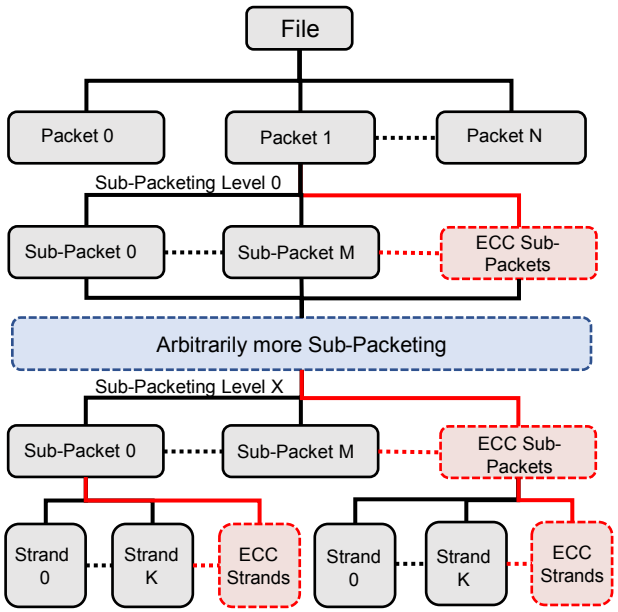

Fig. 1: Hierarchy that FrameD uses in order to allow arbitrary cascading of outer codes. During the outer code encoding process, FrameD infers index pieces to generate a complete unique identifier for each base-sequence.

FrameD handles the communication of information between sub-packet levels, really all the user needs to provide for an outer code is the number of ways a packet should be divided and algorithm to generate the ECC sub-packets given some starting data packets. If a user does not specify a terminating sub-packet level where the sub-packets are strands, FrameD will insert it automatically. While information is relayed between each level of the sub-packeting hierarchy, FrameD adds to each base-sequence the indexing information indicating what packet or sub-packet it is from. This indexing information is simply derived from a counter at each level that uniquely identifies each unit of information. Ultimately, the entire index consists of several smaller indexes, starting with the packet index and ending with the strand index. Consider a system that has 1 sub-packet level and a terminal strand sub-packet level. A base-sequence is uniquely identified by a 3-tuple of integers, with 1 integer for the packet, sub-packet, and strand. So, in general, for a base-sequence that is the kth strand of the jth sub-packet of the ith packet the index is (i, j, k). For outer code schemes that do not utilize a counter but rather use a random seed value to indicate data positions (fountain codes), they can still be supported by FrameD given that we allow any outer code to modify the index related to it's sub-packet level. However, such modification should be reversed on decoding to ensure that FrameD can reason about the resulting sub-packets and their ordering.

Decoding Details

For the most part, decoding in a DNA storage system is identical to encoding with the exception that the flow of information through passes is reversed. However, decoding has an additional problem that needs to be rectified, and that is how to reduce the dimensions of the input DNA data set that will typically consist of multiple reads per encoded DNA strand. There are two general approaches that can be taken. One is to first cluster the DNA strands input to the decoder based on similarity scores like edit distance or using a MinHash-based approaches (Antkowiak et al., 2020; Organick et al., 2018; Rashtchian et al., 2017). Then MSA algorithms, such as Muscle, can be used to aggregate information across strands and help resolve errors through consensus voting (Antkowiak et al., 2020; Edgar, 2004; Yazdi et al., 2017). Because clustering and alignment are usually discrete steps, we build a sub-model for DNA consolidation as shown in Figure 2.

Another approach is to consolidate the strands after completing the inner code, throwing out strands that may violate error checks, and coming to a consensus on the digital representation of information. FrameD supports the use of either approach, or even both. The only real difference between this approach and a DNA-based approach is that it would be placed after each strand is passed through inner decoding in Figure 2. Also, the clustering in this case is also trivial given that indexes are known after the inner decode passes.

