PHILOSOPHICAL TRANSACTIONS A

royalsocietypublishing.org/journal/rsta

Discussion

**Cite this article:** Yelick K *et al*. 2020 The parallelism motifs of genomic data analysis. *Phil. Trans. R. Soc. A* **378**: 20190394. http://dx.doi.org/10.1098/rsta.2019.0394

Accepted: 9 December 2019

One contribution of 15 to a discussion meeting issue 'Numerical algorithms for high-performance computational science'.

**Subject Areas:**
computational biology

**Keywords:**
high-performance data analytics, bioinformatics, parallel computing

**Author for correspondence:**
Katherine Yelick
e-mail: yelick@berkeley.edu

# The parallelism motifs of genomic data analysis

Katherine Yelick[1,2], Aydın Buluç[1,2], Muaaz Awan[1], Ariful Azad[3], Benjamin Brock[1,2], Rob Egan[4], Saliya Ekanayake[1], Marquita Ellis[1,2], Evangelos Georganas[5], Giulia Guidi[1,2], Steven Hofmeyr[1], Oguz Selvitopi[1], Cristina Teodoropol[1,2] and Leonid Oliker[1]

[1]Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[2]Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, USA
[3]School of Informatics, Computing, and Engineering, Indiana University, Bloomington, IN, USA
[4]DOE Joint Genome Institute, Walnut Creek, CA, USA
[5]Intel Labs, Santa Clara, CA, USA

KY, 0000-0003-0957-701X

Genomic datasets are growing dramatically as the cost of sequencing continues to decline and small sequencing devices become available. Enormous community databases store and share these data with the research community, but some of these genomic data analysis problems require large-scale computational platforms to meet both the memory and computational requirements. These applications differ from scientific simulations that dominate the workload on high-end parallel systems today and place different requirements on programming support, software libraries and parallel architectural design. For example, they involve irregular communication patterns such as asynchronous updates to shared data structures. We consider several problems in high-performance genomics analysis, including alignment, profiling, clustering and assembly for both single genomes and metagenomes. We identify some of the common computational patterns or 'motifs' that help inform parallelization strategies and compare our motifs to some of the established lists, arguing that at least two key patterns, sorting and hashing, are missing.

THE ROYAL SOCIETY
PUBLISHING

This article is part of a discussion meeting issue 'Numerical algorithms for high-performance computational science'.

## 1. Introduction

The future of scientific computing will be increasingly data intensive due to the growth of data from sequencers, telescopes, microscopes, light sources, particle detectors and embedded environmental sensors. Open data policies for scientific research are leading to large community datasets of both raw and derived data. Some of resulting data analysis problems involve massive numbers of independent computations, while others require irregular computations in which the objective of the analysis is to discover the underlying structure of the data. Many genomics problems fall into this latter category, where the structure and relationship between different sequences or entire genomes is unknown. These problems require data structures like hash tables, histograms, graphs and very sparse unstructured matrices. They have dynamic sources of load imbalance and little locality, leading to unpredictable communication that is both irregular in space, with arbitrary connections between processors, and irregular in time, where one process may need data on another at any point in time.

In this paper, we describe parallelization challenges and approaches for high-performance genomic data analysis using a series of examples drawn in large part from the ExaBiome project, including k-mer counting, alignment, genome assembly, protein clustering and machine learning. We consider analysis of both DNA and proteins expanding beyond the strict domain of genomics into proteomics. Shared memory programming is a natural fit for these problems, and indeed the most popular genome assemblers and clustering algorithms have typically run on shared memory computers. In developing high-performance computing (HPC) implementations of these applications, we use distributed versions of shared data structures that are updated asynchronously by individual processors with minimal global synchronization.

By contrast, the applications that dominate HPC workloads are scientific simulations that have a natural degree of locality from the underlying physical laws. These simulations often lend themselves to domain decomposition, where the physical domain is partitioned across processors, and while communication may be both global and to nearest neighbours, the presence of timesteps and iterative methods lead to natural phases of communication and computation separated by global synchronization. Figure 1 shows a notional spectrum of simulation and analysis problems and the level of irregularity which tends to correlate with the difficulty of parallelization. On the left are independent parallel jobs, whether from simulation or analysis. These are easily parallelized on a cluster or cloud platform using programming systems like Spark [1], or even geographically distributed computing as in the grid [2]. Simulation problems with physical structure fall in the middle two categories, depending on whether they have global patterns of communication and synchronization, which often stress the global network bandwidth but are simpler to reason about, or involve pairwise exchange of data using synchronous or asynchronous two-sided message passing. These boundaries are neither strict nor precise, with many applications having a mixture of styles, and deep learning landing with simulation. However, the spectrum highlights that the genomics applications will provide an interesting perspective for the design of parallel hardware and software systems.

In addition to summarizing parallelization techniques for genomics analysis, we identify a relatively small set of computational *motifs* that appear multiple times across applications, illustrated in figure 2. While we do not presume that these motifs are sufficient for all such applications, we believe they can substantively inform the design of libraries, programming systems, benchmarks and hardware. Following a brief overview of the ExaBiome project in §§2 and 3 gives an overview of several genomics data analysis problems and the computational motifs that will lead to a particular parallelization strategy. Section 4 summarizes our genomics motifs and compares them to other lists of motifs, showing strong similarity to another list for data

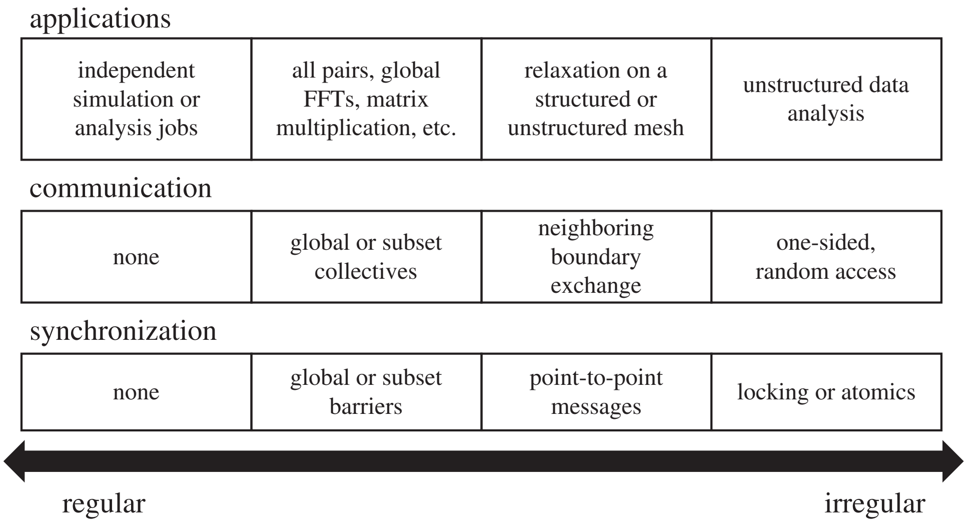
applications
independent simulation or analysis jobs
all pairs, global FFTs, matrix multiplication, etc.
relaxation on a structured or unstructured mesh
unstructured data analysis
communication
none
global or subset collectives
neighboring boundary exchange
one-sided, random access
synchronization
none
global or subset barriers
point-to-point messages
locking or atomics
regular
irregular

**Figure 1.** A spectrum of regularity with different patterns of communication and synchronization. Data analysis is often at the two extremes.

