Future Generation Computer Systems 177 (2026) 108244

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A scalable distributed workflow for accelerating long reads self-correction

Riccardo Ceccaroni, Lorenzo Di Rocco *, Umberto Ferraro Petrillo, Pierpaolo Brutti

Sapienza University of Rome, Italy

ARTICLE INFO

Keywords:
Bioinformatics
Computational genomics
Distributed computing

ABSTRACT

Third-Generation Sequencing (TGS) technologies have transformed genomic research by enabling the extraction of longer nucleotide sequences (referred to as *long reads*) and providing deeper insights into genome structure. However, long reads are often associated with high sequencing error rates, making their correction a major challenge in many computational genomic pipelines. In this paper, we introduce `HyperC`, a distributed workflow designed to accelerate the execution of existing long-read self-correction tools through a hybrid parallelization strategy. By combining MPI and OpenMP, our proposal efficiently scatters and executes tasks across a distributed computing system. Optimized input data handling further reduces I/O bottlenecks and maximizes resource utilization. To assess the effectiveness of `HyperC`, we integrated it with `CONSENT`, a high-performance correction module, and conducted extensive experiments on real-world sequencing datasets. The results show significant reductions in execution time and improved scalability compared to the standalone execution of `CONSENT`, establishing `HyperC`as a robust and practical solution for high-performance genomic analysis and population-scale studies.

1. Introduction

Third-generation sequencing (TGS) technologies have fundamentally improved the study of genetic information, offering substantial improvements over second-generation methods [1,2].

A key advancement of TGS platforms is their ability to generate longer DNA fragments, known as *long reads*. These developments have significantly enhanced many tasks in computational genomics, particularly in gapless telomere-to-telomere assembly [3,4], offering new insights into genome structure and function. Long-read sequencing has substantially deepened our understanding of diploid [5–7] and more complex polyploid genomes [8]. Moreover, it has provided valuable insights into the structure and diversity of metagenomic samples, allowing for a more comprehensive analysis of microbial communities, complex ecosystems, and environmental samples [9,10].

In addition, cancer research has greatly benefited from TGS technologies, particularly through their ability to accurately detect structural variants and complex genomic rearrangements. These advancements contribute to the identification of novel oncogenic drivers, more precise diagnostics, and the development of personalized therapeutic strategies [11,12].

The development of novel computational methods has been crucial for integrating long reads into genomics workflows effectively, with algorithms specifically designed to fully harness their potential [13].

A major challenge with TGS platforms, such as those from Pacific Biosciences (PacBio) and Oxford Nanopore Technologies (ONT), is their relatively high sequencing error rate. Consequently, it is often necessary to supplement the analysis with additional data, as the more accurate *short reads* generated by next-generation sequencing (NGS) technologies [14]. However, relying on different sequencing technologies increases both the cost of computational genomic workflows and the volume of data to process.

As a potential solution, the *self-correction* algorithms employ specialized strategies to polish input long reads without the need for supplementary data. While these procedures are promising and effective in settings where short reads are unavailable, they come at a computational cost, often requiring substantial resources and long processing times.

Parallelizing these correction processes to scale efficiently with increasing data sizes therefore remains a critical challenge. Most self-correction tools are currently limited in using only the cores of a single machine [15]. In this context, distributed architectures offer a promising solution by enabling clusters of multiple machines (referred to as *computing nodes*) to coordinate and solve a single large-scale problem.

Although distributed systems have been successfully applied to accelerate certain tasks in computational genomics, they have yet to become a widely adopted paradigm in this field. This limited adoption is primarily due to the complexity involved in implementing distributed pipelines and leveraging high-performance computing (HPC) resources, especially

* Corresponding author.
E-mail address: lorenzo.dirocco@uniroma1.it (L. Di Rocco).

https://doi.org/10.1016/j.future.2025.108244
Received 3 May 2025; Received in revised form 17 October 2025; Accepted 4 November 2025
Available online 13 November 2025

for bioinformatics practitioners who may lack expertise in parallel and distributed systems. Moreover, using multiple computing nodes does not automatically guarantee improved performance. Indeed, if not managed efficiently, distributed systems can lead to suboptimal outcomes, sometimes even underperforming compared to well-optimized single-node solutions.

In this paper, we introduce HyperC, a distributed workflow designed for scaling existing long-read self-correction procedures across multiple computing nodes. The goal of HyperCis to make current correction methods practical for very large datasets and in reasonable amount of time, by enabling their execution in distributed computing environments, without requiring users to have prior knowledge of parallel or distributed programming.

HyperCspeeds up the correction process through a hybrid parallelization strategy, combining MPI and OpenMP. The former is used for inter-node communication while the latter facilitates parallel execution within each node, optimizing the correction of long reads. Additionally, HyperCintroduces optimized input file management, improving how data are read, partitioned, and assigned as tasks within the distributed architecture.

We validate the effectiveness of our proposal by integrating it with CONSENT, a high-quality error correction tool, and conducting experiments on real-world datasets. The results demonstrate that HyperCsignificantly enhances the performance of CONSENT, improving both runtime and scalability.

2. Biological preliminaries

2.1. Genomic sequencing

DNA governs the essential biological processes occurring within the cells of living organisms. The DNA molecule follows a double-stranded, antiparallel structure, where each strand is composed of linear sequences of nucleotides. These nucleotides are the building blocks of DNA, consisting of four types of bases: adenine (A), cytosine (C), guanine (G), and thymine (T). The two strands run in opposite directions and are held together by complementary base pairing, in which adenine pairs with thymine, and cytosine pairs with guanine. This pairing rule, along with chemical interactions, defines the double-helix shape of DNA.

Starting from one strand, its reverse complement can be derived by following the nucleotide complementarity rules. Consequently, DNA sequences can be encoded as strings drawn from the alphabet $\Sigma = \{A, C, G, T\}$, where each character corresponds to a nucleotide base.

Currently, sequencing technologies are still limited in the length of nucleotide fragments (called *reads*) they can extract, preventing the genome from being read as a single continuous sequence.

As a result, the sequencing process generates reads randomly from either strand of a chromosome, leading to the loss of positional information relative to the original DNA fragment. This results in a fragmented collection of nucleotide sequences to be processed.

Computational genomics pipelines are essential for reconstructing and analyzing the genome from this chaotic and redundant collection of reads. Genome assembly involves piecing together these reads by analyzing their overlaps.

However, DNA contains numerous repetitive regions, making it difficult to accurately align overlapping reads and resulting in ambiguities during genome assembly.

Recent advancements in TGS technologies have enabled the extraction of increasingly longer reads, addressing several challenges by providing more contiguous sequence information. This improvement enhances the genome assembly process and deepens our understanding of the genome.

2.2. Methods for long reads correction

Despite the advantages of TGS platforms, their output is typically affected by high error rates, which pose a significant obstacle to genome assembly.

Long-reads correction methods aim to identify and filter out sequencing errors arising in TGS data. We discern two different possible approaches: *hybrid correction* and *self-correction*.