Detailed Probe Example

Figure 3 provides a detailed illustration of an implementation of a probe that calculates error rate in terms of edit operations by way of calculating the edit distance between a fault injected DNA strand and its pre-injected version. In this illustration, we represent the state of a strand that is being manipulated as a **Strand Object**. This object holds attributes that represent pieces of a strand's information, such as its binary information and DNA information representations. The **DNA** field is initially empty until the strand passes through the Transcoding's encoding pass (top of Figure 3) which populates this field. After which, the edit distance probe takes a snapshot of this field and generates a copy that is placed in an attribute **DNA′** that can be referenced at a later time.

At the bottom of Figure 3 is the decoding pass which is applied after faults have been injected into DNA strands. The injected errors are represented by the red-highlighted letters in the **Strand Object**'s **DNA** attribute. Note that the **DNA′** is still in the **Strand Object**, and not modified. When this **Strand Object** passes through the **Edit Dist. Probe**, the probe is able to compare the now corrupted **DNA** field with its non-corrupted version in **DNA′** to determine the errors and their locations. This data can be transferred to FrameD's statistic tracking support which allows for the statistics to be propagated to the simulation's output files.

Decode Pipeline MPI Communication Pattern

In previous sections, we explained what units of work FrameD finds for parallelization using MPI. While FrameD handles the movement of information to facilitate parallelization for many steps of the decode process, clustering is a step in which we actually provide the user the flexibility to write their own communication pattern. For a user to write their own communication patters, it is necessary to understand the

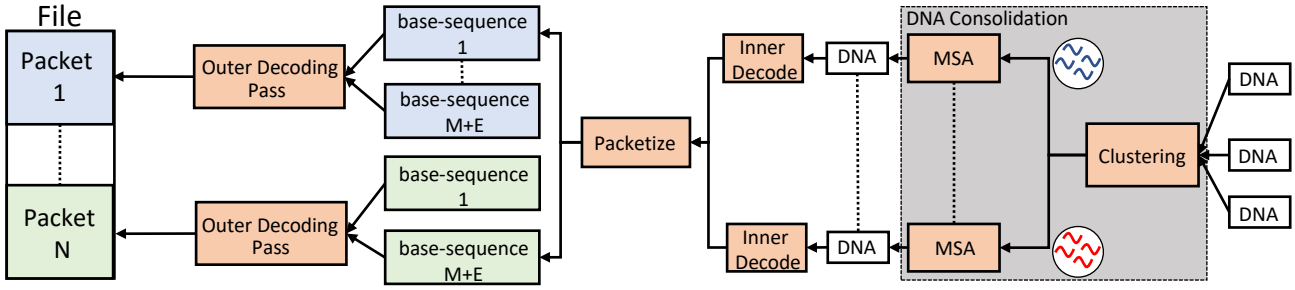

Fig. 2: Model of decoding for FrameD. Decoding is typically just the reversal of the encoding process, but DNA storage systems must address strand copy numbers. In this example, copies are consolidated by clustering the DNA strands and performing multi-sequence-alignment (MSA) on the result. In this figure Inner Decode represents the reverse operation of the Inner Encoding of Figure 1.

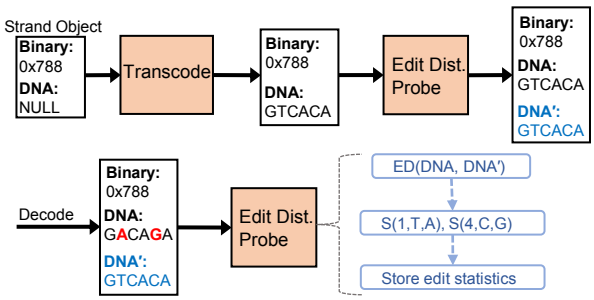

Fig. 3: Example of generating edit rate statistics in FrameD.

surrounding communication patterns to this step so that data is in an appropriate state for each rank. Figure 4 illustrates the transformation and communication of information from the time of instantiating a decode pipeline to the final point of writing packets to a file.

