Genome Assembly using an Asynchronous Distributed Actor-Based Approach

Souvadra Hati
souvadrahati@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Richard W. Vuduc (advisor)
richie@cc.gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

ABSTRACT

We use genome assembly as a representative case to showcase the use of the 'actor model', a novel programming system for high-performance data-intensive workloads. The actor version of the k-mer counting kernel shows on average 1.6× speedup over similar MPI implementation. We provide a novel parallel algorithm that leverages the actor model to traverse de Bruijn graphs in a non-blocking, one-directional manner. Our findings highlight the potential of the actor model for writing simple and efficient parallel programs for data-heavy workloads.

ACM Reference Format:
Souvadra Hati and Richard W. Vuduc (advisor). 2023. Genome Assembly using an Asynchronous Distributed Actor-Based Approach. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This work is part of the ongoing research within the IARPA AGILE[1] program, which aims to design reliable programming systems and computer architectures for large-scale data-driven applications. This program includes the development and deployment of a novel actor-based programming model [7] (Section 2.1), which aims to enable high-performance large-scale data analytics software while also ensuring enhanced programmer productivity. The study in this poster evaluates this programming model on one of AGILE's target workloads, *de novo* genome assembly.

De novo genome assembly is a fundamental problem in bioinformatics and remains a computationally challenging problem. In simple terms, it's the process of reconstructing genomic sequences from short, overlapping DNA fragments (reads). The data generated by the sequencing hardware for complex genomes can be up to tens of terabytes, necessitating scalable parallel solutions to tackle it.

[1]https://www.iarpa.gov/research-programs/agile

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND

2.1 The Actor Model

The actor-model [1, 5, 7, 8] is a new programming system tailored for PGAS (Partitioned Global Address Space) applications, purpose-built to handle sparse data structures with irregular message patterns across multiple processing elements. It employs the Fine-grained-Asynchronous Bulk-Synchronous Parallelism (FA-BSP) model that extends the standard Bulk-Synchronous Parallelism (BSP) model to allow asynchronous communications in between synchronizing supersteps (figure 1). The use of the actor model ensures inherent isolation, preventing mutable state sharing and data races since an actor is guaranteed to process one message at a time, effectively avoiding concurrent contention during local data accesses.

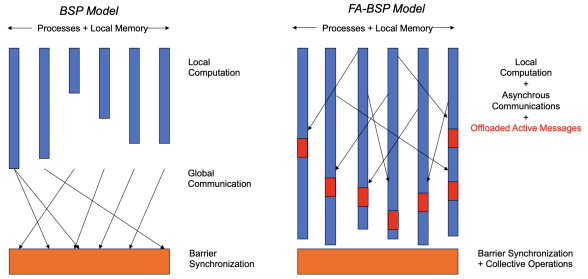

Figure 1: FA-BSP vs. BSP model (graphics adapted from [8])

2.2 k-mer

k-mers are contiguous substrings of length k in a given read where each consecutive k-mer shares an overlap of $k-1$ characters between its prefix and suffix.

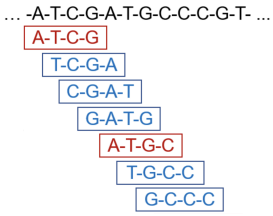

Figure 2: Example of k-mer (for $k = 4$) generation from a toy DNA string. the k-mer in red has been found twice, while others only once.

2.3 Genome Assembly Pipeline

The *de novo* genome assembly pipeline (particularly the Meraculous [2] pipeline) can be summarized in four computational kernels:

(1) Input reading and k-mer counting
(2) Construction of de Bruijn graph
(3) Traversal of de Bruijn graph and 'contig' generation
(4) Scaffolding and final assembly

We explore the utilization of the actor model to redesign and implement efficient parallel programs for kernels 1 and 3.

3 KERNEL 1: k-MER COUNTING

Goal: Given a set of reads, and a parameter k, parse all the reads and count all the distinct k-mers in those reads.

