Scaling Generalized N-Body Problems, A Case Study from Genomics

Marquita Ellis, Aydın Buluç, and Katherine Yelick
University of California at Berkeley and Lawrence Berkeley National Laboratory
Berkeley, California, USA
{mme,abuluc,yelick}@berkeley.edu

ABSTRACT

This work examines a data-intensive irregular application from genomics that represents a type of Generalized N-Body problems, one of the "seven giants" of the NRC Big Data motifs. In this problem, computations (genome alignments) are performed on sparse data-dependent pairs of inputs, with variable cost computation and variable datum sizes. Unlike simulation-based N-Body problems, there is no inherent locality in the pairwise interactions, and the interaction sparsity depends on particular parameters of the input, which can also affect the quality of the output. We build-on a pre-existing bulk-synchronous implementation, using collective communication in MPI, and implement a new asynchronous one, using cross-node RPCs in UPC++. We establish the intranode comparability and efficiency of both, scaling from one to all core(s) on node. Then we evaluate the multinode scalability from 1 node to 512 nodes (32,768 cores) of NERSC's Cray XC40 with Intel Xeon Phi "Knight's Landing" nodes. With real workloads, we examine the load balance of the irregular computation and communication, and the costs of many small asynchronous messages versus few large-aggregated messages, in both latency and overall application memory footprint. While both implementations demonstrate good scaling, the study reveals some of the programming and architectural challenges for scaling this type of data-intensive irregular application, and contributes code that can be used in genomics pipelines or in benchmarking for data analytics more broadly.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; **Distributed computing methodologies**; • **Networks** → *Network measurement*; • **Applied computing** → **Computational genomics**; *Computational proteomics*; Bioinformatics.

KEYWORDS

Generalized N-Body, Big data, asynchronous, bulk-synchronous, irregular applications, irregular all-to-all, many-to-many, benchmarks, bioinformatics, genomics, long read, alignment

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08.
https://doi.org/10.1145/3472456.3472517

ACM Reference Format:
Marquita Ellis, Aydın Buluç, and Katherine Yelick. 2021. Scaling Generalized N-Body Problems, A Case Study from Genomics. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3472456.3472517

1 INTRODUCTION

Generalized N-Body problems [12] are a proper superset of classical N-Body problems, that extends to non-Euclidean and higher dimensional problems. They are characterized by the computation of similarity metrics between many or all pairs or tuples in the input. Naive solutions are $O(N^2)$ or $O(N^3)$... As highlighted in the NRC report, well-established approximations for classical N-Body problems, that reduce the complexity from $O(N^2)$ to $O(NlogN)$ or $O(N)$, such as Barnes-Hut [21] and the Fast Multipole Method [2], do not apply when the similarity metric is non-Euclidean, as in computational genomics. These problems arise from many other domains as well, such as computational physics, chemistry, astronomy, computer vision, and computational biology [5, 8, 23]. To study this class of problems while also contributing to a concrete open problem, our work examines the memory and compute-intensive problem of computing many-to-many *long read alignments* without a reference genome. Given an input set of *long reads*, strings representing DNA fragments from the latest sequencing technology, the problem is to find the best-scoring alignments among all long read pairs – definitions and background from genomics are provided in Section 2. Like other Generalized N-Body problems, one can compute the result by computing the similarity (alignment) of all input long reads to all other – the naive $O(N^2)$, which quickly becomes intractable for large datasets. Rather than relying on fixed physical properties, runtime analysis on the input data is necessary. Domain-specific analysis [4, 7, 13, 15, 17, 22] can effectively reduce the number of interactions (alignments). However, these analyses also typically reveal unstructured sparse (data-dependent) interactions across the input reads, over which the many-to-many pairwise alignment computation must be performed. This poses challenges for parallel load balancing and communication cost minimization. Similar challenges appear in other Generalized N-Body problems from bioinformatics as well, such as fine-grained similarity searches across genomes, metagenome clustering, and protein searches in massive data sets (Section 2). Our work contributes (1) code for solving this specific genomics problem and (2) for facilitating performance characterization of similar problems, and (3) an empirical analysis of two distributed-memory parallelization approaches, a bulk-synchronous one (Section 3.1) and an asynchronous one (Section 3.2). Each approach presents a scalable solution to this specific problem (Section 4), but also presents a set of distinct advantages

Figure 1: The circled matching substrings on the left are used to "seed" the alignment on the right.

Figure 2: Three ways a pair of reads (represented with gray rectangles) can overlap.

and constraints, in terms of computation, communication, load balance, and memory footprint, generalizable to similar problems.

2 APPLICATION BACKGROUND & RELATED WORK

Improvements in DNA sequencing technology have spurred rapid growth in genomic data sets, a trend that has outpaced computing capabilities for analyzing genomic data. DNA sequencers are unable to read entire genomes at once, and thus produce a large number of strings (representing DNA fragments) called *reads* that are short relative to genome sizes. Genome sizes are variable and can be large. For example, *Drosophila melanogaster* (a fruit fly) genome is roughly 144 million base pairs (bps); the wheat genome is much larger at 17 billion bps; and larger still, the flower, *Paris japonica*, is approximately 150 billion bps. The latest sequencing technologies, known as long read sequencers, produce reads that are 3 orders of magnitude longer than previous technologies, but still only 1 thousand to 100 thousand bps long.

Long read sequencers also emit errors at high rates, between 5-35% historically. Sequencer errors include adding a bp into the output read that is not in the genome sequence, excluding a base in the genome from the output read, or substituting a bp at one position in the genome for a different bp in the same position in the read. Additionally, on base calls for which the sequencer has low confidence, it may insert 'N', so the strings for computational analysis have a 5 rather than 4 character alphabet, {'N'}∪ {'A', 'C', 'T', 'G'}. Redundancy in sequencing is used to account for these errors; each region of the genome is sequenced multiple times (a.k.a. sequencing *depth* or *coverage*). Identifying overlaps among the reads and computing their alignments is critical for direct analysis of the read dataset, for correcting errors in the reads, or for reconstructing a more complete representation of the genome from the reads (*de novo* assembly). However, it is also expensive with respect to both memory and computation.