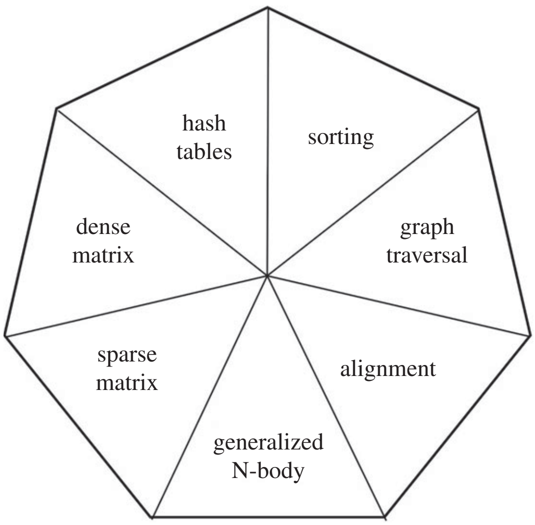

**Figure 2.** Seven parallelism motifs in genomic data analysis.

analysis problems more broadly, although we argue that two key motifs, sorting and hashing, are missing. Section 5 describes how the motifs lead to different types of programming support and hardware requirements, and §6 makes some concluding remarks on the value and limitations of our motifs.

## 2. ExaBiome overview

The ExaBiome project is developing scalable parallel tools to analyse microbial species, such as bacteria, fungi or viruses, which typically live in communities with hundreds of different species mixed together. The genome-level analysis of these communities, *metagenomics*, is key to understanding the makeup of these communities, how they change based on external factors like temperature, moisture or chemicals, to understand their functional behaviour, and to compare them across communities. An estimated 99% of microbial species are not culturable in isolation, making metagenome analysis the preferred technique for understanding these communities. The human microbiome has been linked to a wide range of health issues including diabetes, cancer and mental health, while environmental microbiomes can have both positive or negative impacts on everything from oxygen production and remediation of chemical spills to formation of toxic algal blooms.

ExaBiome, which is part of the Exascale Computing Project [3], is developing HPC solutions for problems that were predominantly computed on shared memory or serial machines, and taking advantage of the processor accelerators such as Graphics Processing Units (GPUs) that are key to future exascale system designs. The project is developing assemblers for both short and long read sequencing data (MetaHipMer and diBELLA), taking the fragmented output of sequencers and constructing long sequences from which genes, corresponding proteins, and taxonomic information can be derived. Working across large protein datasets, PISA and HipMCL extract clusters of related proteins that are useful in understanding ancestry and functional behaviour. The team is also exploring deep learning techniques to relate proteins to three-dimensional structure and function, and a set of methods to compute signatures for metagenomes that can be used for comparisons across microbial samples in space or time and database search.

The project has already demonstrated unprecedented scales in terms of dataset size and performance with the goal of growing the dataset capability by more than an order of magnitude. The largest metagenome assembly to date used wetland soil samples that were a time-series dataset across several physical sites from the Twitchell Wetland in the San Francisco Bay-Delta. These samples consisted of 2.6 terabytes of 7.5 billion *reads*, which are the DNA fragments output from the sequencers, and the assembly computation required 5.1 h on 1024 nodes of the Cori supercomputer at NERSC. We believe it is the largest assembly of any kind done as a single co-assembled computation, i.e. rather than pre-filtering the data in some way or assembling pieces of the data separately. Separate analysis shows the value of such co-assemblies, especially in extracting information about the low abundance species in a sample. The largest protein clustering computation used assembled metagenomes and metatranscriptomes from two community datasets (IMG and NCBI). This unprecedentedly large dataset contained 383 million proteins and 37 billion connections, requiring about 1 h on 729 nodes of the Summit system at OLCF.

## 3. A sampling of genomic analyses

We describe at a high level some of the algorithms and parallelization approaches used in genomic data analysis, selecting a set of problems that represent a diverse set of computational patterns and are prevalent across multiple applications. Our primary focus is on distributed memory parallelization techniques, so we describe the data distribution and communication approaches as well as any load balancing issues, or limitations to scaling when they exist.

### (a) K-mer analysis

Given a set of variable-length strings, a common approach to analysing those strings is to break them into fixed-length substrings called k-mers. For example, the string on the left has the list of 4-mers on the right.

| CCTAAAGCCTA | CCTA CTAA TAAA AAAG AAGC AGCC GCCT CCTA |
|---|---|

Several bioinformatics analyses involve counting the number of occurrences of each distinct k-mer, e.g. to filter low-frequency k-mers that are likely errors, to find high-frequency k-mers that indicate repetitive regions of the genome, or by using the k-mer histogram as a signature for a set of genomics data. K-mers also serve as seeds in determining whether two DNA fragments are likely to align with one another and may also be used on protein data with its 21-character amino acid alphabet in addition to DNA. The most common approach to k-mer counting is to build a hash table of k-mers, possibly using a Bloom filter, an approximate and space efficient data structure that answers queries about set membership, to eliminate singletons. If the k-mer length is small, a direct map may be practical, and sorting is also possible, although to keep memory use in check, k-mers are generated incrementally and identical ones merged while they are sorted to avoid having all of them in memory at once. Manekar and Sathe give an overview of the various approaches and benchmark some of the popular shared memory tools [4].

Distributed memory parallelism for k-mer counting becomes increasingly important to address large sets of environmental microbial genomes and to handle cross-genome comparisons. Raw sequencing data may be several times larger than the final genome, e.g. the dataset may be sequenced repeatedly giving it a sequence *depth* of 10–50× to ensure that every location is sequenced multiple times so that sequencing errors can be eliminated. Large environmental datasets may therefore run to multiple terabytes resulting in input data that does not fit on a single large-memory compute node. The list of k-mers—prior to removing duplicates—requires storage nearly $k$ times larger than the input, and given that typical k-mer lengths may run from 10 to 50 characters, the raw k-mers may not fit even in the aggregate memory of a multi-node system. The ExaBiome project uses a hash-based approach to k-mer counting with a Bloom filter used to avoid storing most of the singleton k-mers. The resulting hash table stores the count of each unique k-mer that occurs more than once in the original data.

In general, there is no communication locality when building the distributed hash table, so on p processing nodes, each k-mer will be communicated remotely with probability (p-1)/p. This creates an irregular many-to-many communication pattern without any predetermined patterns and without natural points for bulk-synchronous communication. A Bloom filter is useful to avoid storing singleton k-mers, but requires the same irregular many-to-many communication as the hash table: all k-mers are communicated, but the Bloom filter requires only a few bits for each unique k-mer. A good hash function can ensure even load balance of the unique k-mers, but significant communication load imbalance still results when the frequency distribution is skewed, as is often the case in real datasets where there are some very high-frequency k-mers. Local aggregation of such 'heavy hitters' can reduce communication bottlenecks for high-frequency k-mers, but the effectiveness depends on having a small number of such k-mers so that a local table can collect and combine them.

Within the ExaBiome project, we have multiple instances of k-mer analysis, which include a basic count/histogram operation and indexing to collect the information about the position of each k-mer in the set of input sequences (reads). Memory utilization is a key factor in design, and to avoid having the full list of k-mers (with duplicates) in memory at any given point in time, one version of the code performs all-to-all exchanges in phases [5–8] and counts k-mers for use in the short read assembly, and another version keeps indexing information for use in computing long read overlaps [9]. Other distributed memory k-mer analysis tools include Bloomfish, which uses a similar MPI all-to-all collective approach but has only a single phase [10] thus limiting dataset size due to memory constraints, and Kmerind, which has demonstrated scaling to over 20 TB datasets by using multiple phases and various memory saving optimizations [11]. The most recent ExaBiome k-mer counting tool is entirely without global synchronization and uses one-sided communication to continually send k-mers while combining and storing local ones. Not only does it avoid global synchronization, but it also hides latency by using non-blocking communication.