Hybrid correction methods exploit accessory sets of short reads sequenced from the same sample, and leverage their higher accuracy to detect errors and polish TGS samples [16–18]. Conversely, self-correction methods rely exclusively on the information contained in the input long reads [19–21].

The effectiveness of either approach depends on several factors, including the *coverage level*.The coverage level refers to the number of times a given region of the genome is sequenced. Hybrid correction performs well even for low-coverage samples, provided that the short-read depth is sufficient. Self-correction, on the other hand, requires higher coverage to generate enough overlapping long reads for effective error detection and correction.

Hybrid approaches, while promising, are not without drawbacks and are constrained by the limitations of second-generation technologies. First, the assignment of short reads to long reads can be ambiguous, particularly when correcting sequences that span DNA regions difficult to resolve with short reads. As a result, the correction procedure may become biased when processing sequencing data from repetitive regions, causing a degradation of the output quality.

Additionally, long reads from certain regions of the genome may be undercovered in second-generation samples due to specific sequence content, such as GC-rich regions. This lack of sufficient coverage hinders error correction in these areas. Finally, employing multiple sequencing protocols increases the cost of analysis, potentially restricting the application of hybrid correction methods.

Conversely, self-correction methods often exhibit more robust performance for complex genomic regions and do not involve multiple sequencing platforms. Moreover, as TGS technologies evolve and error rates decrease, self-correction continues to improve. Although correction is still necessary, recent developments have made self-correction effective with reasonable coverage levels.

2.3. Long reads self-correction

Self-correction algorithms analyze overlapping sequences to identify those likely originating from the same genomic region, enabling comparisons that minimize biases introduced during sequencing. These algorithms fall into several categories, with multiple sequence alignment (MSA) methods among the most widely used.

MSA-based methods have proven highly effective for self-correction, typically addressing this task by identifying the consensus sequence through the alignment of reads derived from the same genomic region. This approach has shown to deliver higher output quality [22], making it a robust pre-processing step for most leading genome assemblers [23].

In practice, MSA-based methods identify reads with significant overlap by computing pairwise alignments. For each target sequence τ, the corresponding overlapping set S_τ consists of the reads that significantly overlap with τ. For each pair (τ, S_τ), a pileup table is constructed to store their MSA. The pileup table arranges the sequences to highlight regions of similarity and differences, enabling the generation of a consensus sequence that serves as the corrected version of the target read.

The specific algorithmic strategies used for MSA construction and consensus extraction can vary, and their performance is often influenced by the tool used to compute overlaps between input long reads and the framework adopted for consensus generation.

Prominent tools employing MSA-based self-correction include FLAS [24], MECAT [20], and Canu [19], each employing distinct strategies to derive consensus sequences from overlapping reads. Among these,

CONSENT is one of the most effective [25]. Like other MSA-based approaches, CONSENTrelies on an external mapping tool, typically Minimap [26], to evaluate pairwise alignments and identify overlaps between input long reads. Based on this mapping output, the corresponding set S_τ is computed for each target read τ. The overlapping regions between τ with elements in S_τ are then divided into smaller windows. Each window is independently aligned using multiple sequence alignment based on a partial order graph [27]. For each window, a consensus sequence is generated and further refined through a local de Bruijn graph-based polishing step to improve accuracy. Finally, the consensus results from all windows are merged to produce the corrected version of the target read τ.

Although several alternatives have been introduced, the CONSENT module continues to provide high-quality results and even outperforms more recent tools in certain scenarios [28]. The analysis of large datasets with CONSENTcan be accelerated thanks to the adoption of a multi-threaded architecture able to process multiple sequences in parallel. However, despite this, its practical applicability is limited by long execution times, making it less suitable for population-scale studies.

3. Genomics in the big data era

3.1. Challenges and opportunities in distributed computing

The increasing volume of data generated by sequencing technologies has led to an exponential growth of sequencing data, expanding our understanding of genomes but also posing significant challenges. Parallelization alone is gradually becoming insufficient to meet the computational demands of modern bioinformatics analysis [29]. In this context, distributed computing plays a pivotal role by enabling the efficient and scalable processing of large-scale data.

Distributed computing has already proven successful in various genomic applications. Notable examples include k-mer counting, nucleotide strings compression, and variant detection and genotyping [30–32]. However, genomic data structures, whether modeled as strings or graphs, presents critical aspects for distributed systems. DNA inherently exhibits long-range dependencies that require extensive information exchange between processes in a distributed system. This exchange can become a bottleneck due to the frequent communication between nodes. Consequently, distributed genomic data processing remains extremely challenging.

To address these challenges, alignment-free (AF) techniques have gained considerable attention within distributed genomic analysis [33, 34]. Unlike traditional approaches, AF methods bypass the sequential and positional constraints of DNA by focusing on statistical patterns such as k-mer frequencies, where k-mers are subsequences of length k that represent local nucleotide composition. This shift in perspective enables more flexible modeling of DNA within HPC architectures, transforming many genomics problems into counting or querying tasks over distributed data structures. Such problems are generally more tractable and benefit from well-established and efficient solutions in the distributed computing domain.

3.2. MPI and OpenMP for scalable genomic workflows

There are different paradigms and frameworks for implementing distributed algorithms, ranging from high-level to low-level solutions.

High-level approaches [35,36] are easier to implement and typically offer more straightforward solutions. In contrast, low-level approaches offer finer control over task distribution and communication but come with greater complexity and higher implementation costs.

Given the complexity of genomic analysis, especially in large-scale tasks, low-level distributed computing techniques may sometimes be necessary to fully exploit the potential of distributed systems [37–39]. One of the most effective solutions in this domain is the use of hybrid approaches that combine MPI (Message Passing Interface) and OpenMP (Open Multi-Processing).

MPI [40] is a widely used standard for distributed memory architectures, providing explicit communication between processes running on different nodes. In MPI, each process is assigned a rank, which serves as a unique identifier within the system. The rank allows each process to communicate efficiently with others by specifying which process to send or receive data from.

OpenMP [41], on the other hand, is designed for shared memory architectures, where multiple threads run on a single machine. In OpenMP, parallelization is managed at the thread level, and each thread is assigned an ID (a concept similar to rank in MPI, but specific to threads within a single node). Each thread works independently on a portion of the data, and the OpenMP compiler directives manage the distribution of tasks among threads. In this model, threads share the same memory space, making communication between them easier than in MPI. OpenMP is particularly useful for exploiting the multiple cores available within a single machine.

Hybrid approaches combining MPI and OpenMP leverage the strengths of both paradigms. MPI manages communication across multiple nodes, while OpenMP optimizes parallelism within individual nodes. This combination provides flexibility, enabling the efficient execution of computational genomics tasks across large machine clusters.