For any pair of reads that might overlap, it is necessary to compute their *pairwise alignment*, in order to determine the direction and extent of overlap (see Figure 2), as well as the edits required to make the overlapping subregions identical. Briefly and informally, pairwise alignment for a given pair of strings (s, t) computes the character substitutions, deletions, or insertions (*edits*) necessary to make s and t match without permutation. For any (s, t) there may be many possible sequences of edits (alignments) that achieve this goal. To quantitatively differentiate pairwise alignments for a given pair of strings, weights are assigned to the various types of character edits (penalties) and to matches (rewards). The sum of the weights for an alignment is the alignment's score. Pairwise alignment algorithms seek the best-scoring alignment for a given input string pair and weighting scheme (*scoring scheme*). Considering the computational complexity and long read lengths, pairwise alignment is not inexpensive in practice. Exact dynamic programming algorithms are $O(n^2)$, if n is the length of the longer read [19][18]. In practice, long read lengths range within and beyond $[10^3, 10^5]$ characters; and for the many-to-many long read alignment problem, *many* such pairwise alignments are computed.

However, *seed-and-extend* pairwise alignment algorithms, which can be average-case $O(n)$, are particularly practical for long-read to long-read alignment, not only for their lower complexity but also for their expected result [1][13]. Seed-and-extend pairwise alignment treats a common substring(s), the seed(s), as fixed between the two strings, and extends the alignment backward and/or forward (illustrated in Figure 1). The intuition behind seed-and-extend alignment is that the best alignment for a pair of reads should include any matching, error-free substrings the two reads have in common. If such substrings can be detected prior to alignment, they can be used as *seeds*. With the high error rates of long reads, and the expectation of overlap (Figure 2) rather than end-to-end alignment in the common case, seed-and-extend alignment is well-suited to efficiently find an alignment corresponding to the true region of overlap.

This provides an overall, average-case complexity of $O(n \times N^2)$ versus $O(n^2 \times N^2)$, where N is the number of reads in the input. However, since N depends on the size of the genome and the sequencing depth, $O(N^2)$ quickly becomes intractable. Hence, most existing approaches heuristically identify pairs of reads that are likely to align well. Pairwise alignments are then computed only for those (a many-to-many computation over the reads), rather than for each read against every other read (an $O(N^2)$ all-to-all computation). One of the most popular methods for identifying these "candidate" overlaps is exact substring (*k-mer*) matching [7, 13, 17, 22]. In short, *k-mers* are computed by moving a sliding window of length k over the reads one character at a time, producing $O(\Gamma \times d)$ *k-mers* for a genome of size Γ and a coverage of d. Note, small k (order 10) is typical in the presence of high error rates. These *k-mers* are filtered implicitly [17] or explicitly [13][11], and only pairs of reads with matching (filtered) *k-mers* are considered overlap "candidates". Filtered *k-mers* can then be used to *seed* the seed-and-extend pairwise alignments.

These data-dependent approaches, while effective, tend to yield large sparse unstructured graphs. The connections between reads in the graph are only discovered through runtime analysis. Further the connections (determined by matching filtered *k-mers*), may be false positives. The cost of individual pairwise alignments can therefore vary drastically, not only from variable read and overlap lengths, but also from early-termination heuristics triggered by false positives - a common technique.

In short, the variability in genome sizes, sequencing coverage, read lengths, and alignment costs lead to highly variable units of computation and communication in this application. These sources

of irregularity are shared by a number of other bioinformatics applications as well. Whole genome to whole genome alignment, for refining or constructing reference genomes for example, computes alignments on genome-length strings. Metagenome assembly –reconstructing the genomes of many organisms in an e.g. soil sample– also involves fine-grained string comparison with highly variable string lengths. Protein searches in massive data sets is another example, with typically shorter reads but also a 20 character alphabet [20].

3 PARALLEL APPROACHES & IMPLEMENTATIONS

Our representative case study builds-on DiBELLA [11], a 3-stage bulk-synchronous pipeline for long-read to long-read alignment. The 1st stage partitions the input reads uniformly by size – a data-independent strategy in that no characteristic other than size in memory is considered. Between the 1st and 2nd stages, it computes a *k-mer* histogram and filters *k-mers* (seeds) based on user criteria, then redistributes the discovered pairwise alignment tasks. The task redistribution preserves the invariant that each task is assigned to the owner of one or both of the required reads, such that the (number) of tasks are roughly balanced across the processors. If an assignee owns one but not both reads, it must retrieve the remotely owned read in order to complete the task.

We examine two approaches for coordinating the subsequent irregular communication and computation. One is bulk-synchronous and prioritizes bandwidth-utilization and message cost amortization via message aggregation. Our implementation extends and refactors DiBELLA's original implementation, as described in Section 3.1. The original DiBELLA focused on a *first* distributed-memory solution to this interdisciplinary problem, and it provided a way to study the data, input, output, and communication and computation patterns for distributed memory scalability (including the challenge of working dataset size explosion and managing memory for communication). Having learned much about these from DiBELLA [11], we were able to make connections across similar problems, placing them in the broader context of Big Data frontiers, and further, provide a more advanced bulk-synchronous solution as well as an alternative asynchronous solution here. The asynchronous solution prioritizes injection speed and communication-hiding via asynchronous communication and computation of the irregular interactions. It simultaneously minimizes memory footprint. We implemented it from scratch, as further described in Section 3.2. Our empirical performance analysis (Section 4) examines the balance of communication, computation, load imbalance, and memory footprint in practice, using our two implementations.

3.1 Bulk-Synchronous Approach

The bulk-synchronous approach we examine exchanges reads in an irregular all-to-all and then computes the pairwise alignments independently in parallel. It uses message aggregation to maximize both bandwidth utilization and message-cost amortization. Given the data-intensive nature of the application, the balance of per-node memory to bisection bandwidth is a significant factor in this approach. Across architectures, per-node memory can limit exchange (message buffer) sizes and therefore also limit effective bandwidth utilization, and increase total synchronization costs by increasing the necessary number of bulk-synchronous supersteps. Per-node memory constraints are particularly relevant for small distributed memory architectures and clusters, and architectures (large or small) in which the ratio of memory to compute resources is relatively low. This includes not only "skinny" node architectures but also "fat" node architectures in which fully utilizing the compute resources severely limits per-core memory, such as on various multi- and many-core architectures - the experimental evaluation includes one such many-core architecture, Cori KNL at NERSC.