## (b) Pairwise alignment

Alignment is performed on both DNA and proteins to find approximate matches between strings, allowing for a limited number of insertions, deletions and substitutions. Pairwise alignment is typically done with some form of dynamic programming, i.e. Needleman–Wunsch [12] for the best overall alignment or Smith–Waterman [13] for the best local substring alignment. Both algorithms find an optimal match based on a given scoring scheme that rewards matches and penalizes mismatches, insertions and deletions. The algorithms operate by filling in an $n \times m$ scoring matrix based on strings of length $n$ and $m$ and compute the optimal score at each position with an overall sequential cost of $O(nm)$. The resulting dependence pattern leads to parallelism along an anti-diagonal wavefront. A popular heuristic algorithm, called X-drop [14], searches only for high-quality alignments by tracking the running highest score and not exploring cell neighbourhoods in the matrix whose score drops by a given threshold below the maximum. It gets its performance benefits from dynamically resizing the anti-diagonal wavefront (i.e. its *band*), therefore reducing the search space, and may stop early when there is no high-quality match.

Pairwise alignment appears throughout genomic data analysis, because both errors in data from sequencers and variations in genomes across individuals lead to imperfect string matches. The ExaBiome project has multiple instances of alignment, which include aligning short reads to partially assembled sequence data (called *contigs*), aligning long reads to each other, or aligning proteins to each other. Typical lengths of DNA from sequencers run from 100 to 250 characters for short reads to over 10 000 for long-read technology reads, while proteins are typically a few thousand characters long. Even if one is aligning against a full genome, e.g. the 3-billion-character reference human genome or a large database of genomes, it will be done by starting from a predetermined location or seed as described in the next section. At the scale of a few hundred to a few thousand characters, pairwise alignment is amenable to SIMD [15–17], multicore, GPU [18] and even FPGA [19,20] parallelism, and can take advantage of narrow data types to represent the four nucleotides in DNA, the 21 amino acids in proteins, or the limited range of values in the scoring matrix. Recent work also shows how dynamic programming problems exhibit essentially linear speedups using the concept of rank convergence [21], in which the pairwise alignment is computed via a series of dense matrix multiplications on the tropical semiring where the scalar addition is replaced with the maximum operator and scalar multiplication becomes integer addition.

Alignment dominates the local on-node computation in ExaBiome applications, as well as other genome analysis tools across scales. However, there is not sufficient work for distributed memory parallelism within pairwise alignment, and even GPU offload requires batch alignment, where a set of pairs are aligned as a single operation, to amortize the startup and data movement overhead.

## (c) All-to-all alignment

Alignment is often done across a set of strings, such as alignment against a database of reference genomes or proteins, a set of patient genomes against a single (large) reference, or a set of reads from a sequencer against each other or against partially constructed genomes fragments as part of genome assembly. The ExaBiome project performs all-to-all alignments as part of short read assembly (merAligner within MetaHipMer), in which case the input reads are aligned against all partially assembled contigs [22], and as the first step in long-read assembly where reads are aligned against each other in BELLA and diBELLA [9,23].

The all-to-all computational pattern is familiar from n-body simulations and, as in that case, computation on all $O(n^2)$ pairs of strings/particles is prohibitively expensive. To tackle this, particle simulations rely on hierarchical tree-based approaches that exploit the physical layout of particles in space, which is not applicable in alignment. Instead, in aligning a set of sequences, one can pre-filter the pairs to find ones that are likely to have a good alignment. Our approach therefore looks for sequences that share at least one short identical string, e.g. a k-mer, which can also be used to seed the alignment. For example, to align a set $S$ against another $T$, store all k-mers from strings in set $T$ in a hash table and lookup all the k-mers from strings in $S$ to find matching pairs, starting each pairwise alignment from the position of the common k-mer.

In distributed memory, the k-mer hash table has an irregular many-to-many communication pattern that is familiar from the k-mer counting, but each k-mer now retains the list of sequences containing that k-mer. The hash table may be viewed as a sparse k-mer$\times$sequence matrix with sequences from set $T$. To compute the set of sequences from S that have a matching k-mer, we can take either a linear algebra or database 'hash-join' view of the problem.

In the former case, we construct a k-mer×sequence matrix for each set, transpose one and multiply them to obtain a sparse sequence×sequence where each nonzero at position $i$,$j$ represents a pair $S_i$, $T_j$ that share a common k-mer. The sparse matrix primitive that performs this operation is known as SpGEMM, for Sparse GEneralized Matrix–Matrix multiplication [24]. It is generalized in the sense that the multiplication can operate on any arbitrary algebraic structure, also known as a semiring, and not just the real field. The single-node shared memory BELLA code uses this approach [23] to align a set of long reads to itself, so $S = T$. Both input and output matrices in BELLA's case are sparse.

The second approach constructs the same k-mer×sequence table for $T$ but does not explicitly compute the sequence×sequence matrix. Instead, as it computes the set of k-mers in $S$, it looks them up in $T's$ table to find sequences in $T$ with a common k-mer. The distributed memory diBELLA uses this approach in a bulk-synchronous series of many-to-many exchanges, while merAligner performs alignments on-the-fly as the read sequences are processed (typically fetching the contig from a remote processor) that contains a matching k-mer. merAligner also caches these contigs as there is enough likely reuse that can be leveraged to save repeated communication of contigs.

All of these distributed memory alignment algorithms involved irregular many-to-many communication either done asynchronously as 1-sided remote look-ups or in batches. The asynchronous approach has more messages, each of which is small, so communication software overhead and latency can limit performance. It has the advantage of overlapping computation and communication together, which makes good use of both networking and computing resources. The bulk-synchronous approach leads to better message aggregation between pairs of processors, but it can suffer from high load imbalance costs due to the implied barriers at each exchange. However, separating communication from computation prevents overlap and is more likely to trigger bisection bandwidth limits in the network. The pairwise alignments that follow communication can either be done one pair at a time or in batches, with the bulk-synchronous version likely having larger batches to do.

K-mer-based matching is not the only method that is used to index large genomic datasets. In particular, suffix trees and their more practical sibling suffix arrays provide an alternative way of indexing large datasets. Rather than hashing, these methods using sorting and search on a compact representation of the suffix substrings and then build a hierarchical index representation of the data. Suffix arrays are significantly more flexible than direct k-mer-based approaches because they effectively index all possible k-mer lengths at once. However, they are harder to implement and they often come with increased computational costs. Recent work on distributed suffix array construction [25] as well as querying [26] has shown scaling to eight nodes but with the potential to make these data structures more popular in HPC approaches.

There are other applications that arise in comparing which genomes or metagenomes align to each other. In this scenario, one is often interested in some sort of 'distance' metric between pairs of genomes or metagenomes, as opposed to merely identifying the candidate pairs that might align. The output is often dense because almost all pairs of (meta)genomes will contain conserved regions that will provide a match using shared k-mers. Using an approach similar to BELLA, Besta *et al.* [27] use parallel sparse matrix computations to compute the Jaccard similarity between all pairs of genomes. They also use the aforementioned SpGEMM primitive, with one difference that the software is optimized for the case where the output genomes×genomes matrix is dense, because it holds the Jaccard similarity.