FrameD starts with all information in its original DNA state in Rank 0. This information is initially scattered across all ranks in the MPI communicator allocated for the decoding pipeline. The strands allocated to each rank are initially passed through a process that reverses the DNA modification steps. This is done first so that regions like primers that may indicate a certain file can be filtered, and so that other possible inserted DNA regions can be removed before reverse-transcoding. There are now two scenarios for processing the DNA strands, either clustering is done or it is not. If it is not, timeline of Figure 4 skips the steps in between the orange dashed lines which are only used when clustering DNA strands. In this case, the inner decode steps are processed in parallel for each DNA strand. After all strands are decoded for each rank, they are gathered into Rank 0 which places each base-sequence in its appropriate packet as indicated by the base-sequence's index. After packetizing the individual pieces of information of each DNA strand, the packets are scattered across ranks where each packet is decoded using the outer code. Finally, all packets are gathered back at Rank 0 so that information can be written to a file to output the information stored in DNA.

The steps in between the dashed orange line of Figure 4 correspond to the steps within the DNA-consolidation model of clustering and subsequent multi-sequence alignment. Initially, strands are gathered at Rank 0. We do this before clustering begins for several reasons. One, while we allow parallel clustering algorithms to be written for FrameD users may also want to write serial clustering algorithms while still utilizing FrameD's automatic parallelization. So, all strands will need to be on a single rank that will run the code for clustering, and in this case we assume Rank 0 for that role. Second, it establishes simple assumption that the user can make about the whereabouts of strands when they want to distribute strands according to their algorithm. Third, the implicit distribution of strands may not even be appropriate for a user's parallel clustering algorithm. Within the clustering algorithm, developers are free to utilize the provided MPI communicator in any way that they need to communicate information. After clustering, clusters should be placed in Rank 0 so that they are in a location that FrameD is aware of. Finally, given clusters represent a single DNA strand and a single piece of independent information, FrameD scatters the clusters before they are processed with MSA algorithms. The resulting strands at each rank from MSA are kept at each rank before the inner decode process since there is no need to gather and scatter again.

Read and Write Cost Methodology

We evaluate the write and read cost of all of the studied pipelines from an information density standpoint. That is, we assume that the read cost and write cost of the system is solely determined by the density in which data can be sequenced and synthesized. We make this assumption because at the moment, the cost to sequence and synthesize data is the main cost factor of a DNA storage system. We recognize that supporting infrastructure will be necessary for a DNA storage system, like compute to implement the decode process at a desired throughput. However, the wide range of possible algorithms, their unique complexities, and possibly different compute paradigms (single-thread/multi-thread/GPU), makes it difficult to provide a complete cost analysis that factors in compute. Furthermore, performing an information-based analysis may provide insight into possible directions in which future computational research should emphasize so that approaches with good information density can be achieved computationally efficiently.

While FrameD allows for parameter sweeps for encodings to optimize error rates, it is computationally inefficient to determine exactly the required outer code error correction for a given error rate and sequencing depth. So, we use results from targeted fault injection runs to build a simple analytical model that provides the probability of decoding a file for a given outer code and a given strand drop out rate if desired. From this, we pick outer