To examine this set of trade-offs in practice, we refactored DiBELLA's 3rd stage (see Section 2) to potentially perform multiple, dynamically-sized communication and computation rounds, depending on the workload and per node memory limits. The read exchange is implemented with MPI_Alltoall and MPI_Alltoallv routines. All pairwise alignments associated with each received read are computed together, when the respective read is accessed from the message buffer. Dynamically sizing the supersteps enables us to study the performance implications of memory-limited exchanges with real workloads, it supports performance analysis at both small and large scale, and also contributes to the usability and portability of DiBELLA.

3.2 Asynchronous Approach

Rather than amortizing message costs via aggregation, our asynchronous approach seeks to hide message costs with the pairwise alignment computation and maximize injection speed. Each task involving a remote read b and local read a is indexed under b; those tasks for which both reads are local may be indexed under either. Once the tasks are organized by remote read, the single program multiple data (SPMD) algorithm proceeds by issuing asynchronous requests for each remote read in parallel. A callback is attached to the request; once a remote read b arrives, all alignment computations involving b are executed as soon as they are dequeued. This "pull" rather than "push" approach avoids memory overflow issues from potentially many parallel processors -unaware of each other- pushing reads to the same target. As in the bulk-synchronous algorithm, only those alignments which meet or exceed the user or default scoring criteria are saved for output. A single exit barrier ensures the partitioned reads remain available to all parallel processors until all tasks are complete.

In contrast to the bulk-synchronous approach, this approach roughly maximizes rather than minimizes the number of messages, except in that parallel processors retrieve remote reads no more than once, and pay the round-trip latency for each variable-sized remote read. However, the approach also requires no more than 1 remote read in-memory at any given time in order to make progress, relatively minimizing memory footprint. **Whether the irregular computation can sufficiently overlap (hide) the extra communication in practice, with state-of-the-art runtime support, is one of the questions** Section 4 considers.

We implemented this algorithm in UPC++, a C++ library for high-performance asynchronous communication and computation. UPC++ supported by GASNet-EX [6] claims lower one-sided message latencies and better programmability [3] over other asynchronous languages and libraries. Given that read lengths are highly

variable and that, therefore, the *number* of reads per parallel processor is non-uniform, our implementation employs UPC++ remote procedure calls (RPCs) to lookup and return reads from remote data structure partitions. It has been demonstrated [14] that RPCs can outperform remote direct memory accesses (RDMA) for large messages and for data structure lookups that involve both an index lookup and retrieval of the data itself. This most closely matches our use-case. We leave a thorough investigation of RDMA versus RPC performance for our application to future work however. One other consideration for implementations in general are the progress guarantees of the underlying language and runtime regarding RPCs and callbacks. In our UPC++ implementation, application-level polling is required to ensure read requests are answered and callbacks are processed. Beyond this, GASNet-EX ensures read requests and callbacks are delivered, under the usual assumptions about the network.

After each parallel processor loads its reads, it enters the first phase of a UPC++ split-phase barrier and computes the tasks for which it has both reads locally (during the time it would otherwise be waiting at a regular barrier). Parallel processors will only exit the second phase of the barrier once all reads are accessible via RPC-lookup, however, to support irregular global accesses. Any waiting time in the split-phase barrier (though little to none is expected) and in the final barrier is included in our performance analysis as synchronization time.

4 EMPIRICAL RESULTS

Experiments were conducted on a Cray XC40, Cori KNL, at the National Energy Research Scientific Computing Center (NERSC). Each Cori KNL node has 96 GB of DDR memory and 16 GB of MCDRAM high-bandwidth memory. Each node is single socket, Intel Xeon Phi Knights Landing processor with 68 cores per node @ 1.4 GHz, and each core is 4-way hardware hyperthreaded. Fully utilizing the compute resources therefore heavily constrains available memory per parallel processor. The nodes are connected by the Cray Aries interconnect with Dragonfly topology.

The three real data sets shown in Table 1 were used in the evaluation. The smallest, E. coli 30×, was used for intranode performance measurements, as it can be processed within the memory of a single node in a reasonable amount of time - within an hour on a single KNL core. The largest dataset, Human CCS, was used for multinode strong scaling measurements. An intermediate data set, E.coli 100×, was used to measure the strong scaling performance, from 1 node to over 100 nodes, under conditions optimal for the bulk-synchronous code – optimal strong scaling conditions for the bulk-synchronous code enable it to exchange reads in a single round of communication, maximally utilizing bandwidth and minimizing per-round communication synchronization, across scales. This is not possible at all scales with the larger Human CCS workload, and the performance impact is shown in the following results.

For each approach's results shown side-by-side, the alignment tasks computed from each dataset, and their partitioning, are treated as fixed inputs. DiBELLA is used to perform the data analysis, including computing seeds for pairwise seed-and-extend alignment. For DIBELLA's seed (*k-mer*) computation and filtering, k was set to 17 and the maximum frequency of retained *k-mers* for each

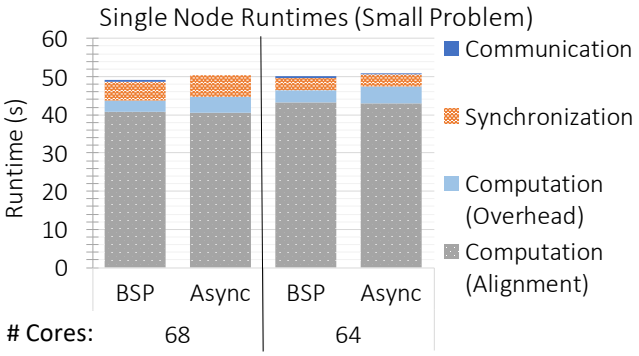

Figure 3: Bulk-synchronous (BSP) and asynchronous (Async) implementations processing the *E. coli* 30× workload on 1 Cori KNL node, with 64 cores running the application and 4 cores isolating system overhead (right) versus all 68 cores running the application (left). At both core counts, the overall runtime difference between the two codes is $< 0.1s$ (and within 0.1% of the overall runtime).

dataset was set according to the BELLA model [13]. BELLA utilizes each dataset's particular sequencing coverage, error rate, and k for its frequency calculation. For individual seed-and-extend pairwise alignments, we employ a performant C++ implementation of X-drop [25] from the SeqAn library [9] in both of our codes. One seed is extended per candidate overlap, simulating expected advances in seed-selection techniques, and focusing the performance analysis on communication performance and load balance.