The Bioinformatics community have been developing alternative space-efficient data structures in order to compute (meta)genome-to-(meta)genome distances for the scenario where a distributed-memory computer is unavailable. MASH [28], perhaps the most popular of such tools, uses the MinHash sketch technique [29] for each (meta)genome and only computes the Jaccard similarity on those sketches, as opposed to finding explicit shared k-mers. Recently, Baker & Langmead [30] took the sketching approach one step further and used the HyperLogLog

(HLL) algorithm for further compression. While we are not aware of any distributed-memory approaches to sketch-based genomic distance calculations, the HLL data structure itself is trivially mergeable. HLL has been used in distributed genome assembly for efficient k-mer counting in the past [5]. We therefore expect forthcoming developments in distributed-memory sketch-based genome comparison.

## (d) Graph traversal for genome assembly

Genome assembly involves the analysis of reads from sequencers to produce longer contiguous sequences of the genome with errors corrected. For short reads with their low error rate (less than 0.1%), the MetaHipMer software performs k-mer analysis and eliminates low-frequency k-mers which are presumably errors. Along with each k-mer in the final hash table, it stores left and right *high-quality* extensions, i.e. the character that frequently appeared to the left and right of the k-mer in the original input. This table is then viewed as a De Bruijn [31] graph in which a k-mer vertex is connected to another if their k-mers overlap in $k-1$ contiguous positions. The left and right extensions with each k-mer make it straightforward to find neighbouring vertices. A depth-first traversal starting from arbitrary k-mers compute the connected components of the graph which are linear sequences called *contigs*. For metagenomes, the same basic method is used but with increasing values of $k$, with contigs from the earlier steps added as reads to the later ones. This iterative process helps to improve coverage of low-depth, highly fragmented genomes in the earlier phases and resolve repeated regions and obtain longer contigs in the later phases. Once the contigs are formed, the assembler builds a graph with contig vertices and uses alignment to find reads that align to multiple contigs and thus form an edge in the contig graph. There are several other graphs traversals performed on both the k-mer and contig graph, which are omitted here. We focus on parallelization of contig construction on the k-mer hash table. More detailed descriptions of contig generation and other graph traversals during assembly are available in the HipMer and MetaHipMer papers [5–8,32].

MetaHipMer takes advantage of the memory and computing performance of distributed memory supercomputers to support large-scale assemblies. The hash tables involved in our algorithms can be up to tens of terabytes and do not fit in a typical shared memory node, and contig generation is written in UPC [33,34] so that hash table buckets are directly accessed by any processor using one-sided communication. During construction, we aggregate multiple insert operations intended for the same remote processor to amortize communication overhead. This is done dynamically and asynchronously: once a particular buffer for a remote node is full, it is sent using one-sided memory operations with atomics to the memory of a remote processor. Hash table inserts and lookups are done in two separate phases, so the delayed inserts from aggregation are not semantically visible—all of the inserts are complete at the end of the phase and the order is not important.

During graph traversal the hash table remains fixed, although multiple traversals happen in parallel from different starting vertices and individual k-mer vertices are marked as visited to avoid duplicate traversals. This is done with fine-grained remote atomics rather than locking to minimize the number of communication round trips, although this stage is latency-limited since each processor is performing a single-threaded traversal of the graph and needs to wait for a remote vertex before continuing. In later stages of assembly, the hash table of contigs is truly read-only and each contig may be used multiple times by a single processor, so caching remote contigs is efficient and preserves correctness. Caching is not performed during contig generation because there is limited reuse.

## (e) Sparse matrix operations for protein clustering

Proteins of the same evolutionary origin are said to be homologous. Homologous proteins often perform similar functions; hence homology finding facilitates protein annotation and the discovery of novel protein families. One often infers homology from excess sequence similarity;

with 'excess' referring to higher similarity than can be encountered by chance. Even then, a simple pairwise similarity metric is just a proxy for homology and can lead to both false positives and false negatives, depending on the parameters used in sequence similarity calculations. A clustering step that takes the similarity matrix as input and exploits topology information (i.e. the transitivity of neighbouring proteins) to find more robust and accurate protein families. This helps eliminate a significant portion of spurious homology connections and recovers many missing links while computing a globally consistent view of the clusters.

A typical pipeline for protein clustering therefore involves first finding highly similar sequences using many-to-many alignments among proteins, using one of the popular tools such as MMseqs2 [35] or LAST [36]. K-mer-based indexing that is similar in spirit to those described in §3c is often used to reduce the number of comparisons. The ExaBiome project is currently working on a novel many-to-many protein similarity search tool called Protein Sequence Aligner (PISA) that is scalable to Exascale architectures. The result of the similarity matrix/graph computation is then fed into a clustering algorithm that discovers the ultimate protein families. Since this two-step process is often very expensive, single-step clustering algorithms [37] have gained in popularity among those who do not have access to high-end computing equipment, despite often resulting in fragmented clusters. We will not be focusing on those methods here because one of the goals of the ExaBiome project is to improve accuracy by using Exascale computers.

The Markov Cluster (MCL) algorithm [38] is arguably the canonical graph-based algorithm for clustering protein similarity matrices. The MCL algorithm treats this similarity matrix as an adjacency matrix of the graph where vertices are proteins and edges are similarities. The graph is sparse because only those similarities that are above a certain similarity threshold are retained. MCL performs random walks from every vertex (protein) in the graph. It exploits the fact that most of these walks will be trapped within tightly connected clusters, hence driving up the probability mass that is accumulated within each cluster. In order to avoid densifying the intermediate matrices and making the computation infeasible, MCL performs various pruning strategies that are shown to not hurt the quality of the final clusters [39].

The simultaneous random walks directly map to a sparse matrix primitive that is commonly known as SpGEMM, which computes the product of two sparse matrices. The high-performance distributed re-implementation of the Markov Cluster algorithm, known as HipMCL [40], uses some of the most general and scalable sparse matrix algorithms implemented within the Combinatorial BLAS [41]. These algorithms include a two-dimensional SpGEMM algorithm known as Sparse SUMMA [24], several different shared memory SpGEMM algorithms [42] that are optimized for different iterations of HipMCL, a fast memory estimator based on sparse matrix dense matrix multiplication for memory-efficient SpGEMM [43], as well as a very fast distributed memory connected components algorithm [44] that is used for extracting the final clusters from the result of the HipMCL iterations. The integration of GPU support as well as faster communication-avoiding SpGEMM algorithms [45] is ongoing work.

## (f) Machine learning for genomics and proteomics

A comprehensive coverage of machine learning (ML) applications in genomics and proteomics is both too large and too fast growing to address here. Instead, we touch on the computational building blocks for the machine learning algorithms that are commonly applied to genomic and proteomic data. A large class of machine learning methods are built on top of basic linear algebraic subroutines that are found in the modern dense BLAS [46], Sparse BLAS [47], or the GraphBLAS [48]. This relationship is illustrated in figure 3.