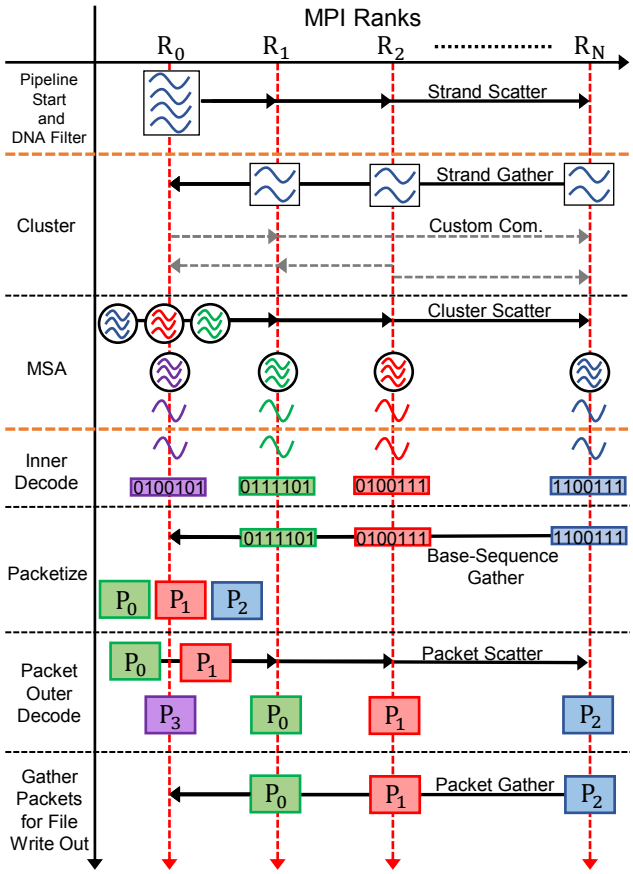

Fig. 4: Timing diagram of the representation of information in FrameD along with communication patterns used to parallelize the decode process using MPI. Events are ordered vertically in time for each MPI rank (dashed red lines) and are described on the left side between black/orange dashed lines. Items on each red line indicate the representation of information at a point in time. Curved lines indicate DNA representation, rectangular boxes with binary numbers indicate binary base-sequences, and rectangles labeled P_x indicate a packet of binary information.

code parameters that force this probability towards 1. Because approaches that use MSA or detect and throw out erroneous strands treat sequencing depth differently, we do two separate analyses.

We first begin with an approach that detects erroneous strands and removes them from decoding. Let $p(\text{drop-out})$ be the probability that an encoded strand does not get sequenced, and let $p(\text{decode}|e)$ be the probability that a strand decodes for a given error rate e. Assuming a constant sequencing depth d for a given strand that does not drop out, let n be the number of strands successfully decoded from the sequencing depth, then $p(n > 0|d, p(\text{decode}|e)) = 1 - (1 - p(\text{decode}|e))^d$ is the probability at least 1 strand is decoded successfully. Factoring in strand drop outs,the total probability that a strand with index I makes it to the outer decoder is $p(\text{I exists}|e, d, p(\text{drop-out})) = p(n > 0|d, p(\text{decode}|e)) \cdot (1 - p(\text{drop-out}))$. Assuming an outer code like Reed Solomon, and given D data symbols in a given encoded codeword, and all indexes that are not decoded successfully are correctly detected, then the outer code will succeed as long as at least D symbols are recovered from the total $N = D + E$ symbols where E is the additional error correction added to the set. The number of recovered symbols M will be distributed according to a Binomial distribution $B(N, p(\text{I exists}))$. Thus the probability that a block decodes is:

$$p(M \geq D) = 1 - \sum_{k=0}^{D-1} \binom{N}{k} p(\text{I exists})^k (1 - p(\text{I exists}))^{N-k} \quad (1)$$

So, for any file with X blocks, the probability to decode a file is succeeding on each block that is decoded independently, e.g. $p(\text{File decodes}) = p(M \geq D)^X$. From this analysis, we can see that there are two main unknowns. One is $p(\text{decode}|e)$ which depends both on the error rate and the decoding algorithm used. This value is the main target of our fault injection studies for each given pipeline. Another unknown is d, the number of reads per strand. In our analysis we assume that every strand that is not dropped out has exactly d reads. This is different than modeling some distribution over d, but we choose to assume a constant d for each strand because this provides a more accurate assessment of the amount of information that will need to be read during sequencing. Furthermore, distributions on the number of reads per strand is very process dependent (Bornholt et al., 2016; Organick et al., 2018; Tomek et al., 2019; Organick et al., 2020), and so an assumption of a distribution here may not provide generally useful results. To finally determine d, we sweep over a range of values to a get a set of $p(\text{I exists})$. With each of these probabilities we then sweep over a range of Reed-Solomon configurations each with a different value for E and D. We ultimately pick the smallest E such that the file can be recovered with a certain mean time to failure (MTTF). In our analyses we fix the file size to be $1MB$ and MTTF to be 10^9 reads. The MTTF for some number of blocks and Reed-Solomon configuration is defined as:

$$\text{MTTF} = \frac{1}{1 - p(M \geq D)^X} \quad (2)$$

For an approach using MSA, the outlined approach stays the same except that taking into account sequencing depth becomes different. Now, instead of individual reads being independently decoded, their information is aggregated using a MSA algorithm, so the probability that an index exists will be $p(\text{I exists}|e, d, p(\text{drop-out}), \text{MSA}) = p(\text{decode}|e, \text{MSA}, d)) \cdot (1 - p(\text{drop-out}))$. Where $p(\text{decode}|e, \text{MSA}, d)$ is the probability of successfully decoding d reads that have been aligned via MSA.

The outlined approaches provide a pathway to estimating the number of strands that are encoded and also the read depth for each encoded strand. However, to compare costs of different approaches, we normalize to a *bits/base* value for both reading and writing. This normalization is necessary because read costs are impacted not only by depth of sequencing, but also by the density of the encoding as well since this impacts total strands in the set of strands read. We define the write density with Equation 3 and the read density with Equation 4, where read density can be written as write density divided by the number of reads made per strand d:

$$\text{Write Density} = \frac{|F|}{\text{Total DNA Strands} \times \text{Length of DNA Strand}} \quad (3)$$

$$\text{Read Density} = \frac{\text{Write Density}}{d} \tag{4}$$

In Equation 3, $|F|$ represents the total number of bits that are encoded in some set of DNA strands.

Reed Solomon Inner Codes

To demonstrate why Reed Solomon inner code configurations that are less dense do not appear in Figure 4, we use Figure 5. In this figure we plot the write density that can be achieved for various strand decode probabilities. Each line represents the number of bytes of data that are allowed in an individual strand, and there is a line according to each configuration of **RS** in our experiment. Using these lines, we plot 4 points corresponding to write density and decode probabilities that have been observed for the 4 different density configurations when considering an i.i.d channel of 5% error rate. The blue star-point corresponds to configuring the Reed Solomon inner code with just error detection for a read depth of 10 reads per strand, and the other points correspond to all studied Reed Solomon inner code configurations for a read depth of 5 reads per strand.

As was pointed out in Figure 4, there is typically an eventual benefit in less dense inner encodings providing higher read densities when the read depth is decreased. A benefit in read density can come from 2 sources: increasing write density and decreasing read depth, as shown in Equation 4. Thus a less dense code can offset decreases in decode probability when read depth decreases, lowering the amount of outer code and subsequently increasing write density. Alternatively, a less dense code can also make lower read depth designs attainable with respect to MTTF metrics.

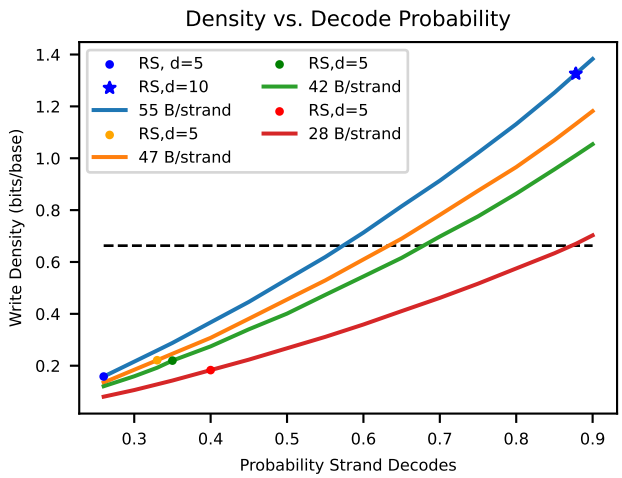

Fig. 5: Relationship between write density and the probability that an index exists for different amounts of bytes/strand.