Each data point is a median global result from many runs, across multiple allocations. Within each run, statistics (minimum, maximum, averages, and sums) are computed via global reductions across parallel processors. These reductions are excluded from the runtime analysis. Lastly, time spent in I/O is left-out of the presentation. Scalable parallel file I/O is employed in each version, but the implementations are different, and file I/O is not the focus of this work. Other differences in computational overhead due to implementation differences are presented in Sections 4.1 and 4.6.

4.1 Single Node Performance

The single node evaluations examine the intranode-scalability and comparability of the two codes, and establish single-node settings for later multi-node experiments. Figures 3-4 present results for two real workloads. The *E. coli* 30× workload is used to collect intranode strong scaling results, as it is small enough to process within the memory of a single Cori KNL node, in reasonable time for repeated experiments. Given this workload, both codes scale perfectly by powers of 2 from 1 to 32 cores, but the speedup tapers-off to $\approx 62\times$ with ≥ 64 cores. Therefore, Figure 3 presents a runtime breakdowns for 64-68 cores, distinguishing useful work from communication and synchronization overheads. The 64-core runs dedicate the 4 additional cores to system overhead isolation. With 68 versus 64 cores (left vs. right in Figure 3), the slight improvement in computation time is cancelled-out by a slight increase in overheads, primarily synchronization overhead. At both core counts, the overall runtime difference between the two codes is < 0.1s (within 0.1% of the

Table 1: Workloads used for evaluation. For each data set, we list a short name for the dataset, the scientific name of the species, the total number of reads, the total number of pairwise alignments (equal to the number of seeds extended in our experiments - one per candidate overlap), and the link to the raw read data. *Exploring 1 seed per overlap candidate.

Short Name	Species	Reads	Tasks	Raw Data Source
E. coli 30×	*Escherichia coli*	16,890	2,270,260	https://bit.ly/2EEq3JM (CBCB)
E. coli 100×	*Escherichia coli*	91,394	24,869,171	https://bit.ly/2POV1Qs (NCBI)
Human CCS	*Homo sapiens*	1,148,839	87,621,409	https://tinyurl.com/y73tfgnw (NCBI)

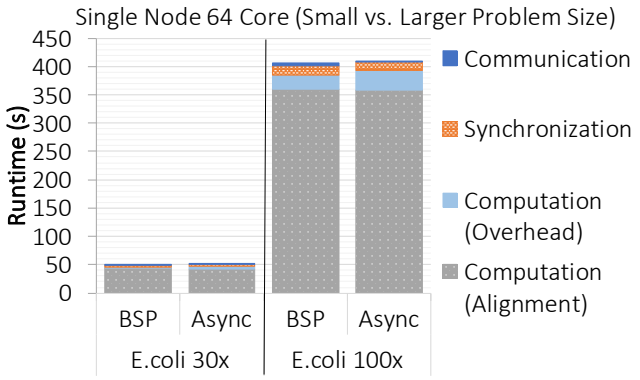

Figure 4: Bulk-synchronous (BSP) and asynchronous (Async) code runtime breakdowns on 2 problem sizes, run on 1 Cori KNL node with 64 cores running application code and 4 cores isolating system overhead. The left-hand results are also shown on the right-hand-side of Figure 3. The runtime of the larger problem (*E. coli* 100×) is ≈ 94% compute-dominated, versus ≈ 90% for the smaller problem (*E. coli* 30×). For the larger problem (*E. coli* 100×), the overall runtime difference between the codes is ≈ 1s (< 0.3% of the minimum runtime).

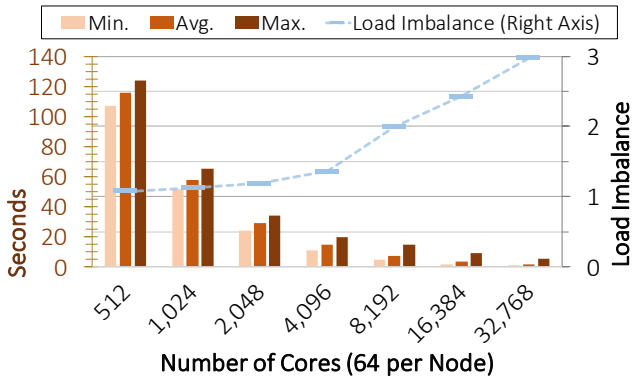

Figure 5: Minimum, average, and maximum times spent in SeqAn seed-and-extend calls cumulatively (left axis) and load imbalance (right axis), strong scaling *Human* CCS on Cori KNL.

overall runtime), and the absolute time-to-solution is effectively reduced from ≈ 1 hour to ≈ 1 minute (on 1 vs. 64-68 cores).

Figure 4 shows the single-node runtime breakdown for *E. coli* 100×, a much larger workload – an estimated ≈ 7 hours would be needed to process this workload on a single Cori KNL core. The overall runtime difference between the two codes is ≈ 1s (< 0.3% of the overall runtime). Together with the *E. coli* 30× results, these establish the comparability of the two codes in terms of single-node baseline performance.

In both Figures 3 and 4, the "Computation (Overhead)" includes data structure traversal, function call overhead, etc. for initializing and invoking the seed-and-extend kernel from the SeqAn library [9]. Though a small fraction of the runtime, this overhead is slightly higher for the asynchronous code, we therefore present additional results for the largest workload, *Human* CCS, with other multinode strong scaling results later. Computing the actual seed-and-extend pairwise alignments ("Computation (Alignment)") dominates the runtime across all scales and experiments. Pairwise alignments are balanced across cores by number but not necessarily by cost, since the seed-and-extend pairwise alignment cost varies dynamically with the lengths of the reads, the length of their "true overlap", the speed of "false positive" detection (early termination), and so on. The synchronization time is dominated by this load imbalance, on which the next section provides further results.

