miniGiraffe: A Pangenomic Mapping Proxy App

Jessica I. Dagostini*, Joseph B. Manzano†, Tyler Sorensen‡* and Scott Beamer*
{jessica.dagostini, tyler.sorensen, sbeamer}@ucsc.edu, joseph.manzano@pnnl.gov
*University of California, Santa Cruz, Santa Cruz, CA
†Pacific Northwest National Laboratory, Richland, Washington
‡Microsoft Research, Redmond, Washington

***Abstract*—Large, real-world scientific applications are often complex, making them difficult to analyze, characterize, and optimize. Such applications typically involve intricate I/O patterns and library dependencies, which can make workflow analysis and tuning difficult. Proxy applications offer a practical solution by emulating the essential characteristics of the original application while significantly reducing complexity.**

In this work, we present miniGiraffe, a proxy application for Giraffe, a sophisticated genomics tool that operates over a pangenome, a graph-based structure capturing genetic variation across a species. We develop miniGiraffe using a principled methodology: carefully characterizing Giraffe's behavior and validating that our proxy faithfully reproduces its key computational features. miniGiraffe contains only 2% of Giraffe's codebase, while producing identical outputs for the most computationally intensive code components and closely matches Giraffe's execution time and scaling behavior in these regions. The simplified design of miniGiraffe enabled rapid experimentation across multiple architectures, which we utilize to perform an autotuning experiment of the mapping workflow; we found that specializing parameters to inputs and architectures provided a geometric mean speedup of $1.15\times$ and up to a $3.32\times$ speedup over the default parameters.

***Index Terms*—proxy applications, pangenomes, genomics, mapping**

I. Introduction

Genomics applications are increasingly utilizing large-scale compute platforms as data availability and computing power is increasing. This is especially true for recent genomics innovations, such as the pangenome, which enables the simultaneous analysis of multiple genomes. These applications are often computationally intensive, I/O-heavy, and built on complex software stacks, making them difficult to analyze, characterize, and optimize. Proxy applications offer a practical way to address this complexity by replicating the core behavior of the original application in a more lightweight and tractable form.

As the size and complexity of pangenomes grow, so do the computational demands of applications that operate on them. This is especially true for the human pangenome, which accurately captures critical genetic variations; its first draft was only released in 2021 [20]. Mapping against such complex reference structures can be extremely resource-intensive—depending on the software used, and pangenome mapping can take over 40 hours on large compute nodes even for relatively small datasets [31]. Given the scientific and medical importance of pangenomes, along with the novelty of their computational methods, we believe they present an especially compelling opportunity for optimization by the computational research community.

TABLE I
Comparing Giraffe vs. miniGiraffe code

Giraffe	miniGiraffe
∼50k lines of code	∼1k lines of code
∼350 source-files	2 source files
∼50 library dependencies	3 library dependencies

a) The Complexities of Scientific Applications: As is the case with many scientific applications, pangenome mapping applications are complex, consisting of large code bases with many library dependencies. Even profiling the code to determine performance-critical regions is difficult, let alone developing and evaluating optimizations.

To manage this complexity, *proxy applications* (often referred to as *proxies* or *miniapps*) are commonly employed. These are simplified versions of full applications that retain key characteristics such as computational kernels, memory behavior, or communication patterns; and as such, these proxies can be used to understand system behavior and, in turn, develop optimization strategies [1]. For example, proxies have been used to improve the energy efficiency of large physics simulations [34] and to evaluate different programming models in shock hydrodynamics applications [14].

While benchmark suites for bioinformatics exist [2], these simplified kernels do not capture the complexity of large, real-world applications, such as pangenome mapping. As such, insights from these benchmarks may not map back to the large application. To the best of our knowledge, no proxy application for this emerging pangenome computation currently exists, which is precisely this work's contribution.

b) Shrinking a Giraffe: The complex pangenome mapping application we consider is *Giraffe* [31]. This application uses *variation graphs* [8], a bioinformatics data structure to represent linear genome data and their respective variations. Giraffe is part of the larger toolkit *VG*, which contains several other applications that utilize variation graphs, and its mappings perform many traversals over the graph. These tools all have complex inter-dependencies, making focused analysis and optimization difficult.

This work presents *miniGiraffe*, a proxy application for Giraffe. The development of this proxy began with a detailed performance analysis of the original application to identify the

most *critical functions*, i.e., functions in the mapping procedure that take the most amount of time. To do this, we combined standard profiling tools with custom instrumentation. We identified critical functions that, on average, accounted for 32% of the total runtime across a diverse set of datasets. Our proxy app, miniGiraffe, was built to encapsulate these critical functions, with I/O obtained by capturing the I/O to those critical functions in Giraffe.

We validate miniGiraffe both functionally and in terms of computational characteristics. Functionally, we demonstrate miniGiraffe outputs exactly the same results as Giraffe. For computational characteristics, we collect a series of performance data, including overall runtime, parallel scaling, and hardware performance counters, and show a close correspondence to Giraffe. Specifically, we show that the execution time of our proxy is within 8.7% of the execution time of Giraffe's critical functions.

c) Tiny but Mighty: A key component of a proxy is its simplicity compared to the original application (Table I). miniGiraffe has only 1K lines of code (LoC) across two source files compared to the 50K LoC across 350 source files in the original application. Furthermore, miniGiraffe uses only three external libraries, while the original uses around 50. As a result, miniGiraffe is substantially more straightforward to compile, requiring a few lines in a Makefile, while the original application contains a labyrinth of Makefiles and dependencies.

Utilizing these simplifications, it becomes feasible to explore the behavior of the application across different architectures and find optimizations. In Section VII, we show the results of running miniGiraffe across four different systems and the impact their hardware characteristics have on performance. Furthermore, we also explored other performance parameters, such as parallel scheduling policies, batch size, and software caching capacity. miniGiraffe enables exhaustive exploration of these parameters, and the results show that specialized parameters (i.e., through auto-tuning) is able to achieve up to a 3.32× speedup and a geometric mean speedup of 1.15×.

d) Contributions: In summary, our contributions are:

- A workload characterization of the Giraffe pangenome mapping tool (Section IV).
- *miniGiraffe*, a proxy which captures the key computational components of Giraffe (Section V).
- An evaluation of miniGiraffe across different systems with different hardware configurations (Section VII).
- Auto tuning performance parameters across inputs and systems, achieving speedups of up to 3.32× (Section VII).

We released the code for miniGiraffe as open-source at https://github.com/jessdagostini/miniGiraffe to help computational researchers explore optimization strategies on this type of workload.

II. Background

A. Mapping, Variation Graphs, and Pangenomes

DNA is composed of four types of bases (ACTG) that are concatenated to form a long sequence. To read in a DNA sample, it is replicated and broken into small sequences called *reads*. These reads can be used to construct the DNA sequence either from scratch (de novo assembly) or mapped against an existing reference (alignment) [33]. Classically, reads have been aligned against a single reference, but it has been shown to be useful to focus analysis on the small variations, which can be encoded in a *variation graph* (VG) [8]. Within a VG, a path represents a sequence, branches represent variations, and merges represent commonalities (Figure 1). The *VG toolkit* is a framework that provides efficient VG data structures (which have been shown to scale to GBs) [8]. VG is implemented with C++ and OpenMP, and it also provides a library interface for external applications.

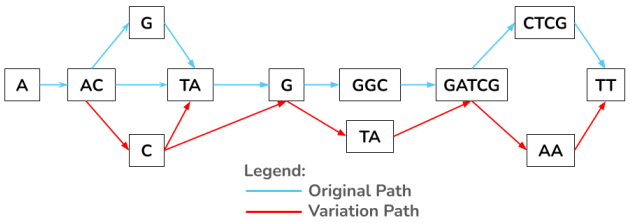


Fig. 1. Example variation graph. Each node in the graph holds a common sequence in the genome reference. The blue vertices represent the original single reference, and the red vertices are the variations included [3].

VGs enable a more flexible reference that can incorporate variations to avoid biases in analyses [29]. These references are called *pangenomes*, which aim to represent the entire set of genes within a population. The main difference between pangenomes and single-reference variation graphs lies in how the variances are captured. Single reference variation graphs anchor all genomic variations to a single linear reference genome. On a pangenome, the cross-reference between hundreds (and, in the future, thousands) of different DNA sequences is stored in the variation graph. Pangenomes can be used at many scales, whether it be a population, species, or even a metagenome [6], [24]. For example, analyzing a human pangenome allows for the discovery of variants that can be specifically associated with specific phenotypes but are missing from a single reference genome [24]. In a pangenomic setting, one can attempt to model the relationships between all of the genomes in an analysis instead of an arbitrarily chosen reference [6].

B. Giraffe Mapping

Changing paradigms from a single linear reference to a pangenome graph brings new computational challenges. A pangenome represented as a VG requires more memory as it essentially holds multiple references. Furthermore, mapping to a pangenome requires exploring many more combinations and thus, a corresponding increase in computational expense. Giraffe [31] is a short-read (sequences between 50 to 300 base pairs) mapper for pangenomes, with a performance goal to map reads at a speed similar to conventional (single-reference) sequence alignment tools.