4. Materials and methods

To the best of our knowledge, no existing tool addresses the challenge of distributing long-read self-correction workflows across multiple computing nodes. In this work, we present HyperC, a hybrid parallelization workflow combining MPI and OpenMP to fully exploit inter-node and intra-node parallelism. Our approach optimizes input data loading and distribution to ensure that the correction process efficiently leverages all available computing nodes and cores.

The following section provides a detailed step-by-step explanation of HyperCfor enhanced scalability of MSA-based correction procedures.

4.1. MSA-based workflow for long reads self-correction

MSA-based self-correction procedures for long reads require two types of input: the raw sequencing reads and information about their alignment. Long reads are usually available through FASTQfiles, a standard file format widely used in bioinformatics for storing sequencing data. Each record in a FASTQfile includes a header, the nucleotide sequence, and quality scores. These reads are stored as plain text strings, with one byte per character being a common representation. Since the length of long reads is highly variable, the size of each read in bytes also varies accordingly.

The alignments are typically represented in a PAF(Pairwise mApping Format) file, i.e., a compact, tab-delimited format in which each row represent an alignment between a pair of input reads, and contains different fields, including :

- *Query Name*: Name of the read being aligned.
- *Query Start*: Starting position of the alignment on the query sequence.
- *Query End*: Ending position of the alignment on the query sequence.
- *Strand*: Indicates whether the alignment is on the forward (+) or reverse (-) strand.
- *Target Name*: Name of the reference or another read to which the query is aligned.
- *Target Start*: Starting position of the alignment on the target sequence.
- *Target End*: Ending position of the alignment on the target sequence.

To correct a target read τ using the MSA approach, the PAFfile is first consulted to identify the query reads that align significantly with τ. The corresponding overlapping set S_τ is then retrieved from the FASTQfile.

Once detected S_τ, the MSA is computed and processed according to the chosen correction module (see Section 2.3). This procedure is inherently scalable with respect to the number of reads, as it involves generating independent tasks for each target read and its set of overlaps. However, in a distributed environment, efficiently reading and distributing data from the PAFand FASTQfiles across computing nodes poses a significant challenge that can lead to performance bottlenecks.

4.2. Challenges in scaling correction pipelines across multiple computing nodes

One of the primary challenges in distributing MSA-based correction pipelines across multiple computing nodes arises from the nature of sequencing data itself. Due to the inherent redundancy in long-read sequencing, the same read may belong to multiple overlapping sets of different target reads. As a result, a single read might need to be loaded multiple times, increasing the overall I/O workload.

A naive approach would assign both input reading and task scheduling operations to a single process. In this strategy, a single process first reads the contents of the input FASTQand PAFfiles, then identifies the target reads along with their overlapping sets, and finally generates error correction tasks to be distributed across the worker processes in the computing cluster. We refer to this approach as the **centralized reader (CR)** strategy.

We experimented with this approach in a preliminary version of our workflow [42]. Although the CR strategy ensures a structured and centralized data flow, it introduces a significant drawback. The single process must both read the sequencing data and act as a manager, orchestrating correction task distribution. This dual role introduces a potential scalability issue, as the process quickly becomes burdened by the increasing number of jobs, slowing down the overall performance of the system. Furthermore, job distribution over a large-scale computing cluster is likely to overload the underlying network, thus introducing delays that degrade performance.

A more effective alternative is to use multiple distributed processes to read and process the input files concurrently. We refer to this approach as the **parallelized reader (PR)** strategy. However, it is important to note that parallel reading does not always yield performance improvements. A major limitation stems from the need to access the FASTQfiles multiple times in a sparse, non-sequential order, which can hinder the potential benefits of parallelization. In addition, when working on HPC infrastructures, input files are often stored on shared network file systems. With the PR strategy, processes on different nodes must compete to access the same physical file, creating contention on the storage subsystem. This contention can cause significant bottlenecks, especially for very large files, further reducing the performance of the system.

4.3. The HyperC workflow

The workflow behind HyperCis designed to load input data efficiently and distribute MSA-based correction tasks for each read across multiple parallel jobs, ensuring optimal task generation and allocation based on input FASTQand a PAFfiles.

HyperCemploys a hybrid parallel programming model based on MPI and OpenMP to achieve scalability and computational efficiency. MPI manages communication between different computing nodes, with each node represented as a distinct MPI rank. Each MPI rank runs as an independent process with its own private memory space. Communication between ranks is explicit and message-based. Within each node, OpenMP coordinates parallel execution across multiple CPU cores using a shared-memory approach. A single MPI process spawns multiple threads that operate within the same address space, allowing for fast data access without the need for inter-thread message passing.

The design of HyperCdirectly addresses the core limitations of both CR and PR strategies. Unlike the CR approach, HyperCavoids the bottleneck of a single coordinating process by scattering input reading, processing, and task generation across multiple MPI ranks.

Compared to the PR strategy, HyperCmitigates the inefficiencies from concurrent access to shared input files. Instead of allowing multiple processes to repeatedly access the same data from disk, HyperCloads a compressed version of files containing multiple read hits directly into memory. This reduces I/O contention and minimizes redundant disk access. Additionally, leveraging OpenMP within each node ensures efficient use of local computational resources through shared-memory parallelism. We also remark that our approach facilitates the integration of non-distributed error correction modules like, e.g., the one provided by the CONSENTframework (see Section 2.3), allowing their parallel execution across different computing nodes, while transparently taking care of all the underlying communication infrastructure. In the following, we describe in detail each step of our workflow, while a summary is presented in Fig. 1.

4.3.1. Loading the FASTQfile

Initially, each MPI rank employs a dedicated thread to read the FASTQfile into an associative data structure, with sequence headers as keys and corresponding nucleotide sequences as values (see box A of Fig. 1). Loading the FASTQ file into the memory of each computing node speeds up random access to the data since access is direct, minimizing disk I/O operations. However, in a distributed architecture, memory usage across nodes remains a concern [43]. Given the potentially large size of FASTQ files, loading the entire file into memory on each node is often impractical. Duplicating hundreds of gigabytes of data on each node is also inefficient. For this reason, HyperCemploys the Zstandard (Zstd) algorithm[1] to compress data structure storing the input reads. Zstd reduces memory usage significantly while allowing fast decompression, ensuring efficient sequence retrieval during processing.

Sequence compression minimizes overall memory requirements and allows large FASTQfiles to be handled more effectively in the memory of each computing node. Unlike traditional compression algorithms, Zstd achieves high compression ratios with rapid decompression, crucial for real-time sequence retrieval.

After loading, the sequences are stored in shared data structures within each computing node, enabling quick lookups by the concurrent OpenMP threads. The headers are also compressed to further optimize the storage space.

Therefore, in our design, each MPI rank loads a compressed copy of the FASTQ dataset into local memory. Although this choice results in dataset replication across nodes, it was intentionally adopted to avoid concurrent access to shared storage, which typically leads to severe I/O contention in HPC environments. The combination of compression via the Zstd algorithm and shared-memory access within nodes ensures that this replication remains memory-efficient while enabling low-latency data retrieval during correction.