Based on these and additional results showing negligible or no benefit from employing 68 versus 64 cores or hardware hyper-threads on Cori KNL for this and similar applications [10], we use 64 cores with 4 cores dedicated to system overhead management as our default setting. The next section shows the results strong scaling the two codes processing *E. coli* 100× workload from a single node with 64 cores (also shown in Figure 4) to 128 nodes (8,192 cores), with additional multinode strong scaling results for the *Human* CCS workload (Table 1).

4.2 Load Imbalance

The work is partitioned statically by number of alignments, but individual seed-and-extend pairwise alignments may have variable costs. The variability in these costs is the source of much of the overall load imbalance (see Figure 5). The costs vary by read lengths and runtime parameters (for example, the value of X for the X-drop algorithm [24]) and cannot be easily determined before runtime. This highlights an opportunity to explore dynamic or semi-static approaches to load balancing alignment tasks in future work.

Variability in read lengths additionally affects communication load imbalance. Figure 6 shows communication load imbalance for the bulk-synchronous exchanges, strong scaling the *Human* CCS workload, in terms of the amount of received read data per processor (core). As shown, there is a large difference between the

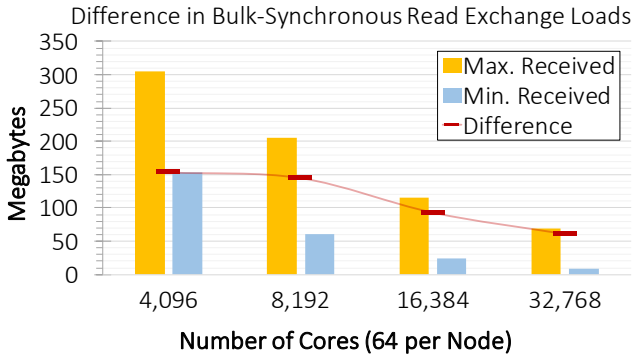

Figure 6: From strong scaling *Human* CCS on Cori KNL, the difference between the maximum and minimum bulk-synchronous exchange loads.

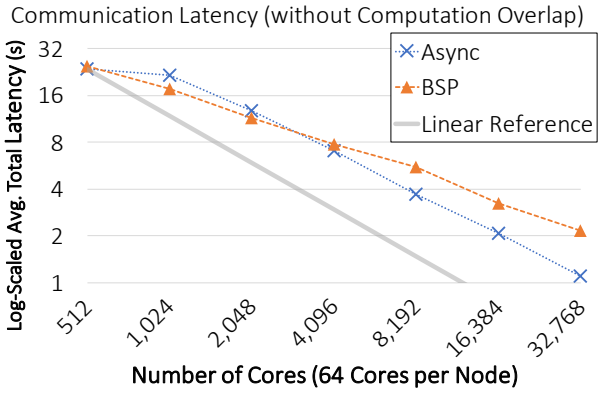

Figure 7: Total average communication latency from strong scaling *Human* CCS on Cori KNL with computation skipped.

minimum and maximum loads, which might explain some of the poor communication scalability, in terms of latency, shown in the next section.

4.3 Absolute Communication Latency

In order to understand the absolute (unhidden) communication latency of the asynchronous code using RPCs, we implemented a mode that executes everything (*except* the pairwise alignment computation). Though it is simpler to extract communication performance from the bulk-synchronous code, we also implemented this mode in the bulk-synchronous code for communication-focused benchmarking. Figure 7 shows the communication latency from strong scaling the *Human* CCS workload in this mode. The reason for the poor scaling of the asynchronous version between 8-16 nodes requires further investigation, but because of the high numbers of outgoing and incoming RPCs at those scales, we speculate that further tuning runtime parameters to the workload (e.g. varying limits on outgoing requests) could improve overall latency. From 16-512 nodes, however, it appears the message latencies scale with the workload – as the number of parallel processors increase, the

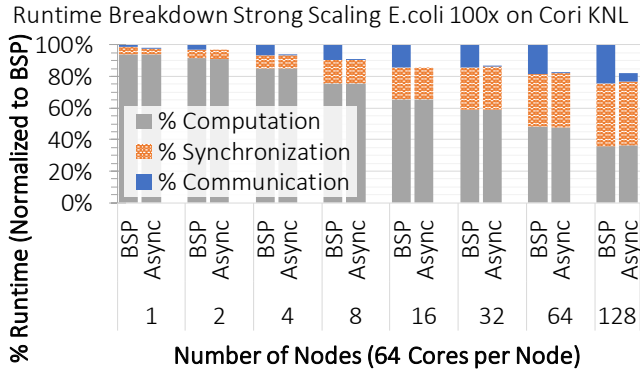

Figure 8: Comparative runtime breakdown, strong scaling *E. coli* 100× on Cori KNL. The computation and synchronization time between the asynchronous (Async) and bulk-synchronous (BSP) performance is the same. Because Async successfully hides most of its communication latency, the runtime normalized to the BSP runtime is lower than 100%. In contrast, the bulk-synchronous (BSP) version's communication latency increases from over 1% of its runtime on a single node (64 cores) to over 24% on 128 nodes (8K cores), despite having enough memory to communicate in a single step (maximizing its bandwidth utilization and minimizing its latency).

number of lookups per processor decrease and so does the total latency. The bulk-synchronous communication latency is lower than the asynchronous latency, but scales sublinearly from 8-512 nodes, resulting in the performance cross-over between 32-64 nodes.

4.4 Computation-Communication Overlap

We compare the asynchronous versus the bulk-synchronous multinode performance using two real worloads, *E. coli* 100× and *Human* CCS. *E. coli* 100× is small enough to process on a single node, but the raw input data is over 3× larger (and the number of alignments is nearly 11× larger) than *E. coli* 30×. We employ it for a strong scaling comparison in which there is sufficient per-processor memory for the bulk-synchronous version to exchange all reads at once, achieving its lowest communication overhead. The second workload, *Human* CCS, is roughly 28× larger than the *E. coli* 100× data set, with respect to raw input sizes, and the initial stages of the DiBELLA pipeline, including the analysis necessary to compute alignment tasks, cannot complete with fewer than (4, 8] Cori KNL nodes. For all experiments, processors are pinned to each full core (exclusive L1 cache) on a node, except for 4 cores dedicated to system overhead isolation, for a total of 64 cores running the application per node (see Section 4.1).