Giraffe uses a seed-and-extend approach to map reads to the reference. In this approach, many candidate short seed matches are evaluated, and top candidates are then extended further [31]. Giraffe takes reads to map and the variation graph reference as inputs. These reads can be single or paired-ended, meaning that the DNA was sequenced from only one end (single) or from both ends (paired) of the fragment [11]. Giraffe uses three indices to seed the matching process: a Graph Positional Burrows-Wheeler Transform [22], a minimizer, and a distance index.

a) Graph Burrows-Wheeler Transform (GBWT): The Burrows-Wheeler Transformation (BWT) permutes an input string to make it more compressible [22]. Extending this approach to graphs yields the *Graph BWT* (GBWT). This approach stores *haplotypes*, groups of genomic variants that tend to be inherited together [12], as paths in the graph, enabling efficient storage and queries. GBWT further utilizes a Full-text Minute-space (FM) index approach, which is a text index based on the BWT [7]. This is the main data structure that holds the pangenome graphs for the Giraffe mapper.

The GBWT is stored in a compressed file (GBZ [30]), which is decompressed at runtime. Giraffe implements a "cached" version of the GBWT object, in which visited nodes are kept decompressed in memory. This allows for more efficient repeated accesses to the same subsets of nodes.

b) Minimizers: Minimizers are an auxiliary index created based on the reference. They utilize a *k-mer* (all possible substrings of length *k*), its length, a window length, and an order on the k-mers [35]. This strategy is used to decrease memory usage in the mapping process and improve computational efficiency. On Giraffe, a matching minimizer between the read and the GBWT is called a *seed*.

c) Distance Index: The distance index maps the minimum graph distance between seeds and clusters to improve speed. The extension will happen within minimizers in high-scoring clusters, where they are extended linearly to form maximal gapless local alignments. Giraffe will try to extend seed alignments in both directions [31].

C. Proxy Applications

When full applications become too complex to analyze and optimize easily, a *proxy application* can be useful. The representativeness of a proxy's computational characteristics is essential if the proxy is to be used to understand the larger application and propose strategies for optimizing it [1]. The goal of a proxy application is to represent the original application workflow without simply repeating it. Recreating all of the original application will naturally require a similar amount of complexity, so to make the proxy simpler, it must judiciously make simplifications and omissions. On the other hand, a benchmark standardizes evaluations, easing comparisons. Many benchmarks include a reference implementation, which is a simple implementation of the benchmark. A reference implementation can be similar to a proxy application. Still, the key difference is that a proxy application is designed to have the same key workload characteristics as the original applications, while a reference implementation is intended only to be functionally correct according to the benchmark.

The methodology for building a proxy can vary, and there is no one-size-fits-all solution. The literature contains diverse accepted methodologies for designing proxies, ranging from extracting key kernels from existing applications [19], developing the proxy from scratch based on known important kernels [28], or even implementing application patterns without being functionally equivalent [9]. Despite different ways to achieve the results, every proxy application needs to match the primary computational bottlenecks of its parent applications/workloads, but they may not necessarily perfectly replicate all metrics [9], [28].

Furthermore, a proxy must be versatile and more portable than the original application. That portability makes it suitable to test the workload on different platforms, using different strategies and different technologies [23]. This can be a difficult task, as some scientific applications contain some rigid functional requirements that are sometimes crucial for correct execution that their proxy versions must address. Nevertheless, deep workflow characterizations and understanding of the target application, as one would perform to create a proxy application, can also help identify the main algorithms and make them portable [23].

III. Methodology

Our proxification consists of an iterative methodology of profiling, analysis, inspection, and modification. Our initial profiling efforts used standard tools (Linux perf & Intel VTune). However, to better customize our profiling, we resorted to manual instrumentation of the Giraffe mapping tool, and we developed a C++ header to capture those profiling results with low overhead. This header captures timestamps from designated regions and stores them in a UThash hash table [10]. Our header dumps all of the profiling data at the end of execution to avoid introducing overhead during execution. We subsequently analyze the gathered data to derive insights into the application's performance. To ensure we are not

TABLE II
Hardware platforms used for evaluation

	local-intel	**local-amd**	**chi-arm**	**chi-intel**
Vendor	Intel	AMD	Cavium	Intel
Processor	Xeon 8260	EPYC 9554	ThunderX2 99xx	Xeon 8380
Sockets	2	1	2	2
Frequency	2.4 GHz	3.1 GHz	2.5 GHz	2.3 GHz
Cores / socket	24	64	32	40
L3 cache / socket	35.75 MB	256 MB	64 MB	60 MB
L2 cache / core	1 MB	1 MB	256 KB	1.25 MB
L1I cache / core	32 KB	32 KB	32 KB	32 KB
L1D cache / core	32 KB	32 KB	32 KB	48 KB
Threads / core	2	2	1	2
DRAM	768 GB	768 GB	256 GB	256 GB
OS	Ubuntu 22.04.4	Debian 12	Ubuntu 22.04.5	Ubuntu 24.04.2
Compiler	gcc 11.4.0	gcc 12.2.0	gcc 11.4.0	gcc 13.3.0

TABLE III
INPUT SETS COMBINING SHORT-READ AND PANGENOME REFERENCES FOR GIRAFFE WORKFLOW CHARACTERIZATION. THE D1 SHORT READS HAVE TWO INPUTS, DIFFERING ONLY BY THE NUMBERS IN THE BRACES ({}).

Input Sets	Workflow	Short Reads	Reads (M)	File Size (GB)	Pangenome Reference	Compressed File Size (GB)
A-human	Single	novaseq6000-ERR3239454	1.0	0.6	1000GPlons_hs38d1_filter	18.0
B-yeast	Single	SRR4074257	24.5	2.5	yeast_all	0.1
C-HPRC	Paired	D1_S1_L001_R{1,2}_004	8.0	1.6	hprc-v1.1-mc-grch38	3.1
D-HPRC	Paired	D1_S1_L002_R{1,2}_001	71.1	13	hprc-v1.0-mc-chm13	3.4

being biased by leftovers from previous similar executions, we also use an approach that creates factorial repetitions for our experiments. Guided by our analysis, we investigated the code to gather knowledge about the key aspects of our application. We iterated through this methodology several times. Eventually, after a few iterations, we implemented and revised our proxy based on our analysis and insights.

We performed our experiments on four servers (Table II). Two of them (`chi-arm` and `chi-intel`) are from the Chameleon Cloud testbed [15].

We used 4 different input sets for the workload characterization (Table III). Input set A-human uses the 1000GPlons pangenome created by the VG team using variants from the 1000 Genome Project [5]. It maps a single-end read input from the NA19239 individual [27]. Input set B-yeast also explores the single-end workflow by using a yeast pangenome based on a full set of yeast samples, available on UCSC Genome Browser [16]. Input sets C-HPRC and D-HPRC explore the paired-end workflow. Input set C-HPRC uses the latest version of the human pangenome graph built using variants from the Genome Reference Consortium Human Build 38 [25]. Input set D-HPRC uses the latest pangenome built from the complete human genome sequence CHM13 [26]. Both pangenomes are publicly available in the Human Pangenome Project [20]. For both paired-end input sets, data comes from different fragments of the genome sequence of the NA24385 individual's son. The data collected comes from VG toolkit on version v1.53.0 "Valmontone".

IV. UNDERSTANDING GIRAFFE

In this section, we provide an overview of the Giraffe short-read pangenome mapping tool that is the target of this work [31]. We dive deep into its functionalities and its code structure, guided by an insightful workload characterization that utilizes various inputs.

A. Giraffe Workload Characterization

VG toolkit's source code is complex and divided into modules and libraries to support the sophisticated set of tools it provides for variation graph construction and analysis. Understanding Giraffe's mapping call graph required significant effort because of its complex sequence of calls for different functions is spread across different libraries. That complexity obscured the data of interest when using existing third-party instrumentation tools. For our investigations, we manually instrumented the application using timestamp collectors, creating regions named according to the function/process being executed. Given all of the dependencies and complexities to compile Giraffe, we only utilize the local-intel machine for these analyses. We run them many times using all the inputs described in Section III.

From our instrumentation of a mapping execution, we observe that all threads run all instrumented functions, and most of them have short execution times but are frequently repeated (Figure 2). Through the zoomed-in portion of the plot, we noticed that Thread 0 only started working a half-second after the others. This pattern could be observed in all input sets run in both single and paired workflows, and is probably due to executing code that is not instrumented. Moreover, closer to the two-second mark, two occurrences of *process_until_threshold_c* and *cluster_seeds* regions take longer to finalize, indicating they can represent a significant portion of the process.

Further investigation over the application's code shows that this behavior is due to the way the application spreads work across threads. VG launches batches of parallel tasks using its own wrappers around OpenMP pragmas. The application uses C++ lambda functions to declare and include functions as parameters to other function calls. These mapping lambdas are buffered and scheduled by Giraffe's main scheduler. This task-scheduler code creates batches of short reads and assigns them to threads. The scheduler (running on the main thread) keeps track of how many threads are busy, and if no more processing resources are available, it processes any queued batches of reads left.