Machine learning has been applied to metagenome assembly in various contexts. For example, MetaVelvet-SL [49] uses Support Vector Machines (SVMs) to identify the potentially chimaeric nodes on a metagenomic De Bruijn graph. A chimaeric node is shared by the genomes of two closely related species and needs to be split into multiple nodes for an accurate assembly. A popular application of ML to proteomics data is to discover ancestral relationships (i.e. homology) between proteins. Kernel-based methods, such as SVMs, have been traditionally

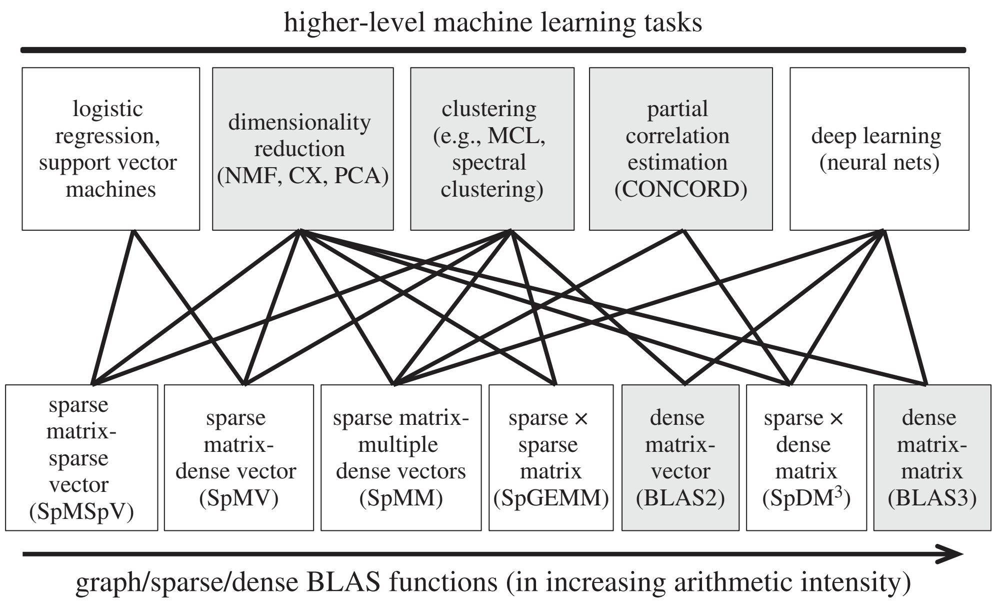
higher-level machine learning tasks
logistic regression, support vector machines
dimensionality reduction (NMF, CX, PCA)
clustering (e.g., MCL, spectral clustering)
partial correlation estimation (CONCORD)
deep learning (neural nets)
sparse matrix-sparse vector (SpMSpV)
sparse matrix-dense vector (SpMV)
sparse matrix-multiple dense vectors (SpMM)
sparse × sparse matrix (SpGEMM)
dense matrix-vector (BLAS2)
sparse × dense matrix (SpDM³)
dense matrix-matrix (BLAS3)
graph/sparse/dense BLAS functions (in increasing arithmetic intensity)

**Figure 3.** Dependencies of various machine learning methods upon linear algebraic primitives. The three grey boxes on the top are unsupervised methods while the two white boxes include supervised methods. Examples of algorithms in each group are in parentheses: non-negative matrix factorization (NMF), principal component analysis (PCA), and Markov cluster algorithm (MCL), convex correlation selection method (CONCORD), a low-rank matrix factorization (CX).

applied to this problem [50]. Other fundamental problems in this domain are protein folding [51], especially the prediction of the three-dimensional structure of the protein [52], and protein function prediction. The function of a protein can be predicted using either the sequence, the three-dimensional structure of the protein, or both [53].

## 4. Comparison with other parallelism motifs

Several computational patterns arise in the ExaBiome application and are common to other genomics applications and data analysis problems more broadly. These are displayed in figure 2 and include:

(i) *Hash tables*. These are used throughout the genome assembly applications, MetaHipMer and diBELLA, to store k-mers for the purposes of counting (histogramming), and for quickly finding pairs of sequences with a common substring.

(ii) *Sorting*. While used less frequently than hashing in our examples, it is another technique for counting k-mers and is used in suffix arrays and to prioritize graph operations, e.g. finding the longest contig as a starting point for a graph traversal.

(iii) *Graph traversals*. Used to connect k-mers into contigs and in other analyses on the contig graph to resolve ambiguities and increase assembly length.

(iv) *Alignment*. The problem of finding the minimum edits required to make two strings match is used on raw read data, assembled genomes, genes and proteins.

(v) *Generalized n-body*. The problem of comparing or aligning all sequences in one set to another set (or the same set), but using a method such as limiting to pairs with a common k-mer to avoid all $O(n^2)$ comparisons.

(vi) *Sparse matrices*. Sparse matrix products used within the generalized n-body problem to find pairs, for protein clustering, etc.

(vii) *Dense matrices*. There is ongoing work by ourselves and others to use machine learning methods in genomic data, which often make use of dense matrix multiplication as

**Table 1.** A comparison of motifs for parallel computing, including our own set for genomic data analysis.

| Colella 7 Dwarfs | Berkeley View Motifs | NRC 7 Giants | Genomics Motifs |
|---|---|---|---|
| dense matrix | dense matrix | dense and | dense matrix |
| sparse matrix | sparse matrix | . . .sparse matrix | sparse matrix |
| structured grid | structured grid | | |
| unstructured grid | unstructured grid | | |
| spectral methods | spectral methods | | |
| particle methods | N-body | Gen. N-body | Gen. N-body |
| Monte Carlo | MapReduce | basic statistics | basic operations[a] |
| | finite-state machine | | |
| | graph traversal | graph theoretic | graph traversal |
| | dynamic Prog. | alignment | alignment |
| | backtracking search | | |
| | graphical models | | |
| | combinatorial | | |
| | | optimization | |
| | | integration | |
| | | | hash tables |
| | | | sorting |

[a] Basic operations include string parsing, string identity and 2-bit encoding of DNA sequences.

described in §3f. Further, pairwise alignment can in theory also be computed using dense matrix computations on semirings.

In addition to these seven motifs of genomic data, local computations such as parsing reads into k-mers and other basic string operations, arithmetic operations, logical operations and more occur in all of our applications. When these can be performed independently on separate data, they can be invaluable in obtaining high-performance parallel implementations, but if the operations are each performed serially they are not instrumental in understanding parallelization. In comparing to other lists of motifs, we include these as 'Basic Operations' in table 1 although they tend to be linear time operations on the input which can be almost trivial to parallelize, and thus less useful as a parallelism motif.

While our selection of problems informing our genomics motifs is naturally biased, we note that independent HPC researchers have been focusing on similar problems. For example, Darwin [54] is a co-processor specifically designed to perform fast all-to-all long read overlapping and alignment in the context of assembly. The body of work from Aluru's group at Georgia Tech similarly encompasses k-mer analysis, alignment, assembly and clustering and uses some of the same patterns albeit pushing in the direction of bulk-synchronous computation [55]. SARVAVID [56] provides a Domain-Specific Language (DSL) with language constructs for k-mer extraction, index generation and look-up, clustering, all-to-all similarity computation, graph construction and traversal for genome assembly, and filtering error-prone reads. Mahadik *et al.* identify this list of 'kernels' as common to a broad variety of genomics applications. We remark that these kernels, with the possible exclusion of read filtering, can be mapped to our own list of motifs.