However, this is not the case for the Reed Solomon inner code. The blue star-point of Figure 5 represents the rightmost point of Figure 4 for the 5% i.i.d channel. For this frontier to be extended, one of two things has to happen. Either a lower inner code density will provide better write density for the same read depth, in this case $d = 10$, or read depth is decreased. The former is not possible, given that the points in the frontier are optimal, a point not in the frontier that increases read and write density is not possible. Thus, we need to look towards smaller read depths. However, we found that as read depth decreases it is offset by extremely low write density for this pipeline. Given read density is the ratio of write density to read depth, an increase in read density is only possible if the ratio increase. This is illustrated by Figure 5, where we look at points that cut read depth in half from 10 to 5. Given a factor of 2 decrease in read depth, the write density must be higher than the black cut-off line that is placed at half the write density of $d = 10$. It is clear that every point for $d = 5$ falls under this cut-off, and thus no design is worthwhile. This showcases even more that Reed Solomon inner codes are not suitable for insertion/deletion channels and DNA data storage.

Additional Run Time Analysis

Here we provide more insight into the performance and runtime characteristics of all pipelines we simulated for this work. While in principle we could compare the parallel performance against that obtained on a single core, it would be intractable to run every pipeline on a single core sequentially. Instead, we analyze how uniformly each mpi-rank executes its assigned load measured by execution time. Each fault injection campaign logged key events such as when an iteration began and ended, and information such as the name of the host node where the rank executed.

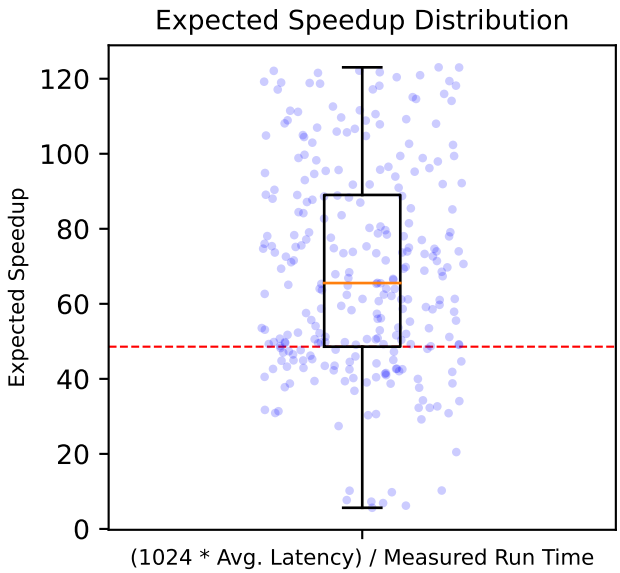

Fig. 6: Distribution of expected speedups calculated between average iteration latencies and the total measured run time. The red horizontal line plots the 25th percentile of the distribution.

Using the information within our logs, we calculated an average execution time across all 1024 iterations of a given pipeline. We also determine the difference between the start of the first iteration and the completion of the last iteration to finish and use this as the total time to complete all iterations. Using the average and the total difference in time for each pipeline, we calculate an *expected speedup* by multiplying the average by 1024 (total number of iterations) and dividing by the measured total difference in time. The distribution of this value is plotted in Figure 6. We found that

this statistic ranges in value from 5.63 to 123.04. Our scheduler allocated equal work to each rank at launch time (with 128 mpi-ranks and 8 iterations per rank). Speedups close to 128 indicate near-ideal distribution of work and performance scaling, while speedups much lower imply that some ranks performed far worse than average. Overall, we find that 75% have expected speedups over 48.57, and we conclude that many simulations benefited significantly from parallelization, but some inefficiencies remain and may benefit from additional optimization.