Figure 8 shows results strong scaling the *E. coli* 100× workload from 1 to 128 Cori KNL nodes (64 to 8K cores). The absolute runtime of both versions with 8K cores (128 nodes) is ≈ 10 seconds, a roughly 40× speedup over the single node time. The time spent in computation (seed-and-extend pairwise alignment) and synchronization are practically the same between the two versions. The synchronization time results primarily from load imbalance in computation due

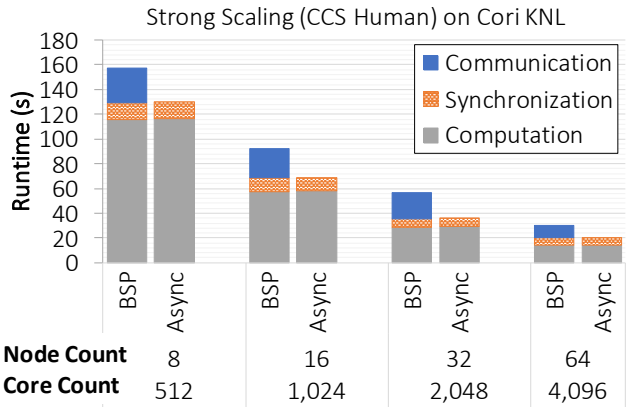

Figure 9: 8 to 64 nodes (512-4K cores) comparative runtime breakdown of the bulk-synchronous (BSP) and asynchronous (Async) code, strong scaling the *Human CCS* workload. Due to the size of the workload relative to memory on node, the BSP code must perform multiple steps of communication and computation up to 64 nodes.

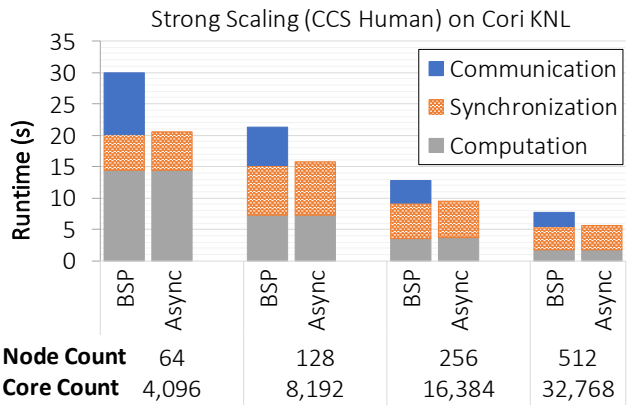

Figure 10: 64 to 512 node (4K-32K cores) comparative runtime breakdown of the bulk-synchronous (BSP) vs. asynchronous (Async) code, strong scaling the *Human CCS* workload. Unlike Figure 9, there is sufficient memory for the BSP code to complete communication and computation in a single superstep.

to variable task costs (see Section 4.2). The visible communication latency sets the two versions apart. The bulk-synchronous version's communication latency increases from just over 1% on a single node (64 cores) to over 24% on 128 nodes (8K cores). In contrast, the asynchronous version manages to hide most communication latency up to 128 nodes (8K cores); the visible latency at 128 nodes (8K cores) is less than 7% of its own runtime. Consequently, the asynchronous version is up to 12% more efficient. Though small, the gap is significant because there is sufficient memory across scales for the bulk-synchronous code to perform a single bulk-synchronous exchange of the reads (achieving its highest bandwidth utilization and lowest communication latency). Even so, successful communication hiding in the asynchronous code yields higher efficiency.

Figures 9-10 show comparative strong scaling results with the *Human* CCS workload. Note, the minimum number of nodes required to process this workload with our setup is 8, as determined by the memory requirement of DiBELLA's initial pipeline stage (see Section 2). The workload is scaled from 8 nodes to 512 nodes (512 to 32K cores respectively). From 8 to 32 nodes (Figure 9), there is insufficient memory for the bulk-synchronous version ("BSP") to complete its read exchanges in a single round; it requires multiple exchange-compute steps. As in the *E. coli* 100× results, the synchronization time between the two versions is practically the same across scales, due to load imbalance from dynamic seed-and-extend computation costs. The communication overhead of the bulk-synchronous version is 17% to 34% of its runtime across scales, while the asynchronous version successfully hides its communication latency. Consequently, the asynchronous version is up to 20% more efficient than the bulk-synchronous version with 8-32 nodes (512-2K cores). With sufficient per processor memory for the bulk-synchronous version to complete the exchange in one step (Figure 10), the efficiency gap decreases from 13% at 64 nodes (4K cores) to 4% at 512 nodes (32K cores).

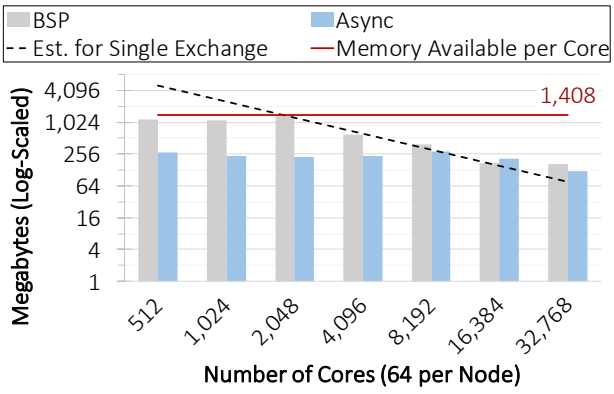

Figure 11: Maximum memory footprint (log-scaled) per core (MB) of the bulk-synchronous (BSP) and asynchronous (Async) approaches, strong scaling Human CCS on Cori KNL. The solid line is the application-available memory per core (<1.4GB). The dashed line is the estimated total memory required to exchange all reads at once. From 512-2K cores (8-32 nodes), BSP performs multiple exchanges limited by available memory. It is only able to perform a single exchange, maximizing bandwidth utilization, between 4K-32K (64-512 nodes). The Async version on the other hand maintains a relatively low memory footprint (<256MB) across scales.