4.3.2. Loading the PAFfile

A single thread in the first MPI rank counts the total number of lines in the PAFfile. It then splits the PAFfile lines among the available MPI ranks (see box B of Fig. 1). Specifically, the partitioning of the PAFfile ensures that all lines referring to the same read are processed within the same MPI rank. Within each rank, OpenMP is utilized to process multiple target reads in parallel.

Each MPI rank has one thread that reads its assigned portion of the PAFfile and generates a queue of jobs (see box C of Fig. 1). The remaining threads within each node fetch jobs from the queue (see box D of Fig. 1), extract the corresponding sequence, and gather the overlapping set from the shared compressed representation of the FASTQfile. Sequences are

[1] Zstd is a fast and efficient data compression algorithm developed by Facebook, available at https://facebook.github.io/zstd/

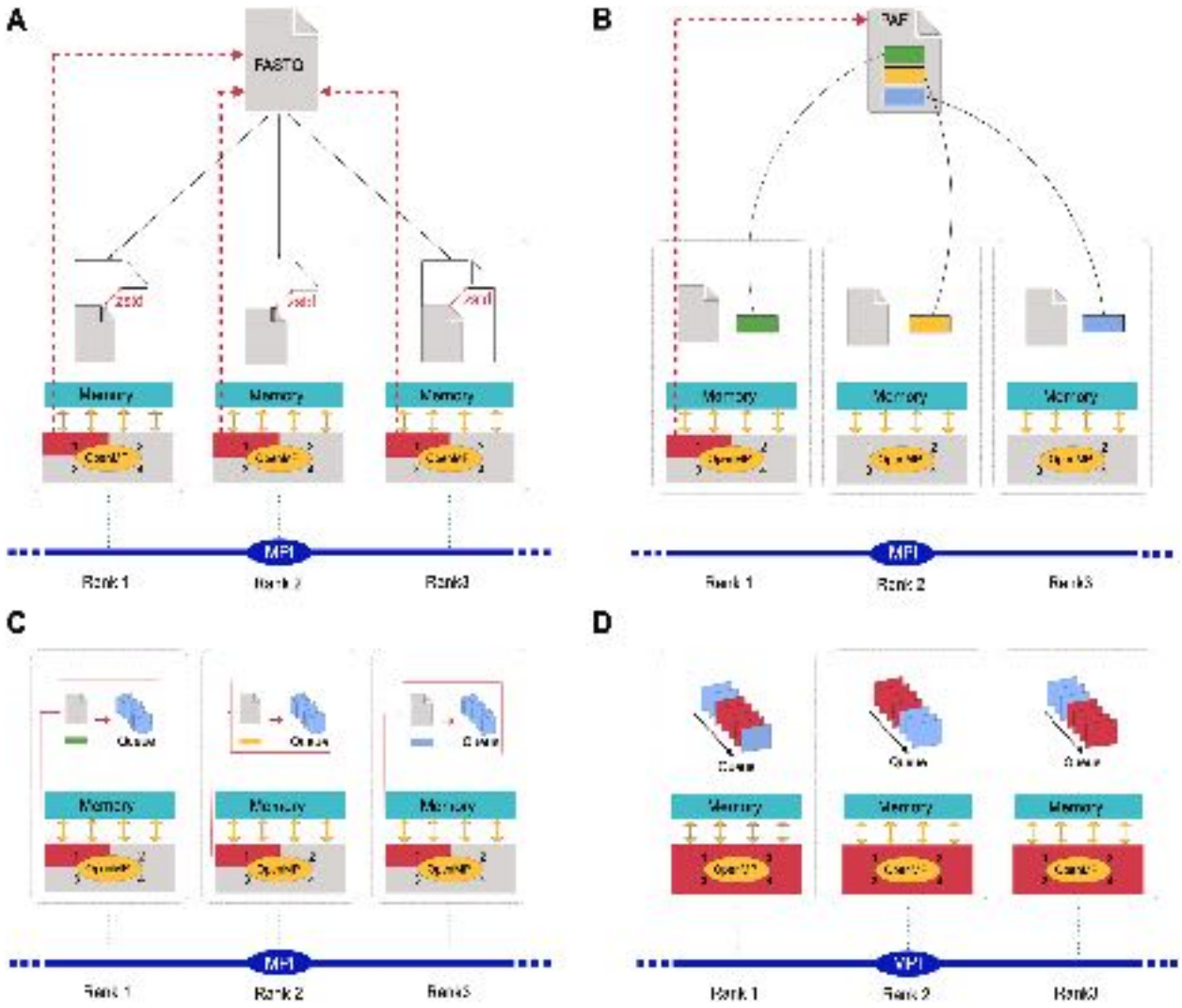


Fig. 1. Overview of the HyperCworkflow. HyperCleverages a hybrid parallelization model combining multiple MPI ranks across computing nodes with OpenMP threads operating in shared memory within each rank. In the illustration, active OpenMP threads involved in each phase of the workflow are highlighted in red. **(A)** Each MPI rank loads the input FASTQ file into memory and compresses it using the Zstandard algorithm. **(B)** A single OpenMP thread from rank 1 partitions the PAF file across MPI ranks, ensuring that all alignments associated with a given target read are handled by the same rank. **(C)** Each rank constructs a local queue of correction tasks based on its assigned portion of the PAF file. **(D)** OpenMP threads within each rank concurrently retrieve jobs from the queue, decompress the required sequences, and execute the correction module.

decompressed from the FASTQmap using Zstd before processing, minimizing memory usage and maintaining fast access. The correction module is applied, and corrected sequences are written to separate output files per MPI rank to avoid I/O bottlenecks.

The job queue is managed dynamically, allowing threads to efficiently fetch and process available tasks.

4.4. Integrating custom correction modules

HyperC is designed to be a highly extensible solution for any MSA-based self-correction module. The only requirement for integration is that the correction module must be able to process a single target read and its corresponding overlapping set, through a C/C++ based interface.

We remark that the internal logic of the correction module does not need to be modified in order to support parallel or distributed execution, as this part is automatically and transparently managed by HyperC. Thus, thanks to this modular design, researchers and practitioners with limited expertise in HPC systems, can enjoy the advantages of a parallel execution of their correction modules with minimal effort.

Once the correction module is implemented, users need to follow these steps:[2]

- **Define the correction module interface.** The correction module should implement a simple interface exposing one function that a target read and its corresponding overlapping reads as input. The function should return the corrected target read (see Listing 1). This modular structure ensures that the correction process is decoupled from the data distribution and parallelization handled by HyperC.

```
1 /**
2  * Custom correction function to be
      integrated into HyperC.
3  *
4  * @param query     - The target read
      sequence to be corrected
```

[2] For more details, see the instructions available at https://github.com/riccardoc95/hipesc

```
 * @param overlaps - A list of overlapping reads with alignment information
 * @return        - The corrected target read
 */
std::string correction(const std::string& query, const std::vector<Overlap>& overlaps) {
    // Implement custom correction logic here
    // For example, prepare input for an external tool like CONSENT,
    // execute it, and return the corrected sequence.

    return corrected\_sequence;
}
```

Listing 1
Definition of a custom correction function.