To obtain a better sense of which role each instrumented region plays in the total execution time of the mapping process, we aggregated the execution time collected for each region in a single runtime per region for the whole execution of each input set. Figure 3 depicts the average percentage of the total execution time each instrumented region occupies. We averaged the results across threads, and our experiments found that threads typically had similar percentages. We have instrumented most of the code, and in this figure, we removed time spent on I/O and parsing input settings to focus on the core of the workload.

The input set A-human spends a larger portion of its execution time on IO and parsing since it has fewer reads. Even with these characteristics, the A-human input set also behaves similarly to the others. We observe that the *process_until_thereshold_c* region was the most time-consuming in all executions, varying from 7% to 32% on input

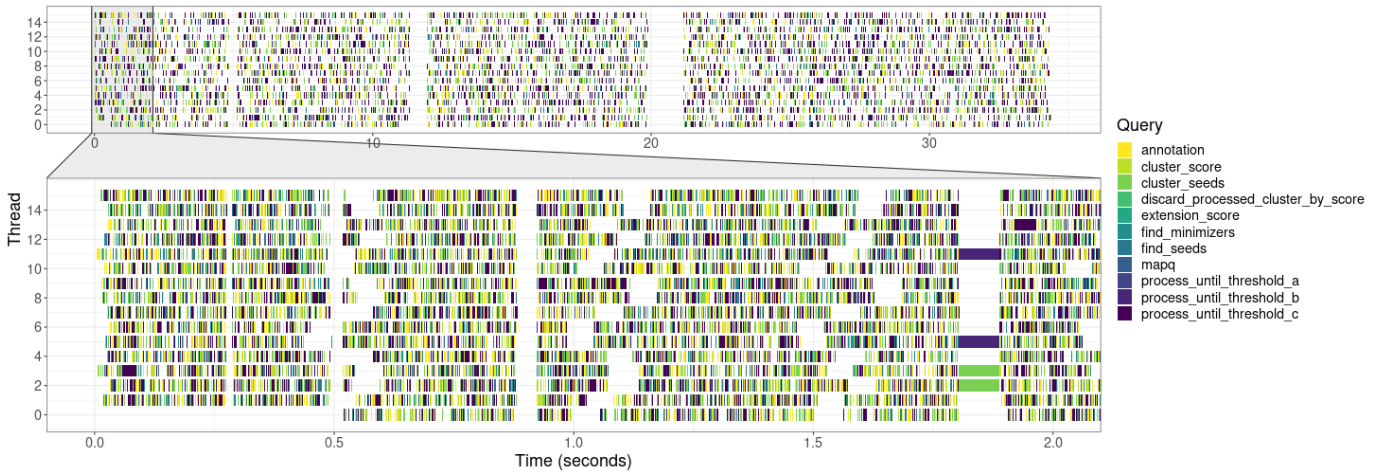


Fig. 2. Timeline of how Giraffe uses 16 threads for the annotated portions of the code while mapping input set A-human. Excludes pre-processing and non-annotated regions. Each thread performs a varying mix of tasks of varying durations.

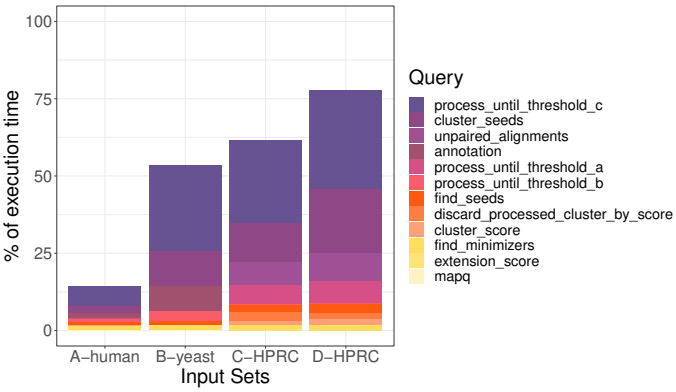
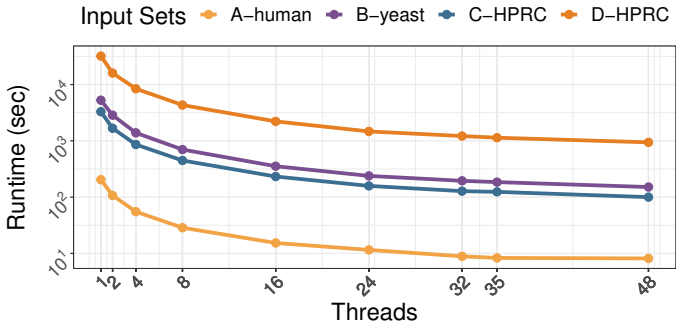
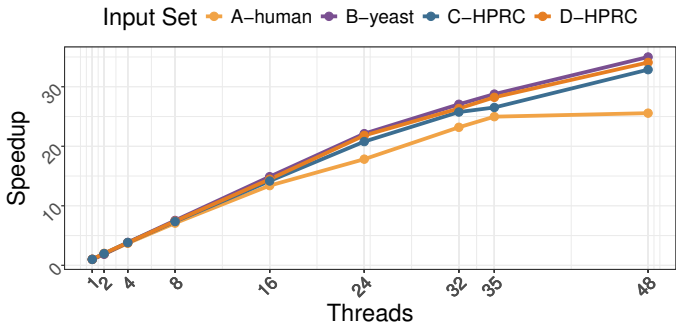


Fig. 3. Percentage of runtime for each instrumented region for all input sets. Times are aggregated per thread, and averaged across threads for each input.



(a) Execution time in log scale.



(b) Parallel speedup.

Fig. 4. Giraffe's parallel strong-scaling results for the extension.

sets A and D, respectively. Moreover, we observe a significant increase in execution time for the *threshold_c* function when processing a larger number of reads (input sets B-yeast and D-HPRC). Additionally, we note that VG's preprocessing causes significant overhead when running small read datasets (input set A-human). If we disregard the IO time, the *threshold_c* function represents 46.4% of total computation time for the input set A-human and 52% on B-yeast. This demonstrates that half of the computational time of Giraffe is spent on this function. The second most time-consuming region was *cluster_seeds* which consumes 11.6% on input set B-yeast and 21% on input set D-HPRC.

To better understand the impact of the input set on the execution time, we analyze its parallel scalability by sweeping the number of threads (and cores) for the *process_until_threshold_c* function. Input sets with a greater number of reads to map naturally consume more execution time (Figure 4a). Input set D-HPRC took more than 8 hours to complete sequentially, but reduced to $\approx$ 40 minutes using 48 threads. Input set A-human is the smallest, so its execution times are smaller, and they shrink from 200 to 8.14 seconds when running with 1 and 48 threads, respectively.

Regarding parallel scalability (Figure 4b), we observe that the speedup for the smallest input (A-human) begins to plateau after 35 threads, indicating that the benefits of adding more resources diminish beyond this point. In contrast, the larger input sets, which contain a greater number of reads to map, do not exhibit this same behavior and continue to show performance gains up to 48 threads. These results clearly demonstrate the significant impact of the number of reads required to map on the application's execution time.

TABLE IV
VTune Top-Down Metrics from Giraffe mapping A-human input set. Data within () is the second level of top-down analysis, being front-end latency and memory operations, respectively.

	Front-End	Back-End	Bad Spec.	Retiring
Bound %	23.5 (10.9)	22.8 (15.6)	10.2	43.4

To move beyond execution time and understand how effectively the hardware is utilized, we performed a Top-Down Microarchitecture Analysis using Intel's VTune Profiler [13] and present the results in Table IV. This top-down result reveals a relatively efficient execution profile, but with clear areas for optimization. The retiring percentage demonstrates that a significant portion of the work is experiences few stalls. However, the performance is considerably affected by bounds from both the front-end and the back-end, which is typical for full-size real-world applications as opposed to simple math kernels. Looking deeper into the back-end results, memory operations contribute 15.6% to the stalls, a finding consistent with our earlier observation that the number of reads is critical to performance. This suggests the CPU is frequently stalled waiting for data from the memory subsystem and could be a key area for optimization. Similarly, the front-end latency at 10.9% points to potential issues with instruction fetching and decoding. Although graph traversals are a significant portion of the extension, the rest of the workload makes the overall characteristics more complex and compute-intensive than typical graph kernels [4].

Based on our characterization, we observe that two main regions play significant roles in the mapping process and execution time: *process_until_threshold_c* and *cluster_seeds*. We focused our further analysis of Giraffe on these portions.

B. Giraffe's Read Mapping Process

The mapping function is the core of Giraffe's execution. Running in each thread, each workflow preprocesses the data to be mapped slightly differently. The first operation finds the minimizers and the distance index information for the short read being processed. Once found, the application creates a vector of *seeds*. A seed is a pair containing the pangenome graph node and a score indicating the probability of a match when starting the mapping walk from that node. These seeds are crucial for the rest of the mapping process since they are where the walks over the graph comparing the read and reference will start.