4.5 Memory Footprint

The memory footprint of each approach is shown in Figures 11-12. Figure 11 shows the maximum per core memory footprint of each approach, strong scaling the *Human* CCS workload. This memory footprint information was gathered from NERSC's completed job logs. Also shown is the application-available memory per core (roughly 1.4GB) and the estimated memory required to exchange

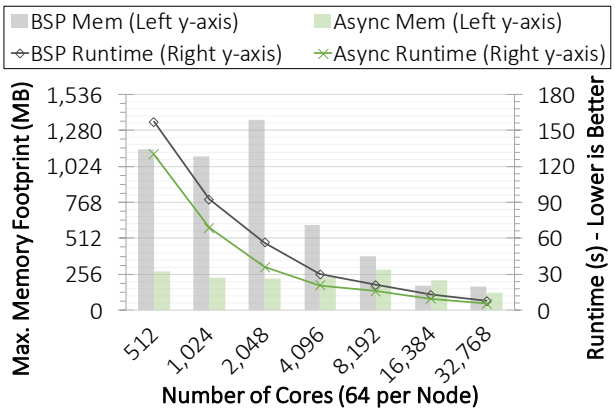

Figure 12: Memory footprint and runtime of the bulk-synchronous (BSP) and asynchronous (Async) versions, strong scaling Human CCS on Cori KNL. Async maintains a relatively low fixed memory footprint while achieving better performance through communication-computation overlap.

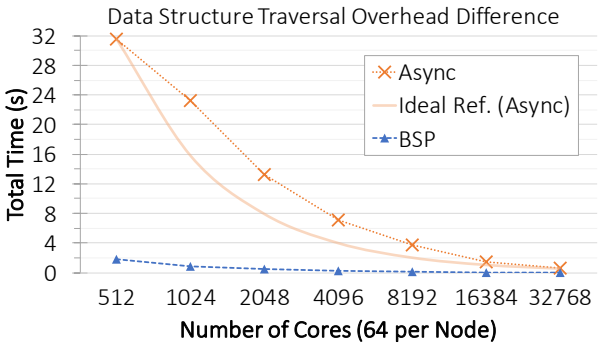

Figure 13: Computational runtime overhead primarily from traversing local data structures storing alignment tasks and associated data, while strong scaling Human CCS on Cori KNL. The BSP code uses flat arrays; the Async code uses C++ standard library data structures. While not of primary focus, the results are presented for completeness.

all reads at once. The estimate is calculated from the total exchange load, divided by the number of processors, plus the average input partition sizes. Between 512-2K cores (8-32 nodes) the bulk-synchronous version (BSP) exchange sizes are limited by available per core memory. From 4K-32K cores (64-512 nodes), the memory footprint matches the estimate fairly closely. The memory footprint of the asynchronous version, on the other hand, remains relatively fixed and below 256MB per core. Figure 12 shows the same memory footprint information on an absolute scale (left axis) along with overall runtimes (right axis). While the asynchronous version maintains a lower runtime via communication-computation overlap (Section 4.4), and typically lower memory footprint, the results are very close at-scale 32K cores (512 nodes).

4.6 Local Data Structure Performance

In the runtime breakdowns in Section 4.1, though negligible to overall performance there, the computational overhead is slightly higher for the asynchronous version, due to local data structure implementation choices. Each code traverses local data structures storing alignment tasks with associated data, in order to issue/buffer requests for remote reads and then compute pairwise alignments. The bulk-synchronous code uses flat arrays, achieving better locality. The asynchronous code uses C++ standard library data structures; while the code is more object-oriented and readable, the trade-off is higher performance overheads. While local data structure optimization is not the focus of this work, for completeness Figure 13 shows the overhead of the data structure traversals in isolation for the largest workload, *Human* CCS. While it scales down to ≈ 4% of the overall runtime, the performance difference between the traversing local flat arrays in the bulk-synchronous code versus the local pointer-based data structures in the asynchronous presents the classic trade-off of performance and programmability.

5 SUMMARY & CONCLUSIONS

This work examined two parallelization approaches to many-to-many long read alignment, relevant to other Generalized N-Body problems, a bulk-synchronous and an asynchronous one. The bulk-synchronous approach prioritized maximizing bandwidth utilization and message cost amortization. The asynchronous approach prioritizes message injection speed and message hiding via communication-computation overlap. Detailed strong scaling results from our respective implementations for 3 real workloads (Table 1) on Cori KNL were presented in Section 4. The relative performance of the two approaches was determined by the performance of communication primitives for many-to-many exchanges and by the performance of the pairwise alignment computation.

The communication latency of the asynchronous approach, prioritizing injection speed, scaled as the workload scaled-out. The number of messages per parallel processor scaled inversely with the number of processors, and the overlapped communication and computation costs in aggregate balanced out. On a high-latency network however, we would expect more aggregation to be necessary – but how much more depends also on the computation costs.

Further, we emphasize that optimizing communication and computation performance is not independent from memory capacity. Increasing message aggregation for better bandwidth utilization and message cost amortization is an effective go-to solution for many applications. For these data-intensive Generalized N-Body problems, the memory enabling (or limiting) message aggregation can limit achievable performance. To analyze the performance of the bulk-synchronous approach under best-case and memory-limited scenarios, we employed a workloads for which the many-to-many bulk-synchronous exchange could be performed in a single, bandwidth-maximizing exchange, within the available memory of 1 to 128 nodes, and a much larger workload for which the many-to-many exchange could not be performed in a single exchange at small scale. The latter revealed significant communication overheads (17-34%) when forced to perform multiple irregular exchanges within

available memory. Overall, Cori's low-latency high-bandwidth network supports both approaches well, but the ratio of computational to memory resources limited the bulk-synchronous message-aggregating approach. Studying the performance across interconnect architectures would be interesting future work. Furthermore, optimizations targeting just the computation will affect the overall performance of each approach differently. For the bulk-synchronous approach, we expect improvements to the computation to decrease overall runtime, and to lower the number of parallel processors at which strong scaling overall application performance becomes communication-bound rather than computation-bound. For the asynchronous approach, we expect overall runtime to improve with alignment computation optimizations until average asynchronous message latency exceeds the average pairwise alignment computation rate, at which point, communication optimizations (e.g. lower latency primitives, message aggregation) will be necessary for any further computational optimizations to be effective. Bisection bandwidth for many-to-many exchanges, supporting bulk-synchronous approaches, versus the (balance of) alignment computation to one-sided or point-to-point message latency, supporting asynchronous approaches, determines the relative performance of each approach across workloads, implementations, and architectures. The analysis is informative for both software and hardware designers seeking to support similar data-intensive Generalized N-Body problems.