- **Modify the HyperC interface.** Users will need to modify a small section of code to link the correction module's functionality. Since HyperCis written in C++, this step is usually limited to providing a wrapper function that invokes the correction process (see Listing 2). No significant changes are required in the parallelization logic of HyperC, as it already manages the distribution of reads and overlaps.
- **Compile and build.** Once the correction module is integrated, users simply need to compile the code with a C++ compiler, ensuring that all dependencies are linked correctly.
- **Run the pipeline.** After integration and compilation, the user can execute the correction process on their dataset using HyperC to accelerate the procedure on an HPC cluster. By leveraging a job scheduling system such as SLURM, users can submit their correction jobs through the terminal, specifying the number of nodes and computational resources they wish to allocate. HyperC will then efficiently distribute the correction tasks across the selected nodes, optimizing performance and ensuring scalable execution.

```
if (!jobs.empty()) {
    Job job = jobs.back();
    jobs.pop_back();

    // Decompress the query read
    auto [q_data, q_len] = fastq_dict.at(job.query_name);
    std::string query = decompress_zstd(q_data, q_len);

    // Build the vector of overlapping reads
    std::vector<Overlap> overlaps;
    for (const auto& t : job.jobs) {
        auto [t_data, t_len] = fastq_dict.at(t.target_name);
        std::string target = decompress_zstd(t_data, t_len);
        overlaps.push_back({target, t.query_start, t.query_end, t.target_start, t.target_end, t.strand});
    }

    // Call user-defined correction module
    std::string query_correction = correction(query, overlaps);

    // Write the output to file
    std::string query_name = decompress_zstd(job.query_name, 128);
    output_file << "@" << query_name << "\n" << query_correction << std::endl;
}
```

Listing 2
Calling the correction function within a HyperC worker thread.

4.5. Failure recovery

In a distributed computing environment, failures like node crashes or process terminations can significantly affect the execution of parallel workloads. HyperC is built to handle such failures, ensuring minimal data loss and enabling recovery without restarting the entire workflow.

One of the primary failure scenarios is an MPI rank failure, occurring when a node becomes unresponsive or crashes. To address this,HyperCimplements a Dynamic Work Redistribution strategy, letting remaining ranks to take over the workload of the failed rank. This strategy ensures continued execution with minimal disruption. When a failure is detected, the system follows these steps:

1. Rank 0 monitors MPI ranks and detects failures;
2. The jobs of the failed rank are redistributed among active ranks;
3. The affected tasks are dynamically re-queued for other nodes to process.

This approach provides fault tolerance without complex checkpointing mechanisms. HyperCalso tolerates coordinator node failures without compromising the workflow execution. To prevent a single point of failure, HyperCintegrates with fault-tolerant MPI runtimes such as ULFM and Reinit++ [44,45], which provide built-in support for communicator recovery and process respawning. By leveraging these runtime capabilities, HyperCreconstructs communicators and reassigns coordinator responsibilities if the coordinator fails, maintaining execution without a full restart.

5. Experiments

We conducted an experimental analysis to assess the performance gains achieved by HyperCwhen correcting input long reads samples across multiple computing nodes. The primary goal of our experiments was to show that using HyperCwithin a distributed computing architecture can significantly reduce the runtime of self-correction procedures in a scalable way, thus allowing for the processing of very-large datasets in a practical amount of time.

To this end, we focused on CONSENT(see Section 2.3), an effective self-correction tool. Although CONSENTis able to leverage the multiple computing units available within a single machine thanks to its multi-threaded architecture, it is still unable to process very large datasets in a reasonable amount of time.

To overcome this limitation, we integrated its correction module with HyperC, showing that the use of multiple computing nodes enables CONSENTto complete processing of TGS samples in just a few hours.

Note that we do not evaluate the accuracy of HyperCrelative to CONSENTsince both use the same self-correction module and thus produce identical results. The only expected difference is in the total runtime of the correction procedure.

Table 1
Summary of the datasets used in our experiments. HUMAN_CHR1 includes reads mapped to chromosome 1, while HUMAN_ALL comprises reads from the entire human genome. Both subsets were derived from the NA12878 sample of the Nanopore WGS Consortium dataset. The number of reads reflects the total count of DNA fragments produced by the sequencing platform that are included in each dataset.

Dataset	Size (GB)	Number of reads
HUMAN_CHR1	14	1,075,867
HUMAN_ALL	164	12,892,860

5.1. Experimental setting

All our tests were performed on 8 computing nodes of an HPC architecture, each equipped with two sockets, 64 cores per socket, for a total of 128 physical cores and 256 logical cores. The processors used are AMD EPYC 7313P 16-Core Processors running at a clock speed of 3.0 GHz. Additionally, each node is provisioned with 1TB of RAM. The system uses OpenMPI 5.0.3 compiled with GCC 13.2.0, and the interconnect is based on InfiniBand using Mellanox MT4124 adapters with a data rate of 200 Gbps. In the remainder of this paper, we will refer to both physical and logical cores as computing units, unless otherwise specified.

In addition, we repeated all the experiments 5 times but reporting exclusively the worst-case scenario, that is, the maximum execution times.

5.2. Dataset and materials

For our experiments, we used sequencing data from the Nanopore WGS Consortium, a publicly available dataset that provides whole-genome sequencing reads generated using Oxford Nanopore Technologies. In particular, we considered the 29x coverage dataset corresponding to the human reference sample NA12878, a widely studied individual from the 1000 Genomes Project, frequently used as a benchmark for evaluating sequencing and genome assembly methods.

From this dataset, we generated two subsets for our analysis:

1. HUMAN_CHR1, containing only the sequencing reads corresponding to chromosome 1;
2. HUMAN_ALL, including all sequencing reads from the entire genome.

The characteristics of the two subsets are summarized in Table 1. This division allows us to evaluate the efficiency and the scalability of HyperCunder two different workload scenarios. For both datasets, we computed the corresponding PAFfiles using the Minimap tool [26].

5.3. Experiment 1: Intra-node scalability evaluation

The goal of this experiment is to evaluate the efficiency and the scalability of HyperCwhen integrated with CONSENT, with respect to the original error correction tool, in small-scale scenarios. By this term we refer to scenarios where the problem size does not require yet the adoption of a distributed infrastructure. This experiment is relevant as it allows us to measure the overhead introduced by HyperCwith respect to the native multi-threaded architecture available in CONSENT.

To perform this evaluation, we executed both tools on the HUMAN_CHR1 dataset using a single computing node and an increasing number of computing units.