With the seeds found, the application calls the first significant code region: *cluster_seeds*. This region is responsible for creating clusters of the seeds found for that specific input read and determining quality scores for them to facilitate the mapping strategy. Such clusters are the main input for the next big step in the process: *process_until_threshold_c*. This region is responsible for finding possible matches and evaluating their quality. This region calls the *extension* function, which performs the whole walk and compare operation. Using the seeds' cluster found for that match, this extension function considers mappings by extending matches (i.e., a new node in the graph is considered a match with the read) according to their characteristics. If the considered node has a high match score, the function walks over the graph to that next node, comparing its set of characters with a specific offset in the sequence read. It continues extending until it stops when the end of the short read is reached. This process is called *seed-and-extend*, as briefly discussed in Section II. Depending on the extension's quality, which is determined using a scoring function, the batched reads are evaluated as matched or not in the reference, and the result is generated.

With the extensions for the read found, the application enters a post-processing phase. The phase performs more data refinement to guarantee that the matches are accurate. The first step scores the extensions found for that cluster and discards non-critical information such as extensions with low scores. The application then continues to the alignment phase, which generates the mapping output.

V. miniGiraffe: the Giraffe Proxy Application

We make the following observations based on Giraffe's workload characterization (Section IV-A): 1) The mapping operation, specifically the function that extends the search from the seeds, is the most time-consuming kernel within Giraffe, followed by the function responsible for clustering the seeds. We call these functions the *critical functions* in Giraffe; 2) Giraffe and VG have a complex source code that contain many dependencies for data structures and functionalities, which makes it difficult to experiment with and modify the code, e.g., to develop new optimizations; 3) Beyond software optimizations, the complexity of Giraffe makes it difficult to directly develop a hardware accelerator for its real-world pangenome mapping approach.

Given these observations, we propose *miniGiraffe*, a proxy application based on the Giraffe pangenome mapping tool. miniGiraffe makes simplifications to make the application lightweight while still capturing the key features of Giraffe's critical functions. For these key functions, miniGiraffe produces the exact same output. Using this proxy, researchers can explore various optimization strategies for pangenome mapping, e.g., reducing execution time, testing new scheduling strategies to improve scalability, reducing memory consumption, and optimizing storage accesses.

The process to build this proxy was based on the observations made during the workload characterization of Giraffe. We followed an iterative methodology that consisted of profiling the parent application, understanding its bottlenecks, implementing the functions we found to be significant for the application's execution time, and validating the results produced by the proxy.

miniGiraffe is based on the seed-and-extension process from the full application. This step consumes the majority of the execution time on average across our input sets, and it is where the actual comparison between the short-read and the pangenome data (variation graph) occurs (Section IV-B).

However, because these functions are deep within the Giraffe code, the inputs for miniGiraffe include some preprocessing Giraffe already performs. We extract these inputs after preprocessing from Giraffe right before executing the seed-and-extension process.

The preprocessed data that is miniGiraffe's input consists of the reads to be mapped and their respective seeds found on the pangenome. Seeds consist of pairs of memory offsets and structures that point to nodes in the variation graph. Along with the seed information, miniGiraffe also takes as input the pangenome reference as a GBWT graph. As mentioned in Section II-B, Giraffe uses the compressed GBZ file format to store GBWT graphs [30], and we kept the usage of this file format in our proxy. The final inputs are parameters for miniGiraffe to control the number of threads, batch size for parallel execution, and an initial capacity for the CachedGBWT structure. Users can also easily enable/disable instrumentation of the execution and hardware counter collection using input options.

Once miniGiraffe has loaded the input, it begins executing the mapping process by iterating over each short-read and its respective seeds in nested loops, which replicates the original application's behavior. In miniGiraffe, this main outer loop is parallelized, and its structure eases experimentation with schedulers different than Giraffe's. Focusing on enabling straightforward exploration of optimizations while keeping as close as possible to the parent application's behavior, miniGiraffe also uses OpenMP. Our results show that OpenMP's dynamic scheduling of batches of reads provides similar execution time and scaling, at least up to 16 threads, when compared with the complex VG scheduler used by Giraffe.

Since miniGiraffe focuses on the mapping process, it does not implement the post-processing Giraffe applies to the mapping results. Thus, miniGiraffe's output consists of the raw mapping results, i.e., the offsets and scores of each match. To validate the functionality of miniGiraffe, we can export the expected output after the corresponding mapping operations from the full application.

VI. miniGiraffe Validation

a) Functional Validation: A proxy app should have similar, if not exactly the same, outputs as the region of the parent application it is derived from. This provides confidence that the proxy is performing computation representative of the original. For the validation, we export the extensions found by Giraffe by running the selected input sets. Since such extensions are also the output from miniGiraffe, we can load both outputs and simply validate by comparing the outputs. The validation asserts two properties: (1) all of the expected queries are found in the proxy output; and (2) the proxy output does not contain matches that are not included in the expected output. Our validation shows a 100% match between both the proxy and the parent outputs for all input sets, demonstrating miniGiraffe's accuracy.

b) Performance Validation: To validate that miniGiraffe is computationally representative of Giraffe, we compare performance and hardware performance counter measurements for both. We collect the following:

- IPC (instructions per cycle)
- Application execution time
- Data cache accesses and misses
- Cache miss rates

TABLE V
Hardware performance counter measurements for validation of seed-and-extension on input set A-human.

Application	Inst.	IPC	L1DA	L1DM	LLDA	LLDM
miniGiraffe	1.49e12	1.91	4.19e11	1.69e9	2.88e8	2.11e8
Giraffe	1.33e12	1.67	3.87e11	4.54e9	3.83e8	2.12e8

TABLE VI
Execution time (seconds) comparison between Giraffe and miniGiraffe across 4 inputs sets.

	A-human	B-yeast	C-HPRC	D-HPRC
miniGiraffe	186	4302	2741	27045
Giraffe	171	4068	2561	24989
% diff over Giraffe	8.77	5.75	7.02	8.22

Since local-intel hosts the VG application, we also used this machine to collect all of the performance validation data. When collecting these metrics for Giraffe, we instrumented only the code sections the proxy covers, specifically the seed-and-extend functions. This allows us to only measure the components of Giraffe that miniGiraffe aims to capture, without the additional overhead, e.g., data pre- and post-processing. The performance counters are quite similar for both executions (Table V). Cache access and misses are indicated by L1 Data Access (L1DA) and L1 Data Misses (L1DM), as well as last-level cache data access (LLDA) and misses (LLDM). The measurements were collected with input set A-human, because it was the smallest input, making single-threaded execution more tolerable.

The total instruction count between miniGiraffe and Giraffe is similar; however, miniGiraffe's IPC is slightly higher than Giraffe's, and thus, miniGiraffe's total cycles were fewer than Giraffe's. miniGiraffe accesses the L1D cache more times but has substantially fewer misses. The miss rate is 0.004 for miniGiraffe and 0.011 for Giraffe. At the last-level cache (LLDA and LLDM), Giraffe performs more accesses, but both have a similar number of misses. Thus, miniGiraffe has a higher miss rate of 0.73 compared to 0.55 for Giraffe. We hypothesize that these cache differences are due to other small operations that Giraffe performs intermittently with the extension, which can make the data in the L1 cache change more frequently than miniGiraffe. Those additional L1D cache misses for Giraffe are well-handled by the next levels of the cache hierarchy. We consider the tight congruence of last-level cache misses to be the most important indicator that the proxy is stressing the same aspects of the system.

To quantitatively support this validation, we performed a cosine similarity analysis between the hardware counters, technique used in [28]. In this analysis, commonly used in text-mining algorithms, the metric is collected from the cosine angle multiplication value of two non-zero vectors being compared [17]. A value closer to 1 indicates that the two vectors are similar. Applying this technique to our metrics, we obtained a score of 0.9996, indicating that parent and proxy applications have nearly identical characteristics.

To conclude the performance validation, we demonstrate that the miniGiraffe's execution time is extremely close to Giraffe's and has a maximum difference of 8.7% across all input sets (Table VI). We ran both miniGiraffe and Giraffe three times each, measured execution times, and calculated the average of these executions for all input sets. Since the goal of proxy applications is to reproduce key workload properties, this demonstrates that our proxy also achieves similar performance, which is a great bonus.

VII. Case Studies Utilizing miniGiraffe

To demonstrate the proxy's utility, we performed two case studies. The first explores the behavior of this pangenome-mapping workload on different systems. Due to the proxy's simplicity, compiling and running the workload on the four different systems (described in Table II) required only minimal effort. The second explores parameter tuning to accelerate the workload, including how the workload's characteristics impact the tuning. Both case studies using the proxy required far less development effort than they would have required if the original Giraffe code base were used.

