Direct-Coding DNA with Multilevel Parallelism

Caden Corontzos and Eitan Frachtenberg, Reed College

***Abstract*—The cost and time to sequence entire genomes have been on a steady and rapid decline since the early 2000s, leading to an explosion of genomic data. In contrast, the growth rates for digital storage device capacity, CPU clock speed, and networking bandwidth have been much more moderate. This gap means that the need for storing, transmitting, and processing sequenced genomic data is outpacing the capacities of the underlying technologies. Compounding the problem is the fact that traditional data compression techniques used for natural language or images are not optimal for genomic data.**

To address this challenge, many data-compression techniques have been developed, offering a range of tradeoffs between compression ratio, computation time, memory requirements, and complexity. This paper focuses on a specific technique on one extreme of this tradeoff, namely two-bit coding, wherein every base in a genomic sequence is compressed from its original 8-bit ASCII representation to a unique two-bit binary representation. Even for this simple direct-coding scheme, current implementations leave room for significant performance improvements.

Here, we show that this encoding can exploit multiple levels of parallelism in modern computer architectures to maximize encoding and decoding efficiency. Our open-source implementation achieves encoding and decoding rates of billions of bases per second, which are much higher than previously reported results. In fact, our measured throughput is typically limited only by the speed of the underlying storage media.

***Index Terms*—DNA encoding, Parallel architectures**

I. Introduction

The size of sequenced genomic data in the world doubles roughly every 18 months, and has already grown to petabyte-scale [1]. Storing and manipulating these sequences, which can be several gigabytes long for whole genomes, requires significant storage and computing resources [2]. Moreover, Next-generation sequencing (NGS) techniques produce millions of short reads, which are less amenable to compression algorithms that exploit the redundancies in longer sequences [3]. Dozens of lossless compression techniques have been proposed for DNA sequences, with varying tradeoffs between compression ratios, encoding and decoding speed, and memory consumption [4]. These tradeoffs matter because in many applications, throughput and ease-of-processing of the encoded stream are critical [5].

When time and memory constraints are the top priorities for a DNA encoder, there is an encoding technique that simply maps each of the four possible base letters (‘A’, ‘C’, ‘G’, or ‘T’) to a unique 2-bit sequence. This encoding is not a compression method per se, even though the trivial 4:1 size reduction it achieves from ASCII files is not far from that achieved with more sophisticated approaches. Standard substitution-based (e.g., Lempel-Ziv) and entropy-based (e.g., Huffman) encoding methods rarely significantly outperform the 4:1 compression ratio of direct coding [6]. More specialized DNA-specific approaches often achieve better size reduction either at the cost of significant computation resources or by referring to a baseline genome that brings its own limitations and disadvantages [2].

In this paper, we explore the idea of storing raw DNA sequences using a bijective mapping to two bits per symbol, which is particularly useful when extremely fast compression, decompression, access, or search is necessary; when random-access to any base in the encoded stream is desired; or when combined with specialized compression techniques. There exist multiple competing text-based formats for raw DNA sequences that use the same ASCII-based base encoding, such as FASTA, EMBL, GCG, GenBank, and IG, as well as binary formats such as BAM [2], [7]. Text-based formats typically include various metadata in addition to base sequences, which are not salient to our discussion of hardware-accelerated binary encoding, so we focus instead on plain-text sequence data, as shared and analyzed in comparable studies [8]. Nevertheless, our techniques can be easily applied to the long sequences of bases between segments of metadata in the more complex formats.

Motivation and Contribution

The main motivation for our work is to balance the ease of processing and interactivity of raw sequences with the moderate storage efficiency of two-bit encoding. As alluded to earlier, there are dozens of general and specialized compression algorithms for DNA sequences, but their resource requirements can be significant: a recent comprehensive benchmarking effort compared dozens of these techniques and found that many encode or decode at a rate well under 1000 MB/sec while consuming gigabytes of RAM [9].

On the other hand, two-bit encoding offers significant efficiency and simplicity advantages. Consequently, it forms the core of compression programs such as GenBitCompress [10] and also the basis for more sophisticated algorithms such as GenCodex and others [11]–[15]. However, these implementations still tend to be naive and highly serialized [16], [17]. In contrast, our novel implementation combines the storage efficiency of binary data formats with the interactivity of ASCII formats, while providing “just-in-time” conversion latencies and using few resources. The main contribution of this paper is an implementation of two-bit encoding that exploits multilevel parallelism in modern architectures, encoding DNA at speeds near the underlying hardware’s bandwidth with nominal memory consumption.