As illustrated in Fig. 2, HyperCwas able to leverage intra-node parallelization through OpenMP, while CONSENTalso benefited from its built-in multi-threaded implementation, which efficiently scales across multiple CPU cores. In this scenario, as the number of computing units increases gradually from 16 to 64, HyperCachieves a scalability comparable to that of CONSENT, with only a slight performance overhead

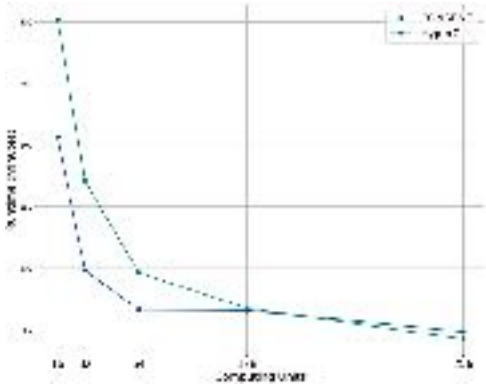


Fig. 2. Execution time required by HyperCto execute the CONSENTmodule on the HUMAN_CHR1 dataset, in terms of minutes, when considering a single computing node and an increasing number of computing units. The reported results include also the time required to load and process input FASTQ files.

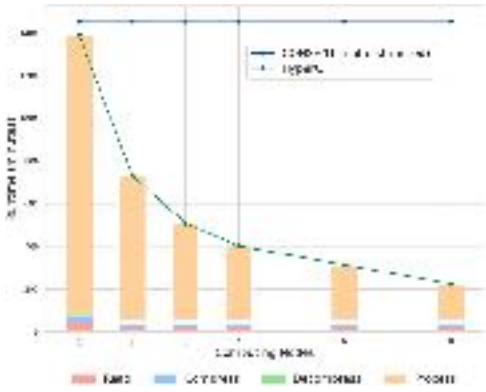


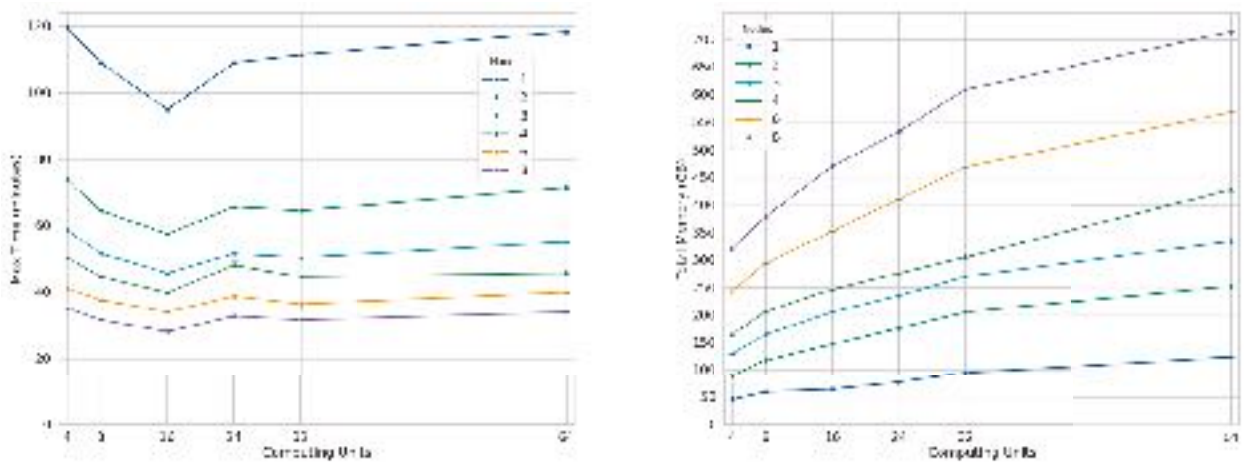
Fig. 3. Execution time required by HyperCto run the CONSENTcorrection module on the HUMAN_ALL dataset using an increasing number of computing nodes (each equipped with 128 threads), compared to the single-node execution time of CONSENT. Note that CONSENTis not a distributed approach and runs only on a single-node, hence its constant performance line across different node configurations.
The bars break down the total runtime of HyperCinto four components: reading the input, compressing the data, decompressing reads before correction, and executing the correction tasks.

due to the additional FASTQcompression step. Beyond 64 units, the scalability of CONSENTreaches a plateau, while the execution times of HyperCcontinues to decrease significantly, slightly outperforming CONSENTat 256 computing units and achieving a runtime gain of 12.3 %. This behavior suggests that HyperCis more effective at exploiting computing nodes equipped with a large number of computing units, where the higher level of parallelism mitigates the performance impact of the FASTQcompression.

5.4. Experiment 2: Inter-node scalability evaluation

The goal of this experiment is to evaluate the efficiency and the scalability of HyperCwhen integrated with CONSENT, in the context of large-scale datasets. This scenario is representative of scenarios where the usage of distributed computing infrastructures becomes essential to allows for practical execution times.

To perform this evaluation, we executed both tools on the HUMAN_ALL dataset. When processed using CONSENT, this dataset required one day of execution time while using the 128 computing units of a single computing node.



(a) (b)

Fig. 4. Execution time, in minutes, and peak memory usage, in GigaBytes, required by HyperCto execute the loading phase of the CONSENTmodule on the HUMAN_ALL dataset, as a function of the computing nodes and of the computing units. The reported results include also the time required to load and process input FASTQ files.

By integrating the correction module of CONSENTwith HyperC, we assessed the correction procedures by scaling with an increasing number of computing nodes, each one using 128 computing units.

As shown in Fig. 3, when processing large input datasets, HyperCachieves a runtime gain of 4.2 % compared to CONSENT, even when running on a single computing node and using the same number of computing units. This advantage arises from the need of CONSENTto index the FASTQfile and repeatedly access it from disk, a process that slows down the performance when working with large datasets. In contrast, HyperCmaintains a compressed version of the entire collection of reads, enabling faster and more efficient access. Increasing the number of computing nodes does not benefit CONSENT, as it is inherently limited to single-node execution. Conversely, HyperCdemonstrates a good scalability level and execution times decrease almost linearly with the number of nodes, reaching less than 4 h when running on 8 nodes, achieving a runtime gain of 84.6 % compared to the single-node solution provided by CONSENT.

Furthermore, Fig. 3 shows a breakdown of HyperC's runtime into four components: reading, compression, decompression, and processing. The process phase, which includes the correction steps, is the most computationally intensive but benefits substantially from distributed parallelism. The other components are relatively stable across all configurations, indicating that the I/O overhead is well contained. This confirms that the use of in-memory compressed structures, combined with task queues distributed across MPI ranks and served by OpenMP threads, is effective in reducing I/O contention and maximizing CPU utilization.

5.5. Experiment 3: Assessing read performance and memory usage

The goal of this experiment is to evaluate the efficiency of HyperCduring the input file loading phase. This phase is relevant as it includes the time to be spent for loading and parsing the FASTQand PAFfiles, plus the time spent preparing and distributing correction tasks over the distributed system.