The results from both implementations motivate further study of the load imbalance. The synchronization time, dominated by imbalance in variable-cost alignment tasks, was visible between the codes across scales. For all results, DiBELLA's [11] direct "blind" partitioning avoids re-partitioning input reads and balances the (number) alignment tasks per processor with a simple heuristic, but the cost of each alignment task varies dynamically. This work focused on many-to-many communication approaches that are independent of the underlying partitioning approach. That is, for improvements in the underlying partitioning approach, we expect the communication performance of these approaches to only improve. The variability in computational costs here and in similar applications perhaps motivates a dynamic approach, but whether the performance improvements can compensate for the overheads of dynamic load balancing in practice will be the question. Supporting future work in performance analysis and genomics, the code [16] from this study can be used for many-to-many long read alignment with general inputs. It can also be modified to solve similar bioinformatics problems with reasonable effort. Finally, the code can also be used for performance benchmarking and characterization of this and other Generalized N-Body problems.

ACKNOWLEDGMENTS

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and by the National Science Foundation as part of the SPX program under Award number 1823034. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. 1990. Basic Local Alignment Search Tool. *Journal of Molecular Biology* 215, 3 (1990), 403–410.
[2] John Ambrosiano, Leslie Greengard, and Vladimir Rokhlin. 1988. The fast multipole method for gridless particle simulation. *Computer Physics Communications* 48, 1 (1988), 117–125.
[3] John Bachan, Scott Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul Hargrove, and Hadia Ahmed. 2019. UPC++: A High-Performance Communication Framework for Asynchronous Computation. 963–973. https://doi.org/10.1109/IPDPS.2019.00104
[4] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. 2015. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature biotechnology* 33, 6 (2015), 623–630.
[5] Nicolas Bock, Matt Challacombe, and Laxmikant V Kalé. 2016. Solvers for O(N) Electronic Structure in the Strong Scaling Limit. *SIAM Journal on Scientific Computing* 38, 1 (2016), C1–C21.
[6] Dan Bonachea and Paul H. Hargrove. 2018. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. (10 2018). https://doi.org/10.25344/S4QP4W
[7] Mark J. Chaisson and Glenn Tesler. 2012. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics* 13, 1 (2012), 238.
[8] National Research Council et al. 2013. *Frontiers in massive data analysis*. National Academies Press.
[9] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. 2008. SeqAn an efficient, generic C++ library for sequence analysis. *BMC bioinformatics* 9, 1 (2008), 11.
[10] Marquita Ellis, Evangelos Georganas, Rob Egan, Steven Hofmeyr, Aydın Buluç, Brandon Cook, Leonid Oliker, and Katherine Yelick. 2017. Performance characterization of de novo genome assembly on leading parallel systems. In *European Conference on Parallel Processing*. Springer, 79–91.
[11] Marquita Ellis, Giulia Guidi, Aydin Buluç, Leonid Oliker, and Katherine Yelick. 2019. DiBELLA: Distributed Long Read to Long Read Alignment. In *48th International Conference on Parallel Processing (ICPP 2019)* (Kyoto, Japan). https://doi.org/10.1145/3337821.3337919
[12] Alexander G Gray and Andrew W Moore. 2001. N-body'problems in statistical learning. In *Advances in neural information processing systems*. 521–527.
[13] Giulia Guidi, Marquita Ellis, Daniel Rokhsar, Katherine Yelick, and Aydın Buluç. 2021. BELLA: Berkeley efficient long-read to long-read aligner and overlapper. In *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*.
[14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proc. 12th USENIX OSDI*. Savannah, GA.
[15] Heng Li. 2018. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (2018), 3094–3100.
[16] Marquita Ellis. [n.d.]. https://sourceforge.net/p/dibella/wiki/Home/.
[17] Gene Myers. 2014. Efficient Local Alignment Discovery amongst Noisy Long Reads. In *Algorithms in Bioinformatics*, Dan Brown and Burkhard Morgenstern (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–67.
[18] Saul B Needleman and Christian D Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology* 48, 3 (1970), 443–453.
[19] Temple F. Smith and Michael S. Waterman. [n.d.]. Identification of Common Molecular Subsequences. *Journal of Molecular Biology* 147, 1 ([n. d.]), 195–197.
[20] Martin Steinegger and Johannes Söding. 2017. MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets. *Nature biotechnology* 35, 11 (2017), 1026–1028.
[21] Laurens Van Der Maaten. 2013. Barnes-hut-sne. *arXiv preprint arXiv:1301.3342* (2013).
[22] Chuan-Le Xiao, Ying Chen, Shang-Qian Xie, Kai-Ning Chen, Yan Wang, Yue Han, Feng Luo, and Zhi Xie. 2017. MECAT: fast mapping, error correction, and de novo assembly for single-molecule sequencing reads. *Nature Methods* 14, 11 (2017), 1072.
[23] Katherine Yelick, Aydın Buluç, Muaaz Awan, Ariful Azad, Benjamin Brock, Rob Egan, Saliya Ekanayake, Marquita Ellis, Evangelos Georganas, Giulia Guidi, et al. 2020. The parallelism motifs of genomic data analysis. *Philosophical Transactions of the Royal Society A* 378, 2166 (2020), 20190394.
[24] Alberto Zeni, Giulia Guidi, Marquita Ellis, Nan Ding, Marco D Santambrogio, Steven Hofmeyr, Aydın Buluç, Leonid Oliker, and Katherine Yelick. 2020. LOGAN: High-Performance GPU-Based X-Drop Long-Read Alignment. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Vol. 1. 462–471.
[25] Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb Miller. 2000. A greedy algorithm for aligning DNA sequences. *Journal of Computational biology* 7, 1-2 (2000), 203–214.