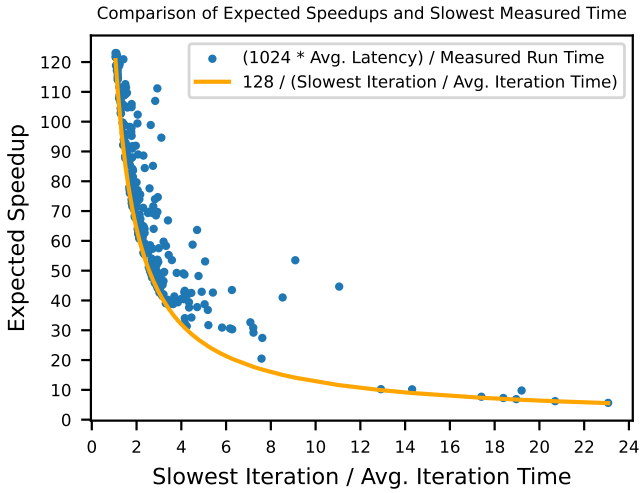

Fig. 7: Analysis of run times of all pipelines studied in this work.

The large variation in *expected speedup* could imply that some specific pipeline configurations perform poorly. However, we did not find any such correlation nor could we identify specific hosts in the cluster that consistently under-performed. Instead, we suspect the culprit is co-execution with other jobs on the cluster. We ran our simulations on shared compute resources used by many other research groups at NC State, and for our runs, we specified to the job scheduler that our mpi-ranks could be co-scheduled with another user's job. This can create scenarios where our job competes for CPU resources, such as cache, memory bandwidth, or memory capacity with other nodes. In such scenarios, one or both jobs scheduled on the node may be negatively impacted. To this end, we study the ranks that have the slowest iteration and determine its relationship with the expected speedups.

Figure 7 takes the data of Figure 6 and plots it against a value we calculate as Average Iteration/Slowest Iteration (blue points). From Figure 7 we can see that there is a strong inverse relationship between the expected speedup and this ratio. This provides evidence that slow ranks are a driving force in low expected speedups. To further investigate this, we take another approach to calculating an expected speedup. In this approach, we consider the ideal speedup of 128 and estimate how much we expect this to degrade by dividing by the ratio of Average Iteration/Slowest Iteration. The reasoning behind this approach is that it provides an estimate of what we expect the speedup to look like if at least one of the ranks executes all work at this slowest iteration pace. We plot this value in Figure 7 as a solid orange line. We see a significant clustering of blue points to the line, implying that slow ranks tend to stay slow.

Table 2. Analysis of the number of iterations per rank that fall in the 5% of iterations that have the longest execution time. The 5% slowest iterations are determined relative to each individual experiment.

Number of Iterations per Rank Executing Slowest 5% Iteration	Percentage of Total Ranks
0	86.71
1	5.84
2	1.94
3	1.15
4	0.99
5	0.68
6	0.49
7	0.48
8	1.73

Lastly, we consider whether these slow ranks are common. If they are outliers, it will imply that inefficiencies in our experiments are likely an artifact of the cluster we are using and that we may be able to improve scalability in these shared compute environments by updating the scheduler to avoid ranks that are executing slowly. To directly determine this, we take each of the ranks of a simulation and we determine the number of iterations within that rank that fall within the slowest 5% of all iterations. Table 2 summarizes this analysis for every rank of every simulation. Values in the first column indicate how many iterations that a rank executes which fall within the slowest 5%, and it varies from 0 to 8 since 8 iterations are assigned to each rank. The values in the second column report the percentage of all ranks that fall in that category. This data shows that a large majority of ranks (86.71%) do not execute any iterations that fall in the slowest 5%. However, we do find that there is a small group of ranks (1.73%) in which all 8 (the last row of the table) of their iterations are within the 5% slowest iterations. From this we conclude that inefficiencies in our experiment executions are likely caused by a small set of slow outlier ranks.

Thus, given the evidence that slow ranks tend to stay slow and the information that the slow ranks tend to be considerable outliers, we conclude that FrameD could see improvement by adapting the scheduler to allocate less work to ranks running slowly.