There are other proposals for the parallelism motifs that cover many applications of scientific simulations, data analysis and more. The original set of 'Seven Dwarfs' due to Phil Colella [57] was meant to capture the most important computational patterns in scientific simulations and are shown in the first column in table 1. The Berkeley View report [58] on multicore parallelism, in the second column, generalized these patterns to capture a broader set of applications including some data analysis problems. A report by the National Academies [59] then defined a set of 'Seven Giants' of Big Data, shown in the third column, which combined sparse and dense matrices into a single motif. Ogres [60] is another similar, yet multidimensional classification of both HPC and Big Data applications based on 51 well-studied NIST applications. Our own genomics motifs in the last column are quite similar to those in the 'Seven Giants' set, but in our view the ideas of hashing and sorting are so essential to understanding data analysis for genomic data and for other large-scale database analyses involving joins that they deserve to be separate categories. They are also standard in other large-scale database operations. On the other hand, optimization and integration are very general techniques that can lead to a variety of parallelism patterns depending on the data and method being used, e.g. they may be dominated by dense or sparse matrix operations, as well as other independent computations. Each list takes a somewhat different approach to characterizing independent operations, which in our view is such a general notion that it does not belong as an algorithmic motif. Colella's Monte Carlo class is a more specific class of problems that do lead to a style of parallelism, albeit dominated by independent calculations.

## 5. Hardware and software support for parallel genome analysis

Although some analysis problems can be done independently or with traditional bulk-synchronous parallelism, we argue that the irregular and asynchronous nature of some of these problems [7,61,62] places different requirements on the programming systems, libraries and network than most simulation problems. In addition, communication optimizations have a somewhat different characteristic than in more structured and regular computations.

Roughly speaking, there are four programming styles for distributed memory communication:

— Bulk-synchronous collectives, such as broadcast, reductions and all-to-all exchanges. For example, MPI collectives have a rich set of collective operations [63].
— Two-sided point-to-point communication, i.e. send and receive, which need not be synchronous, but requires two-sided coordination and is, therefore, often done in bulk-synchronous phases. MPI is again the standard here with various forms of send and receive.
— One-sided shared memory or Remote Data Memory Access, including put, get and atomic memory operations. There are several examples languages that support this style, including UPC used in the original MetaHipMer assembler [33].
— Remote Procedure Call (RPC), which invoke remote computation while communicating input and output arguments between processors. The most recent version of UPC++ provides a set of RPC features with asynchrony-by-default to encourage communication overlap [64].

The majority of simulation codes are written in some combination of the first two styles, while data analytics problems written in a map-reduce framework use collectives. But for analytics problems involving hash tables with random-in-time and random-in-location access, we argue that the latter two are a better fit. Sparse matrix computations such as iterative methods can be programmed elegantly using bulk-synchronous parallelism, as can sorting and generalized all-to-all problems, although the data exchanges are often irregular and unbalanced, with the volume of data between processors varying considerably. Communication imbalance issues can affect sparse matrix multiplication when the distribution of non-zeros is non-uniform, e.g. when a k-mer appears in many of the input sequences, or in parallel sorting when the distribution of values being sorted is nonuniform, e.g. a single value appears with very high frequency. These imbalance factors may encourage designs that avoid global communication and synchronization in favour of overlapped point-to-point or one-sided communication.

While numerical libraries form the basis of many computational simulations, we see distributed data abstractions for hash tables, Bloom filters, histograms and various types of queues for rebalancing data and computational load as keys to our analysis problems. For example, the Berkeley Container Library [65] provides the data structures and CombBLAS provides the distributed memory sparse matrix primitives designed for graph algorithms [41]. These libraries can capture some of the more important communication optimizations, which are familiar ideas but have somewhat different usage.

— Asynchronous communication avoids both global and pairwise synchronization, allowing each thread to progress without waiting to resolve load imbalance from communication or computation that may vary over time.
— Non-blocking communication provides overlap for both computation and other communication events, and is especially important for fine-grained communication to avoid paying full latency costs for each message. In a one-sided model, this means non-blocking put and get operations or fire-and-forget in an RPC model.
— Communication aggregation is a standard technique in bulk-synchronous applications, but in asynchronous ones this involved dynamic buffering of data destined for a single core or node and shipping it when the individual buffer is full or based on some other trigger. In practice, the management of the message buffers creates a critical trade-off between memory footprint and number of messages, but the uncertainty of communication volume and destination makes this particularly challenging.
— Improving spatial locality is not always possible for irregular data, e.g. hash table construction on unknown data, but when insight into the data is possible, a carefully constructed hash function can provide significant benefit in reducing the percentage of remote accesses [22].
— Caching remote data are useful when there is sufficient temporal locality, e.g. in looking up contigs during alignment of reads to contigs during assembly.
— Iteration space tiling used in communication-avoiding algorithms for dense matrix multiplication [66,67] and n-body calculations [68,69] provide provable advantages in reducing communication volume and number of messages at the cost of additional memory. These methods do not simply partition the result matrix or particles/sequences over processors, but instead replicate them to the extent allowed by available memory. For sparse matrices and sparse interactions, the benefits depend more on the sparsity patterns [24,42,43,70], but are useful in clustering [40] and possibly alignment.

From an architectural perspective, these highly irregular applications stress message injection rate, communication latency, and in some cases bisection bandwidth [7,61]. They may never saturate link bandwidth if a multi-core node cannot inject small messages into the network fast enough to saturate bandwidth. While message aggregation is used in our implementations to maximize bandwidth utilization, this tends to put significant pressure on the memory per node due to the nearly random pattern of remote processors with which a single node communicates. Communication overlap can also be critical in these applications, including overlapping multiple communication events with each other. Perhaps the most obvious difference between genome analysis and simulation is that floating point numbers are essentially non-existent in the lower level analyses and only arise in machine learning such as clustering and deep learning application.

## 6. Summary

This papers provides an overview of some of the computational patterns that arise in genomics data analysis using examples from the ExaBiome project. These represent problems like genome assembly and protein clustering that until recently were done only on shared memory machines. These can now be performed orders of magnitude faster and on datasets that were previously intractable, revealing new species and species families. We see a growing number of multi-terabyte datasets but also recognize that many biologists feel constrained in their experimental design by the daunting task of computational analysis. As the demand for better performance and larger datasets continues to grow, a distributed memory approach will be increasingly important.

Our goal in writing this paper is to summarize the work in high-performance data analysis for genomics to help experts outside biology understand the stress placed on parallel hardware and software systems from these applications. These patterns are captured in a set of motifs, closely related to the previous 'Seven Giants' of data analysis, but with the critical additions of hashing and sorting. We believe this list and the overview of application examples and parallelization techniques will help in designing benchmark suites, ensuring they capture some of the most important characteristics of this application space. The described methods can drive requirements analysis for hardware and software, representing problems with fine-grained, asynchronous, non-blocking, one-sided communication, irregular memory accesses and narrow data types for both integers and characters. Our experience also makes the case for reusable software libraries that go beyond algorithms to data structures that are distributed across processors but can be updated by a single process with limited synchronization.

Data accessibility. This article has no additional data.

Authors' contributions. M.A., A.A., B.B., R.E., S.E., M.E., E.G., G.G., S.H. and O.S. developed the parallel software described here. K.Y., A.B., C.T. and L.O. drafted the manuscript. All authors read and approved the manuscript.

Competing interests. The authors declare that they have no competing interests.

Funding. This paper summarizes work supported in part by the Department of Energy Office of Science under Lawrence Berkeley National Laboratory Contract DE-AC02-05CH11231 and DOE contract DE-SC0008700 at UC Berkeley; by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the US 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 no. 1823034 at UC Berkeley. This research used resources of the National Energy Research Scientific Computing Center (NERSC) under contract no. DE-AC02-05CH11231, and the Oak Ridge Leadership Computing Facility under DE-AC05-00OR22725. Both supported by the Office of Science of the US Department of Energy.