A. Parallel Scaling on Different Systems

To observe the impact of different hardware characteristics on pangenome mapping, we execute miniGiraffe on the four different systems described in Table II. The systems vary in hardware characteristics such as core count, threads per core, DRAM capacity, cache hierarchies, core microarchitectures, and vendors. The combination of system characteristics allows us to not only demonstrate how portable our proxy application is, but also brings a broader view of the behavior of this important workload on different systems. Figure 5 depicts the parallel scalability for each of the four input sets. The dotted black line represents the ideal linear speedup for each machine. The systems with less memory (chi-arm and chi-intel), run out of memory for input set D.

We can observe that each system presents significant differences in scalability, with both Intel-based systems (local-intel and chi-intel) demonstrating the most sublinear results due to scaling across sockets and hyperthreads. Once each core is used by one thread (48 threads for local-intel, 80 threads for chi-intel), using the additional hyperthread contexts does not provide much benefit. Multiple threads in the same core contend for core and cache resources. Chi-intel demonstrates near-linear speedups on input set B-yeast for up to 72 threads, while the other two inputs have impaired scalability after 32 threads. On the local-intel system, all input sets scale linearly

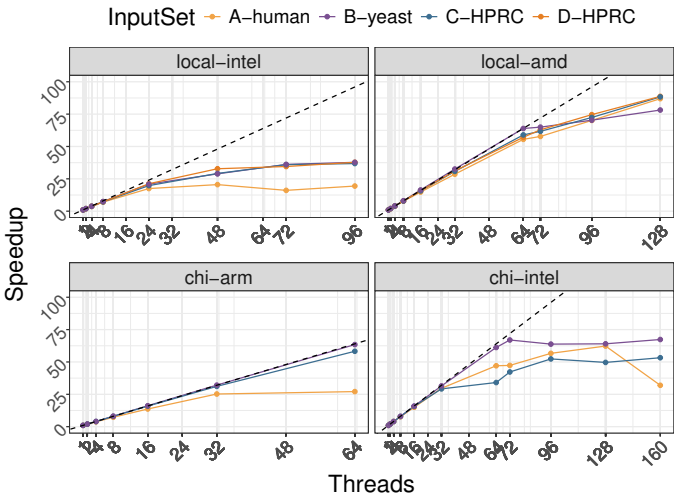


Fig. 5. Parallel scalability of miniGiraffe on four different systems. The dotted black line represents the ideal speedup for each system. Both servers chi-arm and chi-intel ran out of memory for input set D-HPRC.

up through 24 threads (1 socket without hyperthreads), obtain slight speedups with 2 sockets, and plateau or even slow down with hyperthreads.

In contrast, the local-amd and the chi-arm systems often demonstrate near-linear speedups for almost all input sets. Local-amd, which has only a single socket, enjoys near-linear speedups until using hyperthreading (beyond 64 threads), at which point it still obtains an appreciable speedup. At 128 threads, local-amd's speedups are well below linear, but the highest for this experiment with speedups of 86.9×, 78.2×, 88.3×, and 88.7× for input sets A, B, C, and D, respectively. On chi-arm, only input set A-human has sublinear speedups, and it plateaus after 32 threads. This might be explained by input set A's smaller size. Since we process sequences in parallel, the scalability of the application is directly linked to the number of short reads each thread will be responsible for mapping. Another interesting point to notice is that it is an example of great scalability of the application executing on a non-x86 machine.

Despite presenting the most consistent speedups, the absolute execution times from chi-arm are the slowest for all input sets. Table VII present the fastest execution times across all input sets and machines. Focusing on the execution time from local-amd and chi-arm, we observe a difference of more than 8× for input set A-human and 4× for input set C-HPRC. Another significant difference is shown on the execution time of D-HPRC over the local servers, with local-intel performing almost 3× slower than local-amd. Even though chi-intel's performance plateaus, its fastest executions are the second fastest results. The fastest two systems have the largest last-level caches with local-amd having 256 MB and chi-intel having 60 MB per socket. The main operation performed by miniGiraffe is data comparisons, which is data-intensive and can thus benefit from large-scale cache locality. These performance results demonstrate how this workload is directly affected by the memory and hardware cache capabilities of the system being used. Focusing on ways to improve cache

TABLE VII
FASTEST EXECUTION TIMES (IN SECONDS) FOR EACH INPUT SET ACROSS DIFFERENT SYSTEMS.

Input Set	local-intel	local-amd	chi-arm	chi-intel
A-human	9.06	1.60	13.42	3.44
B-yeast	113.75	42.09	137.86	73.44
C-HPRC	74.44	23.25	97.95	59.36
D-HPRC	681.82	229.42	–	–

locality and parallelism can accelerate the workload, which we consider in the next case study.

B. Exposing, Tuning, and Analyzing Parameters

Motivated by the scalability results, we sought to investigate how different parameters and their interactions with the hardware platform affect the application's performance. Utilizing the simplicity of our proxy miniGiraffe, users can easily experiment with different application strategies and apply tuning techniques. mimiGiraffe offers three different tuning parameters: scheduling technique, batch size, and initial CachedGBWT capacity (a software construct). For scheduling, we extended the initial OpenMP implementation by implementing a work-stealing scheduler using C++ threads. In this strategy, the total workload is spread across threads equally, but each thread will process its workload in batches of *batch size* short reads. If a thread finishes its work before others, it can steal a batch-size chunk of work from another thread in a round-robin approach using an atomic read-modify-write instruction. Our approach is lightweight and intended to remove some of the overhead, imbalance, or loss of locality that the default OpenMP dynamic scheduler might have.

Beyond the choice of parallel scheduler, exposing the parameters required code changes and intentional design for miniGiraffe. Although Giraffe has a batch size parameter, it is not easily changed, and we wanted to experiment with changing it from its default value of 512. Giraffe uses a CachedGBWT to hold uncompressed portions of the reference pangenome (Section II-B). Although this structure can grow by performing an expensive rehash operation, we find setting its initial capacity to be impactful. We modified the code to expose this parameter so it could be easily changed from its default value of 256.

We conducted an auto-tuning experiment across all our servers to understand the impact of these parameters on this workload and potential benefits. This experiment is leveraging the ease with which our proxy can run on different platforms, as well as changing parameters that were originally fixed or not externally configurable. To be able to run all four input sets on all machines for all combinations of parameters, we subsampled our input sets. For each input set, we only used the first 10% of reads. This subsampling both reduced the total experiment time but it also shrank the large D-HPRC so it no longer ran out of memory on some systems.

To reduce the parameter search space, we first investigated the impact of the initial CachedGBWT capacity. From a preliminary test using one of the largest inputs (C-HPRC) on local-intel (Figure 6), we observe that the maximum speedups occur when the initial capacity is 4096 or less for both schedulers (OpenMP and our in-house work-stealing scheduler). Larger initial capacity presents performance degradation. Thus, in subsequent explorations, we limited the considered initial capacity to 4096 or less.

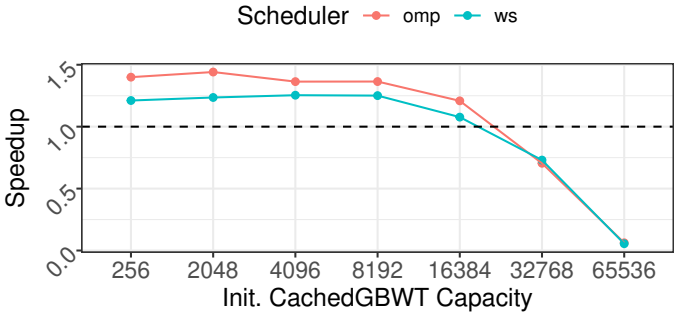


Fig. 6. Speedup results for different initial CachedGBWT capacities against no usage of this caching structure. Tests collected with the C-HPRC test case at local-intel machine

TABLE VIII
CONFIGURATION PARAMETERS FOR FASTEST RESULTS, INCLUDING BATCH SIZE (BS COLUMNS), CACHEDGBWT CAPACITY (CC COLUMNS), AND SCHEDULER (OPENMP UNLESS MARKED WITH * FOR WORK-STEALING).

	local-intel		local-amd		chi-arm		chi-intel	
Input Set	BS	CC	BS	CC	BS	CC	BS	CC
A-human	256	4096	512	2048	256	4096	128	2048
B-yeast	128	512*	128	512	256	1024*	128	1024*
C-HPRC	256	4096	128	4096	512	256*	128	4096
D-HPRC	128	2048	1024	4096	1024	2048*	512	1024

We exhaustively auto-tune (full cross-product) across the scheduler used, batch size, and initial CachedGBWT capacity. For the batch size, we considered batch sizes in powers of 2 from 128 to 2048. For each platform, we use the maximum number of available thread contexts (including hyperthreads), which is 96, 128, 64, and 160 on local-intel, local-amd, chi-arm, and chi-intel. For this analysis, we consider the *makespan*, i.e. the end-to-end execution time.[1] For each system and input, we identify the best performing set of parameters and present their performance in Figure 7 and report those configuration parameters in Table VIII.