II. Evaluation

Our C++ implementation was evaluated on a desktop machine running Linux v. 6.2.0-32 and gcc v. 13.1.0, using a 16-core AMD 5950X CPU, 128 GiB of DDR4-3200 RAM, and Samsung 980 PRO 2TB SSD for primary storage.[1] All experiments were run 100 times to observe their variability and preceded by a Linux buffer (page cache) flush to ensure I/O cost is included in the measured run time.

Direct-coding takes the exact same code paths regardless of the input, so encoding and decoding time does not depend on the actual bases in the input. We can therefore generate input of arbitrary size with a random selection of bases. Further, to compare the encoder and decoder on equal terms, we measure throughput in millions of input bases per second (MBase/s) instead of MB/s, since the decoder’s input file is a quarter of the size of the encoder’s in bytes. With these preliminaries, we can now evaluate the performance implications of different levels of software and hardware parallelism mechanisms.

A. Baseline sequential version

As a basis for comparison, we start with a straightforward sequential implementation of the encoder and decoder using lookup tables (LUT). The encoder’s loop looks similar to this:

Corresponding author: eitan@reed.edu

[1]Source code and experiments are available at github.com/eitanf/fastdna.

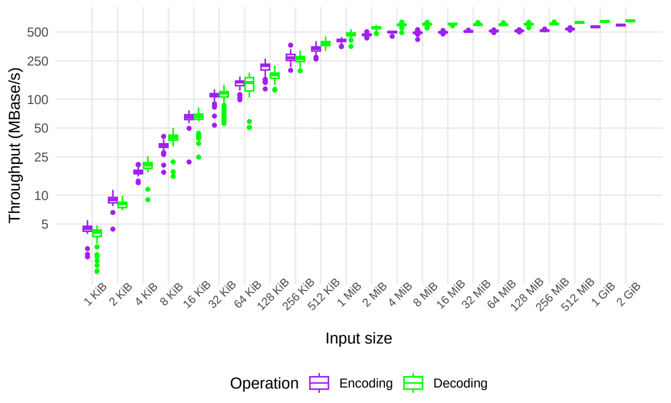

Fig. 1. Distribution boxplot of 100 runs of the encoder and decoder across increasing input sizes (log-log scale)

```
// Encode every 4 bases into one byte
// BMAP maps from 'A','C','G','T' to 0,1,3,2
void encode(char* in, size_t size, char* out) {
  for (i = 0; i < size; i += 4) {
    *out   = BMAP[*in++];
    *out   = BMAP[*in++] | (*out << 2);
    *out   = BMAP[*in++] | (*out << 2);
    *out++ = BMAP[*in++] | (*out << 2);
  }
}
```

The baseline decoder uses a similar loop that maps each byte of input into four bytes (bases) of output, again using a 256-entry compile-time LUT. Any leftover bases beyond the largest multiple of 4 are handled as a special case.

Even this simple baseline implementation runs significantly faster than typical compressing DNA encoders [9], with a maximum encoding throughput of over 500 MBase/s (Figure 1). Most results are narrowly centered around the median. Larger input sizes amortize the overhead of file I/O and setup/teardown, so they reflect better the performance potential of the algorithm. Larger files also hide better the deleterious effects of system-level delays (noise). These slower runs can produce outlier points, reflecting transient effects in the system such as context switches and I/O interference. But their effect is only noticeable for small inputs (up to a few KiB), where their relative magnitude is comparable to that of the overall run time. As the input size grows, the effect of the noise grows negligible. By the time input size reaches 1 GiB, throughput has largely converged and the standard deviation of the run time falls below 1% of the mean. We will therefore use an input size of 3 GiB (full human genome size [6]) for the remainder of the evaluation, for which this baseline implementation yields a median encoding throughput of 589 MBase/s and median decoding throughput of 659 MBase/s.