For simplicity, we have selected an MPI-based k-counting implementation from a recent *de novo* genome assembly toolkit [4] and replaced the MPI collectives (line 8, 9 of algorithm 2) with asynchronous actor messages (line 10 of algorithm 1) while keeping everything else the same.

The actor version of kernel 1 is much simpler to implement for a programmer and figure 3 provides clear evidence that the actor implementation counts 1.6× more k-mers per second on average compared to the MPI counterpart, as observed in our experiments on the PACE Phoenix cluster of Georgia Tech [6].

Algorithm 1 k-mer counting - Actor model

1: **Input:** $R_p \leftarrow$ Set of reads stored in process p
2: **Message Handler**
3: **while** k-mers left to process **do**
4: Add the k-mers *kmer_list*
5: **end while**
6:
7: **Main Code**
8: **for** each read $r \in R_p$ **do**
9: **for** each k-mer $k_r \in r$ **do**
10: *actor-send*(k_r, owner_pe(k_r))
11: **end for**
12: **end for**
13: *Global Barrier*
14: Merge all duplicate k-mers in *kmer_list* into (k-mer, count) tuples
15: **Return** *kmer_list*

4 KERNEL 2: DE BRUIJN GRAPH TRAVERSAL AND CONTIG GENEARATION

Goal: Traverse the de Bruijn graph (dBG) and create a collection of paths ('contigs') ensuring each distinct k-mer is present in only one contig.

In our approach, we adopt the use of a distributed hash table to logically represent the entire de Bruijn graph, inspired by the strategy employed in [3]. The nodes are stored as keys in the hash table, and the associated values contain information about their neighbors.

Algorithm 2 k-mer counting - MPI

1: **Input:** $R_p \leftarrow$ Set of reads stored in process p, $b \leftarrow$ buffer size
2: **for** each read $r \in R_p$ **do**
3: **for** each k-mer $k_r \in r$ **do**
4: Insert k_r in *kmers_per_proc*[*owner_pe*(k)]
5: num_kmers $\leftarrow$ num_kmers + 1
6: **end for**
7: **if** num_kmers $\geq b$ **then**
8: Update the send-receive matrix using `MPI_Alltoall`
9: Update *kmer_list* by sending k-mers via `MPI_Alltoallv`
10: num_kmers $\leftarrow$ 0
11: **end if**
12: **end for**
13: Merge all duplicate k-mers in *kmer_list* into (k-mer, count) tuples
14: **Return** *kmer_list*

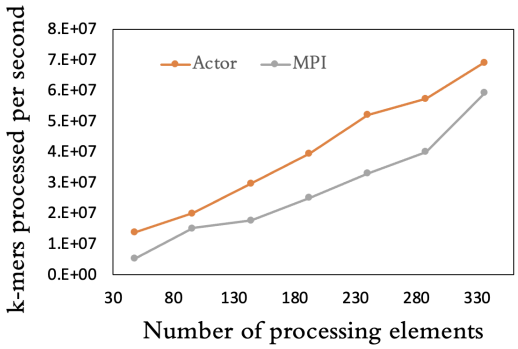

Figure 3: Strong scaling of the actor-based k-mer counting kernel against the MPI implementation on *E.Coli* dataset

Algorithm 3 is a comprehensive redesign of the contig generation process, tailored specifically for the actor model, where remote communication is achieved solely through asynchronous, one-directional, and non-blocking actor messages.

Our theoretical analysis demonstrates that the algorithm 3 converges to the solution within a finite number of iterations of the *while* loop (lines 4 – 12 of algorithm 3). However, our experimental evidence indicates that, for the majority of real-world datasets, the algorithm achieves convergence in just a single iteration.

Figure 4 illustrates the experimental run of the initial implementation of this kernel on Georgia Tech's PACE cluster [6]. The suboptimal strong scaling results from inadequate load balancing of the 'seed' nodes across the processes and the necessity of global synchronizations in the current implementation of the algorithm.