By tuning the parameters, we are able to achieve significant speedups on all input sets on almost all systems. Interestingly, most of the best performers do not share the same configuration, nor use the default values (OpenMP, 512 batch size, 256 initial CachedGBWT capacity). Despite our focus on lightweight performance, our work-stealing scheduler is the fastest on only 5 of 16 scenarios, with 3 of them on the chi-arm system. The most significant overall speedup from all of our tuning was 3.32×, and it was achieved on input set A on the chi-arm system. The smallest overall speedup

[1]We use the term makespan since in the scheduling literature "execution time" can refer to the aggregate CPU time as opposed to wall clock time.

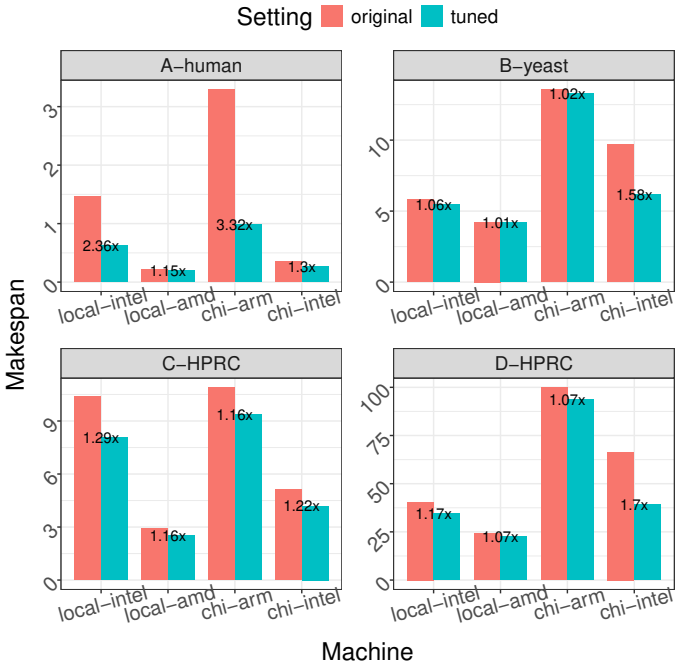


Fig. 7. Makespan (seconds) comparison of the best tuning performers against the default parameters for each input set over each machine. Executions use all available threads in each machine, being 96, 128, 64, and 160 for local-intel, local-amd, chi-arm, and chi-intel, respectively.

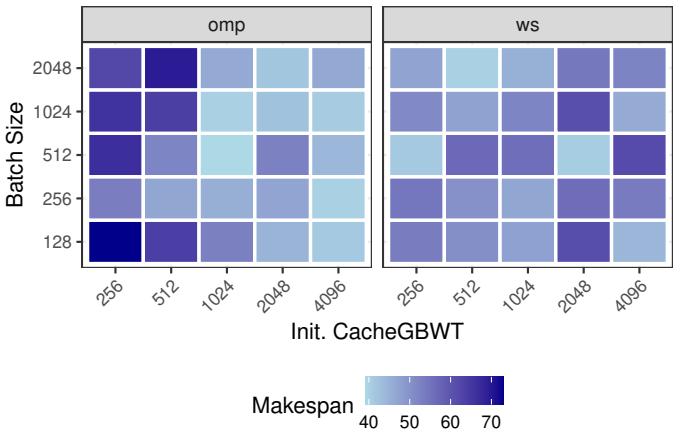


Fig. 8. Heat map with makespan distribution of all different combinations of tuning parameters over input set D-HPRC at chi-intel machine.

was 1.01×, and it occurred on input set B on the local-amd system. In fact, input set B-yeast only benefits substantially from tuning on the chi-intel machine. Only input set C-HPRC presented a smaller variability in speedup between systems. The geometric mean speedup for each input set is respectively 1.36×, 1.07×, 1.10×, and 1.11×.

Regarding absolute performance, the local-amd machine was the fastest for all input sets but gained the least speedups from parameter tuning. The makespan of the fastest executions on local-amd were 0.193 s, 4.18 s, 2.53 s, and 22.5 s for inputs A, B, C, and D, respectively. The second fastest system varied for each input set. These results demonstrate that the application is less affected by different tuning parameters when running on powerful hardware (e.g., with a large capacity L3 cache). They also indicate that the combination of how much work each thread runs, alongside hardware characteristics, directly impacts the application's performance. Thus, finding good combinations of parameters can be crucial for performance improvements on each different set of inputs and servers. For instance, D-HPRC and B-yeast have their best speedups from tuning on chi-intel machine, which has the highest number of available threads. However, both the A-human and C-HPRC input sets have their makespan most impacted by tuning on the local-intel machine, which has the smallest hardware cache capacity.

To understand the nature of the parameter space, we plot the makespan across all parameter combinations for input set D-HPRC on the chi-intel system (Figure 8). We selected this system since it had the largest variance in parallel scalability and this input since it is the largest. We observe a significant difference between the best and worst performers on this system. Most of these combinations significantly impact performance, demonstrating the importance of parameter tuning. By choosing the best tuning combination, it is possible to avoid a 1.76× slowdown for this input set on this system. More interestingly, the default parameters produce one of the slowest executions.

To conclude this case study, we performed an Analysis of Variance (ANOVA) to quantify the impact of each parameter on the makespan. The analysis reveals that the initial CachedGBWT capacity is the most impactful parameter in this case, demonstrating a statistically significant effect on the results (p=0.047). In contrast, neither the number of Batches (p=0.878) nor the choice of Scheduler (p=0.859) was found to have a significant influence on performance for this combination of input set and server The proxy significantly eased performing this case study, as it eased modifying, instrumenting, and porting the code. This demonstrates how miniGiraffe will be an excellent platform for future research in accelerating pangenome read mapping.

VIII. Related Work

Existing proxy applications have been used to understand the energy efficiency of big applications [34], to port protein-ligand docking programs to different HPC computer architectures [32], and to test different programming models over a shock hydrodynamics application [14]. The first proxy solves elastic wave equations, and allowed researchers to learn weaknesses and improve proxy performance and energy efficiency by focusing on memory-centric operations [34]. For the second, the proxy application based on the AutoDock-GPU particle-grid-based program [18] was ported to run in GPUs from different HPC facilities and vendors. For the third and last, the proxy was ported to different programming models such as Chapel, Charm++, Liszt, and Loci [14]. The authors were able to reduce the program source code by 80% when comparing the Loci and Chapel with the MPI versions. All

of these examples show how versatile proxy applications are. Having a smaller version of a real-world complete scientific application can allow the easiest investigation of energy efficiency, hardware portability, and new coding approaches.

In the field of genomics, many tools present different complexities, mainly regarding their big data. These tools need to be aware of their memory usage and data storage and how well they construct their data structures and algorithms to perform queries as efficiently as possible. Moreover, these kernels are in growing demand, which also requires more specialization in the computer science aspect to deal with the specificities of these kernels. There are some existing benchmarks that aim to enable the evaluation of existing hardware and technologies for genomics necessities, such as BioBench [2] and GenArchBench [21].

BioBench is a set of benchmark applications that aims to reflect the diversity of bioinformatics codes in use [2]. This is one the first benchmarks focused on genomics and it has tools for sequence similarity searching, multiple sequence alignment, sequence profile searching, and genome-level alignment. GenArchBench is the first genomics benchmark suite targeting Arm HPC architectures [21]. Their set of tools comprises algorithms focusing on seed chaining, FM-index search, k-mer counting, de Bruijn graph construction, among others. Most of the codes were optimized to run on x86 architectures, which the authors describe to be a challenge to port to non-x86 platforms.

Despite the efforts focusing on benchmarks and the broader existing application of proxy apps, by the time of this work, there are no benchmark algorithms that focus on pangenomes. We believe that having a proxy application based on a pangenome mapping tool will bring a complete tool for developing new technologies aiming to support and enhance genomics algorithms. Pangenomes will soon move from research and lead to better quality healthcare, so it is imperative we are able to support them computationally. When simulating an existing tool with closer characteristics, we can not just provide improvements to the original tool but also use a test case with greater accuracy to test new developments in general.

IX. Conclusions

In this work, we presented miniGiraffe, a proxy application for the pangenome mapping process within the Giraffe tool. miniGiraffe was designed to reproduce the Giraffe's critical functions and key behaviors while being much smaller and easier to work with. miniGiraffe's outputs exactly match Giraffe's mapping behavior. Additionally, we were able to use our proxy to understand the performance of this workload in different machines. Porting miniGiraffe to different hardware was straightforward and allowed us to observe the impact of L3 cache in this workload. Moreover, by using an auto tuning approach, we were able to accelerate the application by at most 3.36×, achieving an overall geometric mean of 1.15×.

Acknowledgment

We are grateful for all the input and guidance from the UCSC Genomics Institute VG Team, in particular to Benedict Paten, Jouni Siren, Xian Chang, and Adam Novak, for all the guidance and support with the VG Giraffe investigation. Part of the results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. This work was partially supported by the US DOE Office of Science project "Advanced Memory to Support Artificial Intelligence for Science" at PNNL. PNNL is operated by Battelle Memorial Institute under Contract DEAC06-76RL01830.

References

[1] O. Aaziz, J. Cook, J. Cook, T. Juedeman, D. Richards, and C. Vaughan, "A methodology for characterizing the correspondence between real and proxy applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 190–200.