B. Device-level parallelism

The encoding and decoding work takes place on the CPU while the I/O takes place on a separate physical device, the SSD. In the baseline implementation, these phases happen at separate times, which means that while one device is busy, the other is idle. We can overlap the writing of outputs to disk with computation if we divide the computation into chunks of the input, and issue writes to each chunk of output while starting to compute on the next chunk. The best chunk size to maximize this overlap depends on the relative speeds of the computation and I/O, so we measured the throughput for multiple chunk sizes. As Figure 2 shows, synchronous encoding and decoding on this platform is fastest at a chunk size of 128 KiB, with

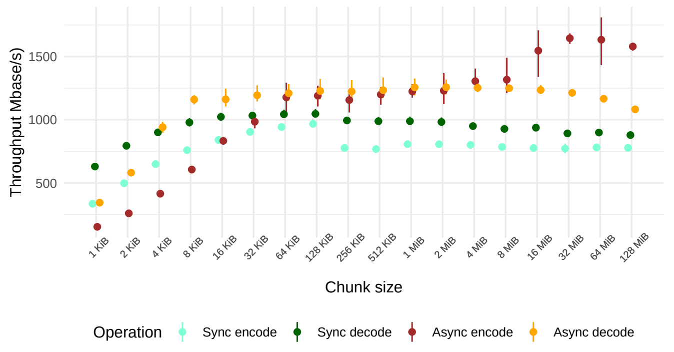

Fig. 2. Median throughput vs. I/O chunk size with both synchronous and asynchrnous I/O (100 runs; 3 GiB input; 95% CI; logarithmic x-axis)

a median encoding throughput of 967 MBase/s (64.3% improvement over Baseline), and a median decoding throughput of 1047 MBase/s (59% improvement).

The overlapping can be extended to reads as well as writes by using the POSIX asynchronous I/O (AIO) library. This library maintains a thread pool to execute I/O requests asynchronously [18]. By carefully orchestrating an active and standby buffer for reads, and another pair for writes, we can rewrite the code so that a previous chunk is written and a future chunk is read while a current chunk is being encoded or decoded. As the results in Figure 2 show, the effort is worthwhile. For example, at a chunk size of 32 MiB, the median encoding bandwidth increases to 1317 MBase/s (36.2% improvement over chunked I/O).

For larger buffer sizes, results grow noisier and require larger input sizes to benefit from this approach. These limitations to generality motivate our next approach, namely, multithreading.

C. Thread-level parallelism

To address the limitations of asynchronous I/O, we turn to multithreading to exploit parallelism both in the CPU and the SSD. Note that multithreading obviates the need to use AIO, because we already reached maximum SSD throughput, so we reuse the simple chunked-I/O implementation with a chunk size of 128 KiB. Our implementation splits the input and output roughly equally among the desired number of threads, and each thread encodes/decodes in its own memory buffers.

The threads use no locks and share no data or resources other than the input and output files, which they each address at independent offsets. One would expect near-perfect scalability of such a model, but in reality, the single shared resource, disk I/O, quickly becomes a contention hotspot (Figure 3). By 8 threads, median encoder throughput has peaked at 3352 MBase/s. This value is close to the 3150 MiB/s bandwidth of the underlying SSD, as measured by the "fio" benchmark using the command `fio --loops=10 --size=3G --name=bw --bs=128k --numjobs=8`. We verified that the I/O bandwidth is indeed the limiting bottleneck by running the same experiment on a different SSD (Intel SSDPEDMW012T4), where fio reports a bandwidth of 2050 MiB/s, and the median encoder throughput was 1818 MBases/s. Decoding throughput is limited even further by the slower write I/O bandwidth.

To expose the actual limits of the code's performance, we work around the I/O bottleneck from this point on. In the next experiment, the input is assumed to already be loaded in memory, and the output goes to another memory buffer (Figure 4). Without the SSD contention, the only shared resource among the threads is RAM access, so scaling is significantly better: at 8 threads, the speedup relative to 1 thread is a respectable 6.23×. Note that the maximum encoding

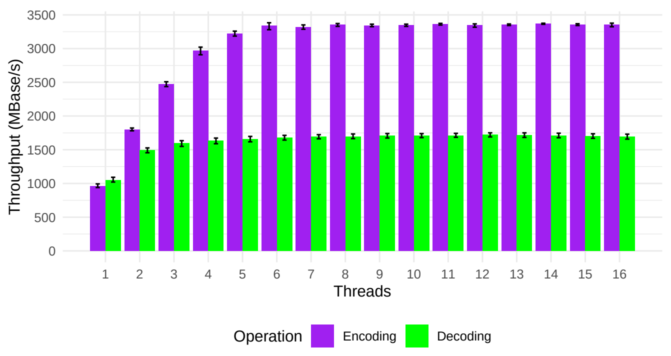

Fig. 3. Median throughput with multithreading (100 runs; 3 GiB input; 128 KiB chunks; 95% CI)