Disclaimer. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

## References

1. Zaharia M *et al.* 2016 Apache Spark: a unified engine for big data processing. *Commun. ACM* **59**, 56–65. (doi:10.1145/2934664)
2. Shiers J. 2007 The worldwide LHC computing grid (worldwide LCG). *Comput. Phys. Commun.* **177**, 219–223. (doi:10.1016/j.cpc.2007.02.021)
3. Messina P. 2017 The exascale computing project. *Comput. Sci. Eng.* **19**, 63–67. (doi:10.1109/MCSE.2017.57)
4. Manekar SC, Sathe SR. 2018 A benchmark study of K-mer counting methods for high-throughput sequencing. *GigaScience* **7**, giy125. (doi:10.1093/gigascience/giy125)
5. Georganas E, Buluç A, Chapman J, Oliker L, Rokhsar D, Yelick K. 2014 Parallel De Bruijn graph construction and traversal for de novo genome assembly. In *SC'14: Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, 16–21 November 2014*, pp. 437–448. New York, NY: ACM and IEEE.
6. Georganas E, Buluç A, Chapman J, Hofmeyr S, Aluru C, Egan R, Oliker L, Rokhsar D, Yelick K. 2015 HipMer: an extreme-scale de novo genome assembler. In *SC'15: Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, Austin, TX, November 15–20, 2015*, pp. 1–14. New York, NY: ACM and IEEE.
7. Georganas E. 2016 Scalable parallel algorithms for genome analysis. Berkeley, CA: University of California at Berkeley.

8. Georganas E, Egan R, Hofmeyr S, Goltsman E, Arndt B, Tritt A, Buluç A, Oliker L, Yelick K. 2018 Extreme scale de novo metagenome assembly. In *SC18: Int. Conf. for High Performance Computing, Networking, Storage and Analysis, Austin, TX, November 11–16, 2018*, pp. 122–134. New York, NY: ACM and IEEE.
9. Ellis M, Guidi G, Buluç A, Oliker L, Yelick K. 2019 diBELLA: distributed long read to long read alignment. In *48th Int. Conf. on Parallel Processing (ICPP), Kyoto, Japan, August 5–8, 2019*, pp. 70:1–70:11. New York, NY: ACM.
10. Gao T, Guo Y, Wei Y, Wang B, Lu Y, Cicotti P, Balaji P, Taufer M. 2017 Bloomfish: a highly scalable distributed K-mer counting framework. In *2017 IEEE 23rd Int. Conf. on Parallel and Distributed Systems (ICPADS), Shenzhen, China, December 10–16, 2017*, pp. 170–179. New York, NY: IEEE.
11. Pan T, Flick P, Jain C, Liu Y, Aluru S. 2017 Kmerind: a flexible parallel library for k-mer indexing of biological sequences on distributed memory systems. *IEEE/ACM Trans. Comput. Biol. Bioinf.* **16**, 1117–1131.
12. Needleman SB, Wunsch CD. 1970 A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* **48**, 443–453. (doi:10.1016/0022-2836(70)90057-4)
13. Smith TF, Waterman MS. 1981 Identification of common molecular subsequences. *J. Mol. Biol.* **147**, 195–197. (doi:10.1016/0022-2836(81)90087-5)
14. Zhang Z, Schwartz S, Wagner L, Miller W. 2000 A greedy algorithm for aligning DNA sequences. *J. Comput. Biol.* **7**, 203–214. (doi:10.1089/10665270050081478)
15. Farrar M. 2006 Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* **23**, 156–161. (doi:10.1093/bioinformatics/btl582)
16. Li H. 2018 Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* **34**, 3094–3100.
17. Suzuki H, Kasahara M. 2018 Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC bioinf.* **19**, 45.
18. Liu Y, Wirawan A, Schmidt B. 2013 CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinf.* **14**, 117. (doi:10.1186/1471-2105-14-117)
19. Di Tucci L, O'Brien K, Blott M, Santambrogio MD. 2017 Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, Lausanne, Switzerland, March 27–31, 2017*, pp. 716–721. Leuven, Belgium: European Design and Automation Association.
20. Li IT, Shum W, Truong K. 2007 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). *BMC Bioinf.* **8**, 185.
21. Maleki S, Musuvathi M, Mytkowicz T. 2016 Efficient parallelization using rank convergence in dynamic programming algorithms. *Commun. ACM* **59**, 85–92. (doi:10.1145/2983553)
22. Georganas E, Buluç A, Chapman J, Oliker L, Rokhsar D, Yelick K. 2015 MerAligner: a fully parallel sequence aligner. In *2015 IEEE Int. Parallel and Distributed Processing Symposium, Hyderabad, India, December 19–23, 2015*, pp. 561–570. New York, NY: IEEE.
23. Guidi G, Ellis M, Rokhsar D, Yelick K, Buluç A. 2018 BELLA: Berkeley efficient long-read to long-read aligner and overlapper. Preprint p. *bioRxiv* 464420.
24. Buluç A, Gilbert JR. 2012 Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM J. Sci. Comput.* **34**, C170–C191. (doi:10.1137/110848244)
25. Flick P, Aluru S. 2015 Parallel distributed memory construction of suffix and longest common prefix arrays. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, November 15–20, 2015*, p. 16. New York, NY: ACM.
26. Flick P, Aluru S. 2019 Distributed enhanced suffix arrays: efficient algorithms for construction and querying. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, Denver, CO, November 17–22, 2019*, p. 72. New York, NY: ACM.
27. Besta M, Kanakagiri R, Mustafa H, Karasikov M, Rätsch G, Hoefler T, Solomonik E. 2019 Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons. (http://arxiv.org/abs/191104200).