[2] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "Biobench: A benchmark suite of bioinformatics applications," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, 2005, pp. 2–9.

[3] J. A. Baaijens, P. Bonizzoni, C. Boucher, G. Della Vedova, Y. Pirola, R. Rizzi, and J. Sirén, "Computational graph pangenomics: a tutorial on data structures and their applications," *Natural Computing*, vol. 21, no. 1, pp. 81–108, 2022.

[4] S. Beamer, "Understanding and improving graph algorithm performance," Ph.D. dissertation, University of California, Berkeley, 2016.

[5] G. P. Consortium, A. Auton, L. Brooks, R. Durbin, E. Garrison, and H. Kang, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, 2015.

[6] J. M. Eizenga, A. M. Novak, J. A. Sibbesen, S. Heumos, A. Ghaffaari, G. Hickey, X. Chang, J. D. Seaman, R. Rounthwaite, J. Ebler, M. Rautiainen, S. Garg, B. Paten, T. Marschall, J. Sirén, and E. Garrison, "Pangenome graphs," *Annual Review of Genomics and Human Genetics*, vol. 21, no. 1, pp. 139–162, 2020.

[7] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM (JACM)*, vol. 52, no. 4, pp. 552–581, 2005.

[8] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, W. Jones, S. Garg, C. Markello, M. F. Lin *et al.*, "Variation graph toolkit improves read mapping by representing genetic variation in the reference," *Nature biotechnology*, vol. 36, no. 9, pp. 875–879, 2018.

[9] W. F. Godoy, J. Delozier, and G. R. Watson, "Modeling pre-exascale amr parallel i/o workloads via proxy applications," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022, pp. 952–961.

[10] T. D. Hanson, "uthash: A hash table for c structures," https://troydhanson.github.io/uthash/, 2024, accessed: 2024-06-03.

[11] I. Illumina, "Paired-end vs. single-read sequencing technology," https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/paired-end-vs-single-read.html, accessed: 2024-12-18.

[12] N. H. G. R. Institute, "Haplotype," https://www.genome.gov/genetics-glossary/haplotype, 2023, [Accessed 22-11-2023].

[13] Intel, "Intel VTune Profiler Find and Fix Performance Bottlenecks Quickly and Realize All the Value of Your Hardware," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html\#gs.j8osh4, 2024, [Accessed 19-12-2024].

[14] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 919–932.

[15] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[16] W. J. Kent, R. J. H. Haussler *et al.*, "The ucsc genome browser," *Genome Research*, vol. 12, no. 6, pp. 996–1006, 2002. [Online]. Available: https://genome.ucsc.edu/

[17] A. R. Lahitani, A. E. Permanasari, and N. A. Setiawan, "Cosine similarity to determine similarity measure: Study case in online essay assessment," in *2016 4th International Conference on Cyber and IT Service Management*, 2016, pp. 1–6.

[18] S. LeGrand, A. Scheinberg, A. F. Tillack, M. Thavappiragasam, J. V. Vermaas, R. Agarwal, J. Larkin, D. Poole, D. Santos-Martins, L. Solis-Vasquez, A. Koch, S. Forli, O. Hernandez, J. C. Smith, and A. Sedova, "Gpu-accelerated drug discovery with docking on the summit supercomputer: Porting, optimization, and application to covid-19 research," in *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, ser. BCB '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388440.3412472

[19] J.-P. Lehr, C. Bischof, F. Dewald, H. Mantel, M. Norouzi, and F. Wolf, "Tool-supported mini-app extraction to facilitate program analysis and parallelization," in *Proceedings of the 50th International Conference on Parallel Processing*, ser. ICPP '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3472456.3472521

[20] W.-W. Liao, M. Asri, J. Ebler, D. Doerr, M. Haukness, G. Hickey, S. Lu, J. K. Lucas, J. Monlong, H. J. Abel *et al.*, "A draft human pangenome reference," *Nature*, vol. 617, no. 7960, pp. 312–324, 2023.

[21] L. López-Villellas, R. Langarita-Benítez, A. Badouh, V. Soria-Pardos, Q. Aguado-Puig, G. López-Paradís, M. Doblas, J. Setoain, C. Kim, M. Ono, A. Armejach, S. Marco-Sola, J. Alastruey-Benedé, P. Ibáñez, and M. Moretó, "Genarchbench: A genomics benchmark suite for arm hpc processors," *Future Generation Computer Systems*, vol. 157, pp. 313–329, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X24001250

[22] G. Manzini, "An analysis of the burrows—wheeler transform," *J. ACM*, vol. 48, no. 3, p. 407–430, may 2001. [Online]. Available: https://doi.org/10.1145/382780.382782

[23] S. Matsuoka, J. Domke, M. Wahib, A. Drozd, A. A. Chien, R. Bair, J. S. Vetter, and J. Shalf, "Preparing for the future—rethinking proxy applications," *Computing in Science & Engineering*, vol. 24, no. 2, pp. 85–90, 2022.

[24] D. Morneau, "Pan-genomes: moving beyond the reference," *Nat. Res*, 2021.

[25] NCBI Resource Coordinators, "Genome reference consortium human build 38," *Database*, vol. 2016, p. baw106, 2016. [Online]. Available: https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001405.26/

[26] S. Nurk, S. Koren, A. Rhie, M. Rautiainen, A. V. Bzikadze, A. Mikheenko, M. R. Vollger, N. Altemose, L. Uralsky, A. Gershman, S. Aganezov, S. J. Hoyt, M. Diekhans, G. A. Logsdon, M. Alonge, S. E. Antonarakis, M. Borchers, G. G. Bouffard, S. Y. Brooks, G. V. Caldas, N.-C. Chen, H. Cheng, C.-S. Chin, W. Chow, L. G. de Lima, P. C. Dishuck, R. Durbin, T. Dvorkina, I. T. Fiddes, G. Formenti, R. S. Fulton, A. Fungtammasan, E. Garrison, P. G. S. Grady, T. A. Graves-Lindsay, I. M. Hall, N. F. Hansen, G. A. Hartley, M. Haukness, K. Howe, M. W. Hunkapiller, C. Jain, M. Jain, E. D. Jarvis, P. Kerpedjiev, M. Kirsche, M. Kolmogorov, J. Korlach, M. Kremitzki, H. Li, V. V. Maduro, T. Marschall, A. M. McCartney, J. McDaniel, D. E. Miller, J. C. Mullikin, E. W. Myers, N. D. Olson, B. Paten, P. Peluso, P. A. Pevzner, D. Porubsky, T. Potapova, E. I. Rogaev, J. A. Rosenfeld, S. L. Salzberg, V. A. Schneider, F. J. Sedlazeck, K. Shafin, C. J. Shew, A. Shumate, Y. Sims, A. F. A. Smit, D. C. Soto, I. Sović, J. M. Storer, A. Streets, B. A. Sullivan, F. Thibaud-Nissen, J. Torrance, J. Wagner, B. P. Walenz, A. Wenger, J. M. D. Wood, C. Xiao, S. M. Yan, A. C. Young, S. Zarate, U. Surti, R. C. McCoy, M. Y. Dennis, I. A. Alexandrov, J. L. Gerton, R. J. O'Neill, W. Timp, J. M. Zook, M. C. Schatz, E. E. Eichler, K. H. Miga, and A. M. Phillippy, "The complete sequence of a human genome," *Science*, vol. 376, no. 6588, pp. 44–53, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abj6987

[27] I. G. S. Resource, "Structural variation data collection," https://www.internationalgenome.org/data-portal/data-collection/structural-variation, 2024.

[28] D. F. Richards, O. Aziz, J. Cook, H. Finkel, B. Homerding, T. Judeman, P. McCorquodale, T. Mintz, and S. Moore, "Quantitative performance assessment of proxy apps and parents," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States …, Tech. Rep., 2018.

[29] S. Secomandi, G. R. Gallo, R. Rossi, C. Rodríguez Fernandes, E. D. Jarvis, A. Bonisoli-Alquati, L. Gianfranceschi, and G. Formenti, "Pangenome graphs and their applications in biodiversity genomics," *Nature Genetics*, vol. 57, no. 1, pp. 13–26, 2025.

[30] J. Sirén and B. Paten, "Gbz file format for pangenome graphs," *Bioinformatics*, vol. 38, no. 22, pp. 5012–5018, 2022.

[31] J. Sirén, J. Monlong, X. Chang, A. M. Novak, J. M. Eizenga, C. Markello, J. A. Sibbesen, G. Hickey, P.-C. Chang, A. Carroll, N. Gupta, S. Gabriel, T. W. Blackwell, A. Ratan, K. D. Taylor, S. S. Rich, J. I. Rotter, D. Haussler, E. Garrison, and B. Paten, "Pangenomics enables genotyping of known structural variants in 5202 diverse genomes," *Science*, vol. 374, no. 6574, p. abg8871, 2021. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abg8871

[32] M. Thavappiragasam, A. Scheinberg, W. Elwasif, O. Hernandez, and A. Sedova, "Performance portability of molecular docking miniapp on leadership computing platforms," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 36–44.