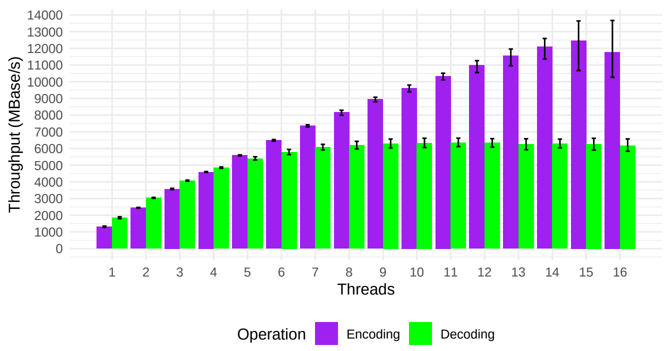

Fig. 4. Median throughput with multithreading and no I/O, just RAM input and output (100 runs; 3 GiB input; 95% CI)

throughput—although much improved at 10,984 MBases/s—is again limited by the storage medium's performance, in this case the DDR4 3200 bandwidth of $\approx 12,300$ MiB/s, as measured by running the `mbw -t2 3000` benchmark. By 16 threads, relative speedup has dropped to only 8.98×.

Again, we confirm this bottleneck by removing it, i.e., by encoding smaller chunks that fit in the CPU cache. As Figure 5 shows, smaller chunks indeed improve throughput but not dramatically, suggesting there is still room for more performance.

D. *Vector- and bit-level parallelism*

We can further improve the code's efficiency by exploiting the fact that all the encoder needs from each input byte are two bits

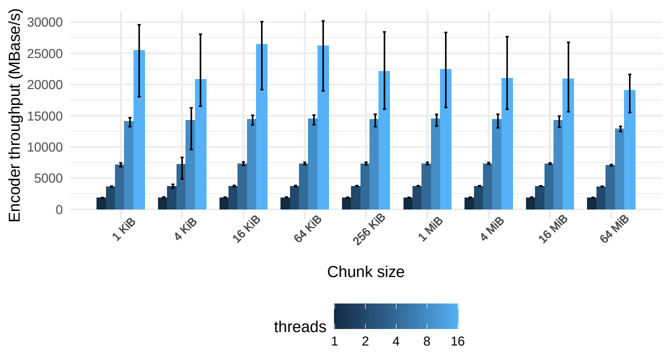

Fig. 5. Median encoder throughput with multithreading and no I/O (100 runs; 3 GiB input; 128 KiB chunks; 95% CI)

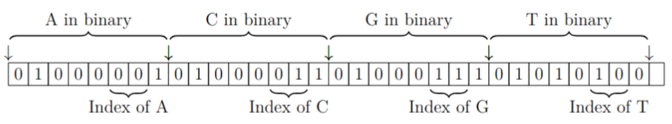

Fig. 6. Direct mapping from ASCII bases to two bits

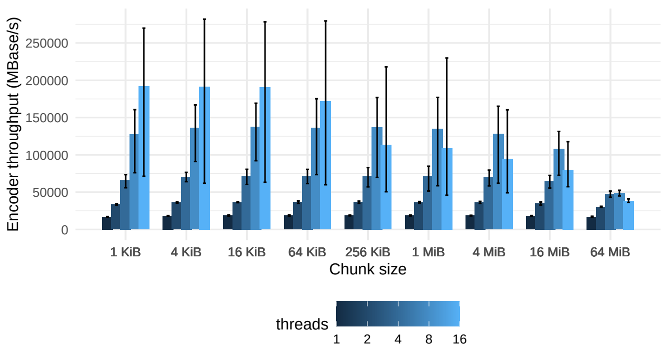

Fig. 7. Median encoder throughput using pext (100 runs; 95% CI; no I/O)

that are unique to that base. It just so happens that the ASCII representation of the bases offers two such bits in the 2^{nd} and 3^{rd} least-significant positions (Figure 6). We take advantage of this property by using bit manipulations to implement the same mapping as in the Baseline algorithm for up to 8 input bases in parallel. Better yet, the BMI2 instruction set in modern x86-64 CPUs contains the vectorized Assembly instruction Parallel Bits Extract (`PEXT`), which implements this parallel mapping in just a few cycles.

We replaced the four lines in the inner loop above with a parallelized mapping from 8 bases to 16 bits. Because of Big-Endianess and byte arrangement in memory, we actually need to flip the byte order in each 64-bit input word, and then again flip the order of the two encoded bytes, in order to preserve file-format compatibility with all the previous encoders. This vectorization does indeed unlock an order-of-magnitude improvement in maximum encoding throughput (Figure 7). Moreover, even when chunk sizes are too large to fit entirely in L3 cache (64 MiB), we can saturate the RAM bandwidth at a much lower thread count, requiring fewer CPU resources.