We are actively engaged in improving the theoretical guarantees for a synchronization-free version of our algorithm to ensure its convergence to correct contigs in the worst-case scenario, and will subsequently update our implementation to fix the scaling issues.

Algorithm 3 Contig generation in a de Bruijn graph (dBG)

1: $N_p \leftarrow$ set of nodes of dBG stored locally
2: Every node $\in N_p$ asks it's neighbors for their k-mer counts. (`remote GET`)
3: Every node $\in N_p$ makes a priority order of its neighbors based on received k-mer counts.
4: **while** any node $X \in N_p$ has more than 2 edges **do**
5: X sends pairing `REQUEST` to its most preferred neighbors. (`remote PUT`)
6: X responds to its neighbors in `YES/NO/MAYBE` as `RESPONSE` to their request. (`remote PUT`)
7: **if** `REQUEST` and `RESPONSE` matches **then**
8: X deletes all other edges.
9: **else**
10: X moves on to the next neighbor in its priority order.
11: **end if**
12: **end while**
13: Iteratively update the sub-contigs of a given contig to have the same *contig_id*. (`cyclic remote PUT`)
14: `Parallel sort` the sub-contigs based on their *contig_id*.
15: Combine locally present sub-contigs into the full contigs and add them in the `contig_list`.
16: **Return:** `contig_list`

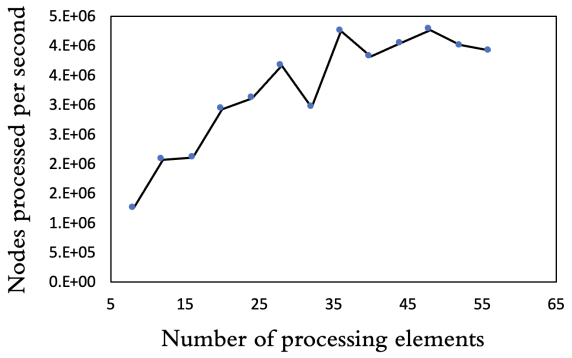

Figure 4: Strong scaling of the contig generation kernel on *E.Coli* dataset

5 FUTURE DIRECTIONS

We plan to incorporate *compiler optimizations* to make the code *locality aware* and minimize the number of actor messages sent across the system. From a theoretical point of view, this strategy will reduce the number of times the message buffer will fill up and send all its messages, hence improving the scalability and runtime.

A natural progression of this work is to write the common communication patterns (remote PUT in our case) as *programmable hardware atomics* that'll allow the program to access the target memory slice directly without involving the message handler of the remote process's actor, making it faster and more scalable on our target hardware.

6 ACKNOWLEDGMENTS

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0083. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government.

REFERENCES

[1] 2022. HClib-Actor Documentation https://hclib-actor.com.
[2] Jarrod A Chapman, Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P Schroth, and Daniel S Rokhsar. 2011. Meraculous: de novo genome assembly with short paired-end reads. *PloS one* 6, 8 (2011), e23501.
[3] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2015. HipMer: an extreme-scale de novo genome assembler. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.
[4] Priyanka Ghosh, Sriram Krishnamoorthy, and Ananth Kalyanaraman. 2020. Pakman: a scalable algorithm for generating genomic contigs on distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1191–1209.
[5] Shams M Imam and Vivek Sarkar. 2014. Selectors: Actors with multiple guarded mailboxes. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*. 1–14.
[6] D PACE. 2017. Partnership for an advanced computing environment (PACE). (2017). http://www.pace.gatech.edu
[7] Sri Raj Paul, Akihiro Hayashi, Kun Chen, Youssef Elmougy, and Vivek Sarkar. 2023. A Fine-grained Asynchronous Bulk Synchronous parallelism model for PGAS applications. *Journal of Computational Science* 69 (2023), 102014. https://doi.org/10.1016/j.jocs.2023.102014
[8] Sri Raj Paul, Akihiro Hayashi, Kun Chen, and Vivek Sarkar. 2022. A Productive and Scalable Actor-Based Programming System for PGAS Applications. In *International Conference on Computational Science*. Springer, 233–247.