Specifically, we assessed how the performance of HyperCscales as the number of computing nodes increases from 1 to 8, with each unit corresponding to a distinct MPI rank. For each configuration, we also varied the number of threads per rank, to analyze their impact on both execution time and memory usage, focusing specifically on the input loading phase.

Fig. 4a shows that increasing the number of computing nodes leads to a reduction in runtime, primarily because the PAFfile is partitioned across ranks, enabling more efficient parallel processing. However, we also observe that increasing the number of threads per rank results in a more contrasted behavior, when the number of threads becomes relatively large.

Initially, the runtime decreases because of the increased parallelism. However, over a certain threshold (16 computing units, in our experiment), the performance starts to get worse.

The observed plateau in execution times beyond 16 computing nodes requires clarification. This behavior occurs because our measurements in this experiment focus exclusively on the loading phase without including the actual correction processing function.

During the loading phase, the primary bottleneck is the partitioning and distribution of the PAF file across MPI ranks. With 16 nodes, the system achieves an optimal balance between parallelization benefits and coordination overhead. However, beyond 16 nodes, the loading time increases because additional nodes remain idle, waiting for correction tasks that are not executed in this isolated measurement.

This phenomenon is specific when measuring only the loading phase and it does not represent the behavior during full pipeline execution. When the correction module is integrated, the correction processing time significantly exceeds the loading time (see Fig. 2 and in Fig. 3), ensuring that the task queue never remains empty and all nodes are effectively utilized. The correction phase is CPU-intensive and provides sufficient work to maintain optimal resource utilization across all available nodes, eliminating the observed loading-phase bottleneck.

From a memory perspective, as shown in Fig. 4b, usage increases with the number of nodes, since the FASTQfile is replicated once per node. In our design, each MPI rank corresponds to a distinct computing node, and the file is loaded independently by each rank. Within a node, threads spawned by OpenMP share memory and thus avoid further replication. This strategy simplifies the implementation and avoids synchronization overhead across nodes. In addition, memory consumption per thread never exceeds 1 GB, ensuring that there is sufficient space for read processing without exceeding the available memory per thread.

6. Conclusions

In this work, we propose HyperC, a workflow designed to distribute the workload of a long-read self-correction pipeline across a distributed memory architecture. Our experimental evaluation on real-world datasets demonstrates that HyperCis a promising tool for processing data generated by TGS technologies. These experiments not only highlight the potential of our approach but also open up several promising directions for future research.

First, it would be interesting to apply this workflow to population-scale studies, enabling the correction of sequencing data from multiple individuals.

Additionally, we are working on integrating various correction modules to enhance flexibility and performance. We are also developing functionality to perform pairwise alignment directly within the distributed system, eliminating the need to rely on external tools such as Minimap for overlap computation.

Finally, an exciting line of research involves extending `HyperCbeyond` correction to also support assembly, particularly in metagenomic contexts. These scenarios involve samples from multiple organisms, where both correction and assembly require not only overlap detection but also the ability to identify reads originating from the same organism.

7. Funding

This research is partially supported by the INdAM - GNCS Project 2025 (CUP: E53C24001950001), and by the Università di Roma "La Sapienza" Research Project 2023, "Advancing Precision Public Health Medicine: High-Performance Computing for Efficient Solutions to Bioinformatics and Biostatistical Challenges."

Additional support is provided by the STILES project ("Strengthening the Italian Leadership in ELT and SKA"), funded bythe European Union and approved by the Italian Ministry of University and Research (MUR) under Ministerial Decree No. 3264 of December 28, 2021. This project is part of the National Recovery and Resilience Plan (NRRP) - NextGenerationEU, Mission 4 "Education and Research", Component 2 "From Research to Business", Investment 3.1 "Fund for the Realisation of an Integrated System of Research and Innovation Infrastructures."

CRediT authorship contribution statement

Riccardo Ceccaroni: Writing – original draft, Software, Methodology, Conceptualization; **Lorenzo Di Rocco:** Writing – original draft, Software, Methodology, Conceptualization; **Umberto Ferraro Petrillo:** Writing – review & editing, Supervision, Funding acquisition; **Pierpaolo Brutti:** Supervision.