The decoder works similarly with the reciprocal parallel-deposit instruction (`PDEP`) and byte reordering, but requires an extra translation step, because the '`10`' bit sequence actually maps back to 'E', not to 'T'. Although this translation is also implemented with bit-parallel operations, it still slows down decoding compared to the baseline.

E. *Instruction-level parallelism*

The code using `PEXT` and `PDEP` is already fairly efficient: when running it single-threaded from cache, Linux's perf tool reports an instruction-level parallelism of ≈ 4.4 instructions-per-cycle (IPC). Both IPC and total instruction count can be improved further by removing the code that reorders bytes during encoding and decoding (Figure 8). Although the encoded file will no longer maintain the same base order as the raw input, the decoded file will still be identical to the raw input, using far fewer instructions. This small change boosts IPC to ≈ 6.5 and median single-threaded encoding throughput to 34,949 MBase/s (90.7% improvement over the ordered version). We also unrolled the main loop twice to obtain an additional 4% improvement in maximum throughput.

Because of the cost of the 'E'-to-'T' translation with PDEP, we chose to reimplement the unordered decoder with a LUT, this time mapping from 8 two-bit bases to 8 ASCII bases. This change yielded

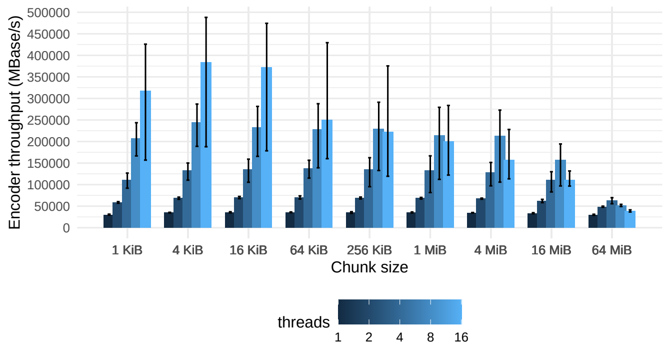

Fig. 8. Median encoder throughput with pext, loop unrolling and no reorder (100 runs; 95% CI; no I/O)

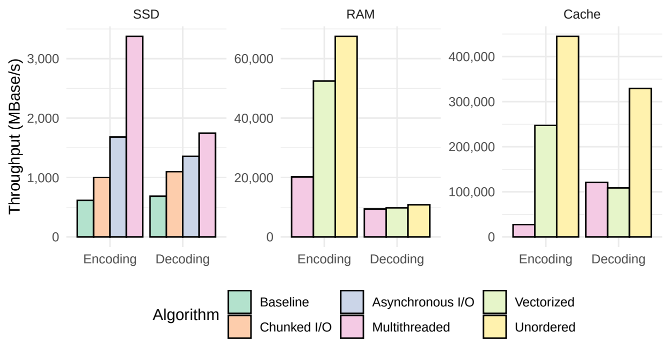

Fig. 9. Maximum throughput comparison (100 runs)

a significant performance boost in decoding, but at the cost of incorporating a larger 512 KiB LUT in the code, increasing the binary size from 34 KiB to 545 KiB.

III. Discussion

Figure 9 summarizes the best-case performance (not median) for all algorithms with three underlying storage media: SSD, RAM, and L3 CPU cache. The best encoder throughput approaches the CPU's L3 memory-copy bandwidth of 532 GB/s, as measured by AIDA 64. Moreover, the memory consumption of both the encoder and decoder is dominated by the configurable buffer sizes. Since we use modestly-sized buffers for improved performance (typically 128 KiB), even with 16 threads the encoder and decoder require less than 5 MiB of RAM. For comparison, even the SSD-based encoding is about 5 $\times$ faster than the fastest encoder recorded in the sequence compression benchmark [9], lzturbo-20-4t, using $\approx 0.25\%$ of its RAM.

These results demonstrate that direct encoding can be so efficient that it is feasible to encode or encode data on the fly even with modest resources, including on embedded devices like smartNICs and FPGAs. Direct-encoding can also be incorporated as part of other DNA storage or compression mechanisms such as reference-based and de-novo compression [3] with little additional overhead.

There remain several interesting questions for future work: Why does decoding throughput sometimes exceed encoding throughput and sometimes not? How is IPC affected by the storage media and by the number of threads? What is the effect of multithreading on performance variability? And, can we successfully use these algorithms in GPUs, with their high core count and memory bandwidth?