28. Ondov BD, Treangen TJ, Melsted P, Mallonee AB, Bergman NH, Koren S, Phillippy AM. 2016 Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biol.* **17**, 132. (doi:10.1186/s13059-016-0997-x)
29. Indyk P, Motwani R. 1998 Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. thirtieth annual ACM symposium on Theory of computing*, pp. 604–613. ACM.
30. Baker DN, Langmead B. 2019 Dashing: fast and accurate genomic distances with HyperLogLog. *Genome Biol.* **20**, 265. (doi:10.1186/s13059-019-1875-0)
31. De Bruijn NG. 1946 A combinatorial problem. In *Proc. Koninklijke Nederlandse Academie van Wetenschappen*. vol. 49, Amsterdam, The Netherlands, June 29, 1946, pp. 758–764. Eindhoven, The Netherlands: Eindhoven University of Technology.
32. Georganas E, Hofmeyr S, Oliker L, Egan R, Rokhsar D, Buluc A, Yelick K. 2017 Extreme-scale de novo genome assembly. In *Exascale scientific applications: scalability and performance portability* (eds T Straatsma, K Antypas, T Williams), ch. 18, p. 409. Boca Raton, FL: CRC Press.
33. Carlson WW, Draper JM, Culler DE, Yelick K, Brooks E, Warren K. 1999 Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences.
34. UPC Consortium and others. 2005 Unified Parallel C language spec. v1.3. Technical Report.
35. Steinegger M, Söding J. 2017 MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets. *Nat. Biotechnol.* **35**, 1026. (doi:10.1038/nbt.3988)
36. Kiełbasa SM, Wan R, Sato K, Horton P, Frith MC. 2011 Adaptive seeds tame genomic sequence comparison. *Genome Res.* **21**, 487–493. (doi:10.1101/gr.113985.110)
37. Li W, Godzik A. 2006 Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics* **22**, 1658–1659. (doi:10.1093/bioinformatics/btl158)
38. Enright AJ, Van Dongen S, Ouzounis CA. 2002 An efficient algorithm for large-scale detection of protein families. *Nucleic Acids Res.* **30**, 1575–1584. (doi:10.1093/nar/30.7.1575)
39. Van Dongen S. 2008 Graph clustering via a discrete uncoupling process. *SIAM J. Matrix Anal. Appl.* **30**, 121–141. (doi:10.1137/040608635)
40. Azad A, Pavlopoulos GA, Ouzounis CA, Kyrpides NC, Buluç A. 2018 HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Res.* **46**, e33. (doi:10.1093/nar/gkx1313)
41. Buluç A, Gilbert JR. 2011 The combinatorial BLAS: design, implementation, and applications. *Int. J. High Perform. Comput. Appl.* **25**, 496–509. (doi:10.1177/1094342011403516)
42. Nagasaka Y, Matsuoka S, Azad A, Buluç A. 2019 Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Comput.* **90**, 102545. (doi:10.1016/j.parco.2019.102545)
43. Cohen E. 1998 Structure prediction and computation of sparse matrix products. *J. Comb. Optim.* **2**, 307–332. (doi:10.1023/A:1009716300509)
44. Azad A, Buluç A. 2019 LACC: a linear-algebraic algorithm for finding connected components in distributed memory. In *2019 IEEE Int. Parallel and Distributed Processing Symp., IPDPS 2019, Rio de Janeiro, Brazil, 20–24 May 2019*, pp. 2–12. New York, NY: IEEE.
45. Azad A, Ballard G, Buluc A, Demmel J, Grigori L, Schwartz O, Toledo S, Williams S. 2016 Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM J. Sci. Comput.* **38**, C624–C651. (doi:10.1137/15M104253X)
46. Dongarra JJ, Cruz JD, Hammarling S, Duff IS. 1990 Algorithm 679: a set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw. (TOMS)* **16**, 18–28. (doi:10.1145/77626.77627)
47. Duff IS, Heroux MA, Pozo R. 2002 An overview of the sparse basic linear algebra subprograms: the new standard from the BLAS technical forum. *ACM Trans. Math. Softw. (TOMS)* **28**, 239–267. (doi:10.1145/567806.567810)
48. Buluç A, Mattson T, McMillan S, Moreira J, Yang C. 2017 Design of the GraphBLAS API for C. In *IPDPS Workshops, Orlando, FL, May 29–June 2, 2017*, pp. 643–652. New York, NY: IEEE.
49. Sato K, Sakakibara Y. 2014 MetaVelvet-SL: an extension of the Velvet assembler to a de novo metagenomic assembler utilizing supervised learning. *DNA Res.* **22**, 69–77. (doi:10.1093/dnares/dsu041)
50. Rangwala H, Karypis G. 2005 Profile-based direct kernels for remote homology detection and fold recognition. *Bioinformatics* **21**, 4239–4247. (doi:10.1093/bioinformatics/bti687)

51. Dill KA, Ozkan SB, Shell MS, Weikl TR. 2008 The protein folding problem. *Annu. Rev. Biophys.* **37**, 289–316. (doi:10.1146/annurev.biophys.37.092707.153558)
52. AlQuraishi M. 2019 End-to-end differentiable learning of protein structure. *Cell Syst.* **8**, 292–301. (doi:10.1016/j.cels.2019.03.006)
53. Gligorijevic V *et al.* 2019 Structure-based function prediction using graph convolutional networks. Preprint p. *bioRxiv* 786236. (doi:10.1101/786236)
54. Turakhia Y, Bejerano G, Dally WJ. 2018 Darwin: a genomics co-processor provides up to 15 000 x acceleration on long read assembly. In *ACM SIGPLAN Notices*, vol. 53, pp. 199–213. ACM.
55. Aluru S. 2016 Genomes Galore: Big Data Challenges in the Life Sciences. In *2016 IEEE 23rd Int. Conf. on High Performance Computing (HiPC), Hyderabad, India, December 19–22, 2016*, pp. 1–1. New York, NY: IEEE.
56. Mahadik K, Wright C, Zhang J, Kulkarni M, Bagchi S, Chaterji S. 2016 SARVAVID: a domain specific language for developing scalable computational genomics applications. In *Proc. 2016 Int. Conf. on Supercomputing, Istanbul, Turkey, June 1–3, 2015*, p. 34. New York, NY: ACM.
57. Colella P. 2004 Defining software requirements for scientific computing. Presentation at the DARPA High Productivity Computing Program Meeting, Fairfax, VA.
58. Asanovic K *et al.* 2006 The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.
59. National Research Council2013 *Frontiers in massive data analysis*. New York, NY: National Academies Press.
60. Fox GC, Jha S, Qiu J, Ekanayake S, Luckow A. 2015 Towards a comprehensive set of big data benchmarks. *Big Data High Perform. Comput.* **26**, 47.
61. Georganas E, Ellis M, Egan R, Hofmeyr S, Buluç A, Cook B, Oliker L, Yelick K. 2017 MerBench: PGAS benchmarks for high performance genome assembly. In *Proc. Second Annual PGAS Applications Workshop, Denver, CO, November 17, 2018*, p. 5. New York, NY: ACM and IEEE.
62. Ellis M, Georganas E, Egan R, Hofmeyr S, Buluç A, Cook B, Oliker L, Yelick K. 2017 Performance characterization of de novo genome assembly on leading parallel systems. In *European Conf. on Parallel Processing*, pp. 79–91. Berlin, Germany: Springer.
63. Gropp W, Gropp WD, Lusk ADFEE, Lusk E, Skjellum A. 1999 *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. Cambridge, MA: MIT Press.
64. Bachan J, Bonachea D, Hargrove PH, Hofmeyr S, Jacquelin M, Kamil A, van Straalen B, Baden SB. 2017 The UPC++ PGAS library for exascale computing. In *Proc. Second Annual PGAS Applications Workshop*, p. 7. ACM.
65. Brock B, Buluç A, Yelick K. 2019 BCL: A Cross-Platform Distributed Data Structure Library Library. In *48th Int. Conf. on Parallel Processing (ICPP), Kyoto, Japan, August 5–8, 2019*, pp. 102:1–102:10. New York, NY: ACM.
66. Agarwal RC, Balle SM, Gustavson FG, Joshi M, Palkar P. 1995 A three-dimensional approach to parallel matrix multiplication. *IBM J. Res. Dev.* **39**, 575–582. (doi:10.1147/rd.395.0575)
67. Solomonik E, Demmel J. 2011 Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms. In *European Conf. on Parallel Processing*, pp. 90–109. Berlin, Germany: Springer.
68. Hendrickson B, Plimpton S. 1995 Parallel many-body simulations without all-to-all communication. *J. Parallel Distrib. Comput.* **27**, 15–25. (doi:10.1006/jpdc.1995.1068)
69. Driscoll M, Georganas E, Koanantakool P, Solomonik E, Yelick K. 2013 A communication-optimal n-body algorithm for direct interactions. In *2013 IEEE 27th Int. Symp. on Parallel and Distributed Processing, Cambridge, MA, 20–14 May 2013*, pp. 1075–1084. New York, NY: IEEE.
70. Ballard G, Buluc A, Demmel J, Grigori L, Lipshitz B, Schwartz O, Toledo S. 2013 Communication optimal parallel multiplication of sparse random matrices. In *Proc. twenty-fifth annual ACM Symp. on Parallelism in algorithms and architectures, Montreal, Canada, July 23–25, 2013*, pp. 222–231. New York, NY: ACM.