Data availability

Data and code are available at the linked Github repository

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] S.L. Amarasinghe, S. Su, X. Dong, L. Zappia, M.E. Ritchie, Q. Gouil, Opportunities and challenges in long-read sequencing data analysis, Genome Biol. 21 (1) (2020) 30.
[2] E.L. Van Dijk, Y. Jaszczyszyn, D. Naquin, C. Thermes, The third revolution in sequencing technology, Trends Genetics 34 (9) (2018) 666–681.
[3] H. Li, R. Durbin, Genome assembly in the telomere-to-telomere era, Nat. Rev. Genet. 25 (9) (2024) 658–670.
[4] K.H. Miga, S. Koren, A. Rhie, M.R. Vollger, A. Gershman, A. Bzikadze, S. Brooks, E. Howe, D. Porubsky, G.A. Logsdon, et al., Telomere-to-telomere assembly of a complete human X chromosome, Nature 585 (7823) (2020) 79–84.
[5] D.J. Giguere, A.T. Bahcheli, S.S. Slattery, R.R. Patel, T.S. Browne, M. Flatley, B.J. Karas, D.R. Edgell, G.B. Gloor, Telomere-to-telomere genome assembly of phaeodactylum tricornutum, PeerJ 10 (2022) e13607.
[6] M. Rautiainen, S. Nurk, B.P. Walenz, G.A. Logsdon, D. Porubsky, A. Rhie, E.E. Eichler, A.M. Phillippy, S. Koren, Telomere-to-telomere assembly of diploid chromosomes with Verkko, Nat. Biotechnol. 41 (10) (2023) 1474–1482.
[7] E. Volpe, L. Corda, E. Di Tommaso, F. Pelliccia, R. Ottalevi, D. Licastro, A. Guarracino, M. Capulli, G. Formenti, E. Tassone, et al., The complete diploid reference genome of RPE-1 identifies human phased epigenetic landscapes, bioRxiv (2023).
[8] Y. Wang, J. Yu, M. Jiang, W. Lei, X. Zhang, H. Tang, Sequencing and assembly of polyploid genomes, Polyploidy Methods Protoc. (2023) 429–458.
[9] X. Feng, H. Cheng, D. Portik, H. Li, Metagenome assembly of high-fidelity long reads with hifiasm-meta,19 Nat. Methods 19 (6) (2022) 671–674.
[10] M. Kolmogorov, D.M. Bickhart, B. Behsaz, A. Gurevich, M. Rayko, S.B. Shin, K. Kuhn, J. Yuan, E. Polevikov, T.P.L. Smith, et al., MetaFlye: scalable long-read metagenome assembly using repeat graphs, Nat. Methods 17 (11) (2020) 1103–1110.
[11] L. Ermini, P. Driguez, The application of long-read sequencing to cancer, Cancers 16 (7) (2024) 1275.
[12] Y. Sakamoto, S. Sereewattanawoot, A. Suzuki, A new era of long-read sequencing for cancer genomics, J. Hum. Genet. 65 (1) (2020) 3–10.
[13] M.O. Pollard, D. Gurdasani, A.J. Mentzer, T. Porter, M.S. Sandhu, Long reads: their purpose and place, Hum. Mol. Genet. 27 (R2) (2018) R234–R241.
[14] Y. Ono, M. Hamada, K. Asai, PBSIM3: a simulator for all types of PacBio and ONT long reads, NAR Genomics Bioinf. 4 (4) (2022) lqac092.
[15] H. Zhang, C. Jain, S. Aluru, A comprehensive evaluation of long read error correction methods, BMC Genomics 21 (2020) 1–15.
[16] O. Choudhury, A. Chakrabarty, S.J. Emrich, HECIL: a hybrid error correction algorithm for long reads with iterative learning, Sci. Rep. 8 (1) (2018) 9936.
[17] L. Salmela, E. Rivals, LoRDEC: accurate and efficient long read error correction, Bioinformatics 30 (24) (2014) 3506–3514.
[18] C. Firtina, Z. Bar-Joseph, C. Alkan, A.E. Cicek, Hercules: a profile HMM-based hybrid error correction algorithm for long reads, Nucleic Acids Res. 46 (21) (2018) e125.
[19] S. Koren, B.P. Walenz, K. Berlin, J.R. Miller, N.H. Bergman, A.M. Phillippy, Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation, Genome Res. 27 (5) (2017) 722–736.
[20] C.-L. Xiao, Y. Chen, S.-Q. Xie, K.-N. Chen, Y. Wang, Y. Han, F. Luo, Z. Xie, MECAT: fast mapping, error correction, and de novo assembly for single-molecule sequencing reads, Nat. Methods 14 (11) (2017) 1072–1074.
[21] G. Tischler, E.W. Myers, Non hybrid long read consensus using local de Bruijn graph assembly, BioRxiv (2017) 106252.
[22] P. Morisse, T. Lecroq, A. Lefebvre, Long-read error correction: a survey and qualitative comparison, BioRxiv (2020) 2020–03.
[23] H. Cheng, G.T. Concepcion, X. Feng, H. Zhang, H. Li, Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm, Nat. Methods 18 (2) (2021) 170–175.
[24] E. Bao, F. Xie, C. Song, D. Song, FLAS: fast and high-throughput algorithm for PacBio long-read self-correction, Bioinformatics 35 (20) (2019) 3953–3960.
[25] P. Morisse, C. Marchet, A. Limasset, T. Lecroq, A. Lefebvre, Scalable long read self-correction and assembly polishing with multiple sequence alignment, Sci. Rep. 11 (1) (2021) 761.
[26] H. Li, MiniMap2: pairwise alignment for nucleotide sequences, Bioinformatics 34 (18) (2018) 3094–3100.
[27] C. Lee, C. Grasso, M.F. Sharlow, Multiple sequence alignment using partial order graphs, Bioinformatics 18 (3) (2002) 452–464.
[28] J. Hu, Z. Wang, Z. Sun, B. Hu, A.O. Ayoola, F. Liang, J. Li, J.R. Sandoval, D.N. Cooper, K. Ye, et al., NextDenovo: an efficient error correction and accurate assembly tool for noisy long reads, Genome Biol. 25 (1) (2024) 107.
[29] L. Di Rocco, U. Ferraro Petrillo, S.E. Rombo, DIAMIN: a software library for the distributed analysis of large-scale molecular interaction networks, BMC Bioinf. 23 (1) (2022) 474.
[30] I. Nisa, P. Pandey, M. Ellis, L. Oliker, A. Buluç, K. Yelick, Distributed-memory k-mer counting on GPUs, in: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2021, pp. 527–536.
[31] A.I. Maarala, O. Arasalo, D. Valenzuela, V. Mäkinen, K. Heljanko, Distributed hybrid-indexing of compressed pan-genomes for scalable and fast sequence alignment, PLoS ONE 16 (8) (2021) e0255260.
[32] P. Rao, A. Zachariah, D. Rao, P. Tonellato, W. Warren, E. Simoes, Accelerating variant calling on human genomes using a commodity cluster, in: Proceedings of the 30th ACM International Conference on Information & Knowledge Management, 2021, pp. 3388–3392.
[33] L. Amorosi, L.D. Rocco, U.F. Petrillo, Scheduling K-mers counting in a distributed environment, in: Optimization in Artificial Intelligence and Data Sciences: ODS, First Hybrid Conference, Rome, Italy, September 14–17, 2021, Springer, 2022, pp. 73–83.
[34] L. Di Rocco, U.F. Petrillo, Efficient and scalable alignment-free distributed genotyping of SNPs and short indels, IEEE Trans. Comput. Biol. Bioinf. (2025) 1288–1298.
[35] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M.I. Jordan, et al., Ray: a distributed framework for emerging {AI} applications, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 561–577.
[36] S. Salloum, R. Dautov, X. Chen, P.X. Peng, J.Z. Huang, Big data analytics on apache spark, Int. J. Data Sci. Anal. 1 (3) (2016) 145–164.
[37] J.M. Abuín, N. Lopes, L. Ferreira, T.F. Pena, B. Schmidt, Big data in metagenomics: apache spark vs MPI, PLoS ONE 15 (10) (2020) e0239741.
[38] Y. Li, G. Guidi, High-performance sorting-based K-mer counting in distributed memory with flexible hybrid parallelism, in: Proceedings of the 53rd International Conference on Parallel Processing, 2024, pp. 919–928.
[39] S. Vargas-Pérez, F. Saeed, A hybrid MPI-OpenMP strategy to speedup the compression of big next-generation sequencing datasets, IEEE Trans. Parallel Distrib. Syst. 28 (10) (2017) 2760–2769.
[40] W. Gropp, T. Hoefler, R. Thakur, E. Lusk, Using Advanced MPI: Modern Features of the Message-Passing Interface, MIT Press, 2014.

[41] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.
[42] R. Ceccaroni, L. Di Rocco, U. Ferraro Petrillo, P. Brutti, A distributed workflow for long reads self-correction, in: European Conference on Parallel Processing, Springer, 2024, pp. 105–116.
[43] L. Di Rocco, U.F. Petrillo, R. Giancarlo, G. Cattaneo, Distributed compressive genomics: fundamental pattern matching primitives via spark, Future Gen. Comput. Syst. 176 (2025) 108169.
[44] G. Georgakoudis, L. Guo, I. Laguna, ReiNit++: evaluating the performance of global-restart recovery methods for MPI fault tolerance, 2021, https://arxiv.org/abs/2102.06896.
[45] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, J. Dongarra, Post-failure recovery of MPI communication capability design and rationale, Int. J. High Perform. Comput. Appl. 27 (2013) 244–254. https://doi.org/10.1177/1094342013488238