IV. Conclusion

With the exponential growth in genetic data, encoding and decoding DNA quickly grows critical to computational biology. Direct-coding offers the fastest alternative for DNA compression, yielding a modest but predictably constant compression ratio of 4:1. For direct-coding to take full advantage of the capabilities of ubiquitous modern hardware, we need to exploit parallelism on all levels: processes, threads, devices, vectors, bits, and instructions. This paper proposes an efficient implementation to unleash the full parallelism of modern CPUs, maximizing throughput to the underlying storage media's limits while minimizing computational resource use.

References

[1] D. Mansouri, X. Yuan, and A. Saidani, "A new lossless dna compression algorithm based on a single-block encoding scheme," *Algorithms*, vol. 13, no. 4, p. 99, 2020.

[2] R. Gilmary, A. Venkatesan, and G. Vaiyapuri, "Compression techniques for dna sequences: A thematic review." *Journal of Compututer Science and Engineering*, vol. 15, no. 2, pp. 59–71, 6 2021.

[3] Z. Zhu, Y. Zhang, Z. Ji, S. He, and X. Yang, "High-throughput DNA sequence data compression," *Briefings in Bioinformatics*, vol. 16, no. 1, pp. 1–15, 12 2013.

[4] D. Pratas, M. Hosseini, J. M. Silva, and A. J. Pinho, "A reference-free lossless compression algorithm for dna sequences using a competitive prediction of two classes of weighted models," *Entropy*, vol. 21, no. 11, p. 1074, 11 2019.

[5] J. Ouyang, P. Feng, and J. Kang, "Fast compression of huge dna sequence data," in *Proceedings of the 5th International Conference on BioMedical Engineering and Informatics*. IEEE, 2012, pp. 885–888.

[6] A. J. Cox, M. J. Bauer, T. Jakobi, and G. Rosone, "Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform," *Bioinformatics*, vol. 28, no. 11, pp. 1415–1419, 05 2012.

[7] M. Hosseini, D. Pratas, and A. J. Pinho, "A survey on data compression methods for biological sequences," *Information*, vol. 7, no. 4, p. 56, 2016.

[8] R. Wang, Y. Bai, Y.-S. Chu, Z. Wang, Y. Wang, M. Sun, J. Li, T. Zang, and Y. Wang, "Deepdna: A hybrid convolutional and recurrent neural network for compressing human mitochondrial genomes," in *International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, Dec. 2018, pp. 270–274.

[9] K. Kryukov, M. T. Ueda, S. Nakagawa, and T. Imanishi, "Sequence compression benchmark (scb) database—a comprehensive evaluation of reference-free compressors for fasta-formatted sequences," *GigaScience*, vol. 9, no. 7, p. giaa072, 7 2020.

[10] R. Mitra and S. Roy, "A survey of genome compression methodology," *International Journal of Computer Science and Engineering*, vol. 6, pp. 983–991, 8 2018.

[11] R. Challa, G. P. Devi, K. Arava, and K. SrinivasaRao, "A novel compression technique for dna sequence compaction," in *Proceedings of the International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES)*. IEEE, 10 2016, pp. 1351–1354.

[12] S. Du, J. Li, and N. Bian, "A compression method for dna," *Plos one*, vol. 15, no. 11, p. e0238220, Nov. 2020.

[13] A. Gupta, V. Rishiwal, and S. Agarwal, "Efficient storage of massive biological sequences in compact form," in *Contemporary Computing: Third International Conference*. Noida, India: Springer, 8 2010, pp. 13–22.

[14] B. Saada and J. Zhang, "Dna sequences compression algorithm based on extended-ascii representation," in *Proceedings of the world congress on engineering and computer science*, San Francisco, CA, 2 2015.

[15] D. Satyanvesh, K. Balleda, A. Padyana, and P. Baruah, "Gencodex-a novel algorithm for compressing dna sequences on multi-cores and gpus," in *Proceedings of the 19th International Conference on High Performance Computing (HiPC)*. Pune, India: IEEE, 2012.

[16] A. Mehta and B. Patel, "Dna compression using hash based data structure," *International Journal of Information Technology & Knowledge Management*, vol. 3, pp. 383–386, 2010.

[17] S. Goel *et al.*, "A compression algorithm for dna that uses ascii values," in *IEEE International Advance Computing Conference (IACC)*. IEEE, 2014, pp. 739–743.

[18] B. C. LaHaise, "An aio implementation and its behaviour," in *Ottawa Linux Symposium*, 2002, p. 260.