[33] B. Wajid and E. Serpedin, "Do it yourself guide to genome assembly," *Briefings in Functional Genomics*, vol. 15, no. 1, pp. 1–9, Nov. 2014, _eprint: https://academic.oup.com/bfg/article-pdf/15/1/1/27029244/elu042.pdf. [Online]. Available: https://doi.org/10.1093/bfgp/elu042

[34] X. Wu, V. Taylor, and Z. Lan, "Performance and energy improvement of ecp proxy app sw4lite under various workloads," in *2021 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2021, pp. 17–24.

[35] H. Zheng, C. Kingsford, and G. Marçais, "Improved design and analysis of practical minimizers," *Bioinformatics*, vol. 36, no. Supplement_1, pp. i119–i127, 07 2020. [Online]. Available: https://doi.org/10.1093/bioinformatics/btaa472

A. Artifact Appendix

A.1 Abstract

This artifact contains the source code and documentation for miniGiraffe. We demonstrate how to reproduce the validation of miniGiraffe, the parallel scalability analysis of the proxy, and the autotuning experiments. More specifically, we address the reproducibility of the results discussed in Figures 5 and 7, as well as Tables 5, 7, and 8.

A.2 Artifact check-list (meta-information)

- **Program:** miniGiraffe, Python, Rscript, Bash
- **Compilation:** gcc, CMake, Make
- **Run-time environment:** Ubuntu 22.04 or newest
- **Metrics:** Makespan, Hardware Counters
- **Output:** .csv files from the application, .png from analysis scripts
- **How much disk space required (approximately)?:** 40GB for one input set
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 3-24 hours (depending on the dataset)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** 10.5281/zenodo.16930194

A.3 Description

A.3.1 How to access

miniGiraffe artifact is available at Zenodo[1] and GitHub[2]. All scripts for reproducibility and instructions for environment setup can be found there. The source code from miniGiraffe is available at GitHub as well[3].

A.3.2 Hardware dependencies

miniGiraffe can run on any hardware. To obtain results as close as possible to those presented in this publication, we recommend that users use hardware platforms with similar characteristics given in Table II. The smallest input set provided in this paper needs 32GB of RAM.

A.3.3 Software dependencies

The artifact requires: `Python3, CMake, R-base (R language), Perf, libcurl, libssl, libfontconfig, libharfbuzz, libpng, libtiff, libjpeg` (all libs are dependencies of the R-tidyverse library). miniGiraffe only requires `CMake`.

To execute the experiments, the following Python libraries are required: `pyDOE3, requests, pandas`. The data analysis and plot require R `tidyverse` package.

A.3.4 Data sets

Input set A-human uses the 1000GPlons pangenome created by the VG team using variants from the 1000 Genome Project. It maps a single-end read input from the NA19239 individual. Input set B-yeast also explores the single-end workflow by using a yeast pangenome based on a full set of yeast samples, available on the UCSC Genome Browser. Input set C-HPRC uses the latest version of the human pangenome graph built using variants from the Genome Reference Consortium Human Build 38. The input set D-HPRC utilizes the latest pangenome, built from the complete human genome sequence CHM13. For both paired-end input sets, data comes from different fragments of the genome sequence of the NA24385 individual's son.

A.4 Installation

A.4.1 Installing dependencies

Download/clone the artifact repository [4]. Install all software dependencies (listed above) and clone miniGiraffe's repository with:

```
git clone --recursive \
https://github.com/jessdagostini/miniGiraffe.git
```

Important to clone miniGiraffe at your home directory since the experiments scripts will consider `$HOME/miniGiraffe` as the default path to find miniGiraffe's executables.

Those using Ubuntu/Debian OS can navigate to the artifact repository and run:

```
bash config-env.sh
```

This script will install all necessary dependencies and clone miniGiraffe's repository. It is important to note that for those using a Linux-based OS, to collect `perf` metrics, the user needs to run `sudo sysctl -w kernel.perf_event_paranoid=-1` to enable collection. This step is performed by the `config-env.sh` script for users of Ubuntu/Debian.

After installing all required packages, run the following script to install Python and R-script dependencies:

`bash install-python-and-R-deps.sh`.

Windows users can refer to this file for the necessary packages.

A.4.2 Compiling miniGiraffe

```
cd miniGiraffe
bash install-deps.sh
make miniGiraffe
make lower-input
source set-env.sh
./miniGiraffe
```

A.5 Experiment workflow

A.5.1 Reproducing miniGiraffe's hardware metrics

In this experiment, we collect hardware counters from the miniGiraffe's execution, highlighting the miniGiraffe's ease of use for collecting critical hardware metrics. In the paper, we also use these metrics alongside Giraffe's hardware counter metrics to validate the significance of our proxy compared to the parent application. Due to the significant complexity of building and modifying the complete VG Giraffe application to re-collect this data, we provide the original hardware metrics (located at: `data-from-paper/intelxeonplatinum8260cpu@240ghz/1`) and the specifications of the machine used for that initial collection (described at the `README.md` file in the artifact repository). Users running miniGiraffe on a machine with similar characteristics can use this provided data to reproduce the validation experiment closely.

To collect miniGiraffe's hardware metrics, run:

`python3 experiments/hw-counters-pipeline.py`.

This will collect the six hardware metrics from miniGiraffe and store them at `results/`. To parse the results and perform the analysis, run `Rscript analysis/table5-with-new-results.R`.

[1] https://doi.org/10.5281/zenodo.16930194

[2] https://github.com/jessdagostini/iiswc-minigiraffe-ae

[3] https://github.com/jessdagostini/miniGiraffe

[4] https://doi.org/10.5281/zenodo.16930194

A.5.2 Reproducing scalability analysis

In this experiment, we will reproduce the scalability analysis made with miniGiraffe and presented in Figure 5. Navigate to `experiments/` and execute `python3 scalability-pipeline.py`. This script will automatically download the A-Human (1000GP) input set used in the paper's experiments. It will run a set of different executions with varying numbers of threads, according to the machine's available threads.

It is also possible to run different input sets and collect their scalability performance. To do that, simply run:

```
python3 experiments/scalability-pipeline.py \
    <path/to/sequence-seeds.bin> \
    <path/to/giraffe-gbz> \
    <input-set-name>
```

miniGiraffe expects two input files: a `.bin` file with the seeds collected from Giraffe, and the pangenome graph in `.gbz/.gbwt` format. Given the size of the other input sets, they are not available for download in the direct miniGiraffe format. We provide a step-by-step guide on how to generate new input sets at the "Generate new input sets" description on the `README.md` of both miniGiraffe's source code and artifact repositories.

To visualize/analyze the results, after finishing the execution, execute `Rscript figure5-with-new-results.R`. This R-script will read the output data and plot a speedup graph similar to Figure 5. This script will generate the plot file with the scalability results and a table with the best makespan found for each input set. If other input sets are run, this script will automatically parse them and include them in the plot.

A.5.3 Reproducing auto-tuning experiments

In this experiment, we investigate the effect of various parameters on the performance of the mapping process. The goal is to find the best set of parameters for each case, and mainly to demonstrate how this can impact performance on this type of operation. Figure 7 presents this tuning for four different machines using four different input sets.

To reproduce the experiment, navigate to `experiments/` and execute `python3 tuning-pipeline.py`. It will create a sampled version of the original input set given and then combine different batch sizes, initial GBWTCache capacities, and schedulers exhaustively, running with all available threads on the system. This factorial set of experiments will then be launched by the script, collecting data and storing them in the `results/` folder.

It is also possible to run different input sets and collect their tuning metrics:

```
python3 experiments/tuning-pipeline.py \
    <path/to/sequence-seeds.bin> \
    <path/to/giraffe-gbz> <input-set-name>
```

To parse/visualize/analyze the results, after finishing the experiment, execute `Rscript figure7-with-new-results.R`. This script will generate a plot with the best setting comparison and also a table identifying which values were used in each parameter. If this is run with multiple input sets, the plot will automatically include their results.

A.6 Evaluation and expected results

Upon successful completion of the experiments, users should be able to 1) Obtain miniGiraffe's hardware counters to understand possible bottlenecks this mapping operation can present on current hardware (and, if running on a similar hardware where Giraffe's hardware counters were collected, validate the fidelity of miniGiraffe behavior to its parent application); 2) Observe the scalability of the mapping operation on pangenomes using the selected hardware and understand its tradeoffs, and the effect that different input sets can also present; 3) Evaluate the impact of tuning parameters when running this application on different hardware, and achieve performance gains in the application's execution. This experiment aims to demonstrate how miniGiraffe eases the testing of different strategies, parameters, and tuning for this sequence-to-pangenome mapping workload.

A.7 Notes

Users can also reproduce the analysis performed with the original datasets collected during the experiments for this paper. Data and scripts to re-plot Figures 5 and 7 and Tables 5, 7, and 8 are available at `data-from-paper/`.

A.8 Methodology

Submission, reviewing, and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae