MerBench: PGAS Benchmarks for High Performance Genome Assembly

Evangelos Georganas[5], Marquita Ellis[1,2], Rob Egan[3], Steven Hofmeyr[2], Aydın Buluç[1,2]
Brandon Cook[4], Leonid Oliker[2], Katherine Yelick[1,2]

[1]EECS Department, University of California, Berkeley, USA
[2]Computational Research Division, Lawrence Berkeley National Laboratory, USA
[3]Joint Genome Institute, Lawrence Berkeley National Laboratory, USA
[4] National Energy Research Scientific Computing Center, USA
[5] Parallel Computing Lab, Intel Corp.

ABSTRACT

De novo genome assembly is one of the most important and challenging computational problems in modern genomics; further, it shares algorithms and communication patterns important to other graph analytic and irregular applications. Unlike simulations, it has no floating point arithmetic and is dominated by small memory transactions within and between computing nodes. In this work, we introduce *MerBench*, a compact set of PGAS benchmarks that capture the communication patterns of the parallel algorithms throughout HipMer, a parallel genome assembler pipeline that has been shown to scale to massive concurrencies. We also present results of these microbenchmarks on the Edison supercomputer and illustrate how these empirical results can be used to assess the scaling behavior of the pipeline.

1 THE HIPMER ASSEMBLY PIPELINE

In this section we describe the basic algorithms used in the HipMer pipeline, our parallelization strategy, and the consequent communication patterns that motivate the MerBench suite. We refer the interested reader to our previous work for a detailed analysis of HipMer's algorithms and performance [1–4]. HipMer is implemented in Unified Parallel C (UPC) [5], a Partitioned Global Address Space (PGAS) parallel programming language, and its parallel algorithms heavily rely on the one-sided communication capabilities of UPC.

Although we focus on HipMer, the algorithms are relevant to all *de novo* assembly pipelines that are based on de Bruijn graphs. We focus on four major stages of HipMer (see Figure 1): *k-mer analysis*, *contig generation*, *read-to-contig alignment* and *scaffolding*, as well as *gap closing*, which is part of the scaffolding stage. The input to the genome assembly pipeline is a set of *reads*, which are short, *erroneous* sequence fragments of 100-250 letters sampled at random from a genome. The sampling is redundant at a depth of coverage d, so on average each position (base) in the genome is covered by d reads. This redundancy is used to filter out errors in the first stage (k-mer analysis). Sequencers produce reads in pairs with a known distance between them, a fact which is exploited later in the pipeline (scaffolding) to improve the assembly.

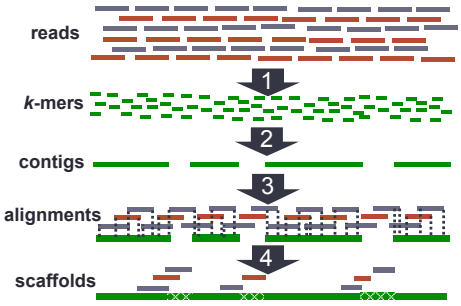

Figure 1: The HipMer assembly pipeline.

1.1 k-mer Analysis

In this step, the input reads are processed to exclude errors. Each processor reads a portion of the reads and chops them into *k-mers* (strings of length k), which are formed by a sliding window of length k. A deterministic function is used to map each k-mer to a target processor, assigning all the occurrences of a particular k-mer to the same processor, thus eliminating the need for a global hash table. The k-mers are communicated among the processors using **irregular all-to-all communication**, which is performed when each processor fills up out of its outgoing buffers and is repeated until all k-mers have been redistributed. A total of $\Theta(\frac{Gd}{L}(L-k+1))$ k-mers need to be communicated, where G is the genome size and L is the read length. Next, all the k-mers are counted, and those that appear fewer times than a threshold are discarded as erroneous. This filtering is enabled by the redundancy d in the read data set: k-mers that appear close to d times are likely error-free, whereas k-mers that appear infrequently are likely erroneous.

1.2 Contig Generation

The filtered k-mers from the pervious step are assembled into longer sequences called *contigs*, which are error-free (with high probability) sequences that are typically longer than the original reads. In HipMer, contig generation utilizes a de Bruijn graph, which is a special graph that represents overlaps in sequences. The k-mers are the vertices in the graph and two k-mers are connected by an edge if they overlap by $k-1$ consecutive bases and have corresponding extensions that are compatible A hash table is used to store a

PAW17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5123-2/17/11…$15.00
https://doi.org/10.1145/3144779.3169109

compact representation of the graph: A vertex (k-mer) is a key in the hash table and the incident vertices are stored implicitly as a two-letter code [ACGT][ACGT] that indicates the unique bases that immediately precede and follow the k-mer in the read dataset.

During the parallel hash table construction, the input k-mers are hashed and sent to the proper bucket of the hash table. We avoid fine-grained communication and excessive locking on the hash table buckets with a dynamic aggregation algorithm [1]. This algorithm dynamically aggregates the k-mers in batches before they are sent to the appropriate processors. The pattern here is similar to k-mer analysis but is done asynchronously, where a single processor will send an aggregation of remote hash table inserts without waiting for other processors. The resulting de Bruijn graph is traversed in parallel to identify the connected components, which are linear chains of k-mers. In our specialized parallel traversal algorithm [1], a processor P_i chooses a random k-mer as seed and initializes with it a new subcontig. Then P_i attempts to extend the subcontig towards both of its endpoints by performing **lookups** for the neighboring vertices in the distributed hash table. The extending process continues until no more new edges can be found, or there are forks in the graph. The access pattern in the distributed hash table consists of **irregular, fine-grained lookup** operations. If two processors work on the same connected component, race conditions are avoided via a lightweight synchronization scheme [1] based on **remote atomics**. The parallel traversal is terminated when all the connected components in the de Bruijn graph are explored. Since the size of the de Bruijn graph is proportional to the genome size, the traversal involves accessing $\Theta(G)$ vertices via atomics and irregular lookup operations.

1.3 Read-to-Contig Sequence Alignment

The alignment phase [3] maps the original reads onto the contigs to provide information about the relative ordering and orientation of the contigs, which is used in the final step of the assembly pipeline. First, each processor stores a distinct subset of the contigs in the global address space so that any other processor can access them. Then, substrings of length k, called *seeds*, are extracted in parallel from the contigs and stored in the seed index, which is a distributed hash table. Each hash table entry has a seed as the key and a pointer to the corresponding source contig as the value. There are $\Theta(G)$ seeds in total, because the contigs constitute a fragmented version of the genome. The seed index is constructed via an irregular **all-to-all communication** step similar to the hash table construction in the contig generation phase.

The seed index is then used to align reads onto contigs. Each read of length L contains $L-k+1$ seeds of length k. For each seed s in a read, a **fine-grained lookup** in the global seed index produces a set of candidate contigs that contain s. Although an exhaustive lookup of all possible seeds would require a total of $\Theta(\frac{Gd}{L}(L-k+1))$ lookups, in practice we perform $\Theta(\frac{Gd}{L} \cdot a)$ lookups where $a < L-k+1$, through the use of optimizations that identify properties in the contigs during the seed index construction [3]. Finally, after locating a candidate contig that has a matching seed with the read under consideration, the Smith-Waterman algorithm [6] is executed in order to perform local sequence alignment between the contig and the read. The output of this stage is a set of reads-to-contig alignments.

1.4 Scaffolding and Gap Closing

The scaffolding step aims to "stitch" together contigs into sequences called *scaffolds* by assessing the paired-end information from the reads and the reads-to-contigs alignments. A graph of contigs can be created by generating links for all the contigs that are supported by pairs of reads. The contig graph is stored in a distributed hash table, which requires **irregular all-to-all communication** for construction. A parallel traversal of the contig graph is then performed to identify and remove "bubbles", which are localized structures involving divergent paths. This requires **irregular lookups** and **global atomics**. A final traversal of the contig graph is done by selecting start vertices in order of decreasing contig length (this heuristic tries to first stitch together "long" contigs). The graph of contigs (and consequently the number of links among them) is orders of magnitude smaller that the k-mer de Bruijn graph because the connected components in the k-mer graph are contracted to single vertices in the contig-graph. According to the Lander-Waterman statistics [7], the expected number of contigs is $\Theta(dG/L \cdot e^{-d})$, where e is Euler's number.

It is likely that there will be gaps between the contigs within a scaffold. A distributed hash table is used to localize the unassembled reads onto the appropriate gaps. Construction of the table uses an **irregular all-to-all communication** pattern, but accessing the information in the table requires **irregular lookups**. Assuming that a fraction γ of the genome is not assembled into contigs, this communication step involves $\Theta(\gamma Gd/L)$ reads. Finally, the gaps are divided into subsets and each set is processed by a separate thread, in a parallel phase. The localized reads are used to attempt to close the gaps via a mini-assembly algorithm. The outcome of this step is a set of scaffolds (possibly with some remaining gaps), which constitutes the result of the HipMer assembly pipeline.

1.5 Summary of Communication Patterns

Table 1 summarizes the main communication patterns along with the corresponding volume of communication for each stage. These communications patterns govern the efficiency of the parallel pipeline at large scale, where most of the stages are communication bound. The different communication patterns have, however, vastly different overheads. For example, the all-to-all communication exchange is typically bounded by the bisection bandwidth of the system, assuming that the partial messages are large enough and there is enough concurrency to saturate the available bandwidth. Conversely, fine-grained, irregular lookups and global atomics are typically latency-bound.

2 THE MERBENCH SUITE

The MerBench microbenchmarks are designed to capture the irregular access patterns in HipMer. In this section we introduce these microbenchmarks and we also present results at different scales. We also compare the performance of the microbenchmarks with the empirical performance of the corresponding communication operations in HipMer. For the results presented in this paper we

Stage	Communication pattern	Volume of data
k-mer analysis	all-to-all exchange	$\Theta(Gd \cdot (L-k+1)/L)$
Contig generation	all-to-all exchange	$\Theta(G)$
	irregular lookups	$\Theta(G)$
	global atomics	$\Theta(G)$
Sequence alignment	all-to-all exchange	$\Theta(G)$
	irregular lookups	$\Theta(Gd \cdot a)$
Scaffolding	all-to-all exchange	$\Theta(G)$
	irregular lookups	$\Theta(G)$
	global atomics	$\Theta(dG/L \cdot e^{-d})$
Gap closing	all-to-all exchange	$\Theta(\gamma Gd/L)$
	irregular lookups	$\Theta(\gamma Gd/L)$

Table 1: Major communication operations in the HipMer pipeline. G is the genome size, L is the read length L, d is the coverage, a is the average number of contigs that each read aligns onto (with $a < L-k+1$), and γ is the fraction of reads that are not assembled into contigs.

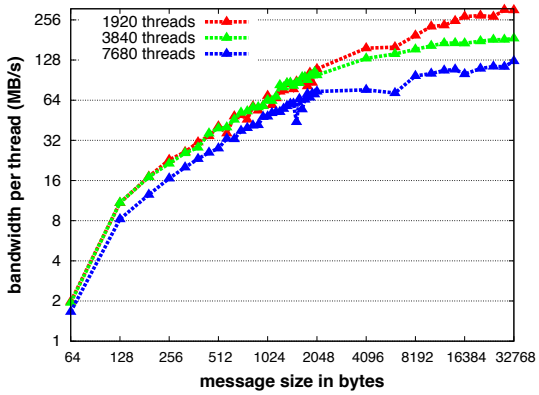

Figure 2: Proxy micro-benchmark for the all-to-all exchange in the distributed hash table construction with dynamic message aggregation.

utilized the Edison supercomputer at NERSC [8]– a Cray XC30 system – but the methodology is general and applicable to any other platform. The communication system of Edison is the Cray Aries high-speed interconnect with Dragonfly topology [9]. MerBench is implemented in UPC and leverages its one-sided communication capabilities.

2.1 Irregular All-to-all Exchange Benchmark

Here we describe a proxy benchmark that mimics the asynchronous all-to-all exchange in HipMer's distributed hash table construction phases. In this micro-benchmark:

(1) Every processor picks a random processor id p'
(2) Performs a remote atomic `fetch_and_add()` on an integer variable that belongs to p'
(3) Performs an aggregate remote transfer of size S to processor p'
(4) Repeat steps 1-3 multiple times

The atomic `fetch_and_add()` corresponds to the required action that ensures atomicity in the data exchange (i.e. avoid overwrite of remote data by multiple processors).

Concurrency (number of threads)	1,920	3,840	7,960
BW of proxy benchmark (GB/s)	300	533	543
BW of application code (GB/s)	87	112	109

Table 2: Comparison of communication proxy benchmark and application code that performs the all-to-all exchange. The message size is 6,400 bytes.

The Cray Aries interconnect on Edison has the physical infrastructure to provide at most 23.7 TB/s global bisection bandwidth. However, the Edison system is shared among multiple users and realistic use case scenarios do not involve the entire supercomputer. As a result, the attainable bandwidth at different scales varies from the full-system bisection bandwidth. All the experiments are performed in the same allocation of 7,680 cores (320 Edison nodes). Figure 2 illustrates the results of this microbenchmark for three different concurrencies: 1,920, 3,840 and 7,680 threads. The x axis shows the message size S and the y axis shows the attained effective bandwidth per thread. As we increase the message size, the achieved per thread bandwidth is also increased and after some point it reaches a plateau. The proxy micro-benchmark also dictates the minimum required aggregate message size to saturate the available bandwidth at each concurrency. Note though that the dynamic message aggregation requires $S \times P$ times more memory per thread (given P threads in total). Therefore, one should tune the S parameter at each concurrency according to the available memory.

Table 2 compares the aggregate bisection bandwidth performance of the proxy benchmark to the all-to-all exchange in the distributed hash table construction phase. The message size in these experiments is 6,400 bytes. We emphasize that the proxy benchmark captures only the communication cost (e.g. it ignores buffer copies, hashing overheads, skewed data distributions, local computations) and consequently yields a loose upper bound in performance. Nevertheless, the empirical performance of the application is always within 5× of the upper bound provided by the proxy benchmark. By further investigating the HipMer results at the concurrency of 3,840 cores, we found that 47% of the execution time is required for local computations. In other words, the effective bisection bandwidth of the application considering only the communication operations is 219 GB/s, which is within 2.5× of the upper bound provided by the proxy microbenchmark.

2.2 Global Atomics Latency Benchmark

The next aspect of the communication interconnect that is stressed by the MerBench suite is the efficiency of global atomics. In particular, we focus on the global atomic `compare_and_swap()` that is crucial for the parallel de Bruijn graph traversal described in section 1. The proxy benchmark that models the communication and synchronization behavior of the graph traversal algorithm is the following:

(1) Every processor fetches a random entry from a distributed hash table
(2) Performs a global atomic `compare_and_swap()` on an integer variable of the previously selected entry in the hash table
(3) Repeat steps 1-2 multiple times

This proxy micro-benchmark represents the minimum required latency overhead in order to visit a vertex in the distributed de

Concurrency (number of threads)	1,920	3,840	7,960
Latency (μs) in proxy to visit a vertex	9.8	14.78	21.6
Latency (μs) in HipMer to visit a vertex	24.2	32.21	43.9

Table 3: Comparison of proxy benchmark and application code that performs the parallel de Bruijn graph traversal.

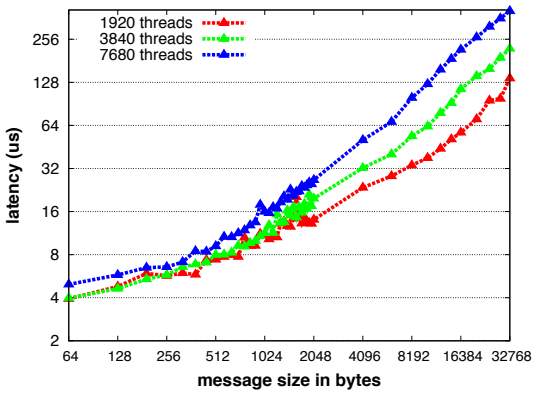

Figure 3: Latency of irregular lookup.

Bruijn graph traversal. It captures only the communication cost of this primitive operation and it does not consider any overheads due to the high-level synchronization protocol, hashing and local computations. Therefore, the proxy micro-benchmark provides a loose upper bound on the performance of the parallel graph traversal. Table 3 compares the performance of the proxy benchmark and the parallel graph traversal in HipMer. We conclude that the empirical performance of the application is always within roughly 2× and 3× of the upper bound provided by the proxy communication benchmark. The results also indicate that global atomics which take advantage of hardware support will speedup significantly this latency-sensitive parallel algorithm.

2.3 Irregular Lookup Latency benchmark

The last microbenchmark in MerBench measures the latency of the one-sided get operation at scale:

(1) Every processor fetches a random entry from a distributed hash table
(2) Repeat step 1 multiple times

This primitive is used to implement the lookup functionality in the distributed hash table and therefore is the limiting factor in most of the parallel algorithms with irregular lookup accesses (e.g. the sequence alignment algorithm). Figure 3 illustrates the remote get latency for different message sizes at three different concurrencies. We are particularly interested in the regime of small messages sizes because the hash table entries typically have size up to 128 bytes; for such message sizes the latency varies between 4 to 6 microseconds. Table 4 compares the latency of a get operation with the latency of a lookup in the distributed hash table used in the sequence alignment algorithm (the message size is 64 bytes). Note that the lookup operation also includes the hash computation, overhead to validate that the fetched entry is the one with the requested key and potential overheads to follow remote chain-pointers at hash table buckets with conflicts. Nevertheless, the empirical performance of the lookup is always within 3× of the latency of a single get operation at all concurrencies. The fact that the lookup latency remains almost constant at all concurrencies justifies the efficient scaling of the parallel sequence alignment. As we increase the number of cores, proportionally fewer lookups are executed concurrently on the critical path and since they have constant latency the total execution time decreases proportionally with the number of cores. The same scaling argument applies to all key operations in our parallel algorithms that involve irregular accesses in the distributed hash tables.

Concurrency (number of threads)	1,920	3,840	7,960
Latency of remote get (μs)	3.9	4.0	5.0
Latency of lookup in the hash table (μs)	11.7	11.4	9.81

Table 4: Comparison of get latency and latency of lookup in the distributed hash table. The message size is 64 bytes.

3 CONCLUSIONS

In this paper we outlined the key communication patterns in the HipMer de novo assembly pipeline and we introduced MerBench, a microbenchmark suite that measures the performance of these communication patterns, namely irregular all-to-all exchanges, irregular global atomics and irregular, fine-grained lookups. We presented results of MerBench on the Edison supercomputer and illustrated how they are representative of the application's empirical behavior. Even though we developed MerBench in order to assist HipMer's evaluation, MerBench can be used as a tool to understand the potential of a communication infrastructure that is stressed by a highly irregular application at scale. For an in-depth analysis of HipMer's scalability and its communication requirements we refer the interested reader to our previous work [4], where MerBench is used for the benchmarking of multiple systems. MerBench is publicly available at: `https://sourceforge.net/projects/hipmer`.

REFERENCES

[1] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, 2014.
[2] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "Hipmer: an extreme-scale de novo genome assembler," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, 2015.
[3] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "merAligner: A Fully Parallel Sequence Aligner," in *Proceedings of the IPDPS*, 2015.
[4] M. Ellis, E. Georganas, R. Egan, S. Hofmeyr, A. Buluç, B. Cook, L. Oliker, and K. Yelick, "Performance characterization of de novo genome assembly on leading parallel systems," in *European Conference on Parallel Processing*. Springer, 2017, pp. 79–91.
[5] P. Husbands, C. Iancu, and K. Yelick, "A performance analysis of the Berkeley UPC compiler," in *Proc. of International Conference on Supercomputing*, ser. ICS '03. New York, NY, USA: ACM, 2003, pp. 63–73.
[6] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.
[7] R. C. Deonier, S. Tavaré, and M. Waterman, *Computational genome analysis: an introduction*. Springer Science & Business Media, 2005.
[8] "Edison supercomputer," http://www.nersc.gov/users/computational-systems/edison/, accessed: 2017-12-6.
[9] G. Faanes, A. Bataineh, D. Roweth, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, J. Reinhard *et al.*, "Cray cascade: a scalable hpc system based on a dragonfly network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 103.