Exploring Suffix Array Algorithms in Chapel

Michael P. Ferguson
Hewlett Packard Enterprise
Silver Spring, Maryland, USA
michael.ferguson@hpe.com

Bonnie Hurwitz
Biosystems Engineering
University of Arizona
Tuscon, Arizona, USA
bhurwitz@arizona.edu

Shreyas Khandekar
Hewlett Packard Enterprise
San Jose, California, USA
shreyas.khandekar@hpe.com

***Abstract*—In this project, we implemented algorithms to construct suffix arrays in Chapel, and used them to compute minimal unique substrings and similarity rankings. We developed these algorithms as building blocks for strain detection in metagenomic analysis. With Chapel, we were able to create parallel programs for each of these tasks in short order, and we are seeing good performance with minimal optimization.**

Keywords—Chapel, parallel programming, suffix arrays

I. Introduction

Suffix arrays are used in a variety of bioinformatics applications including genome assembly [1] comparative genomics [2] because of their space efficiency and speed. Suffix array construction is computationally intensive, so building them in parallel is crucial. We used the Chapel programming language to build parallel suffix array algorithms.

II. Suffix Array Algorithms in Chapel

A. Suffix Array Construction

We implemented the general Difference Cover algorithm [3] with a configurable difference cover size. Although other algorithms can be faster in practice [4], we chose this algorithm for our initial effort due to its theoretical parallel speedup and historical effectiveness in [5]. We focused on Difference Cover with v=133 (DC133) as we observed larger v to have better performance. This implementation is available at [6].

In addition to the suffix array, the subsequent steps also require a longest common prefix (LCP) array. For this purpose, we implemented the PAR-PLCP algorithm from [6].

B. Computing Similarity

We have created a new tool that computes all-to-all similarity between a set of documents. This tool creates a suffix array for the input documents and subsequently computes a LCP array as described above. Then, by processing the suffix and LCP arrays in parallel, the tool efficiently computes similarity scores between all pairs of documents [8].

C. Finding Unique Substrings

We developed a tool [9] to compute the minimal unique substrings in parallel based upon the MinUnique-LeftEnd algorithm from [10].

III. Experience Report

Chapel's parallelism and generic programming capabilities streamlined the implementation of computationally intensive tasks, such as suffix array construction, leading to efficient and scalable solutions.

A. Productivity

We developed the suffix array construction, similarity, and unique substrings tools in less than a month in pure Chapel code with no dependencies. The implementations of these components are parallel throughout and use Chapel's data parallel 'forall' loops where possible. The high-level parallelism of 'forall' loops keeps the code concise and ready for multi-node or GPU execution. At times, due to algorithmic constraints, we couldn't use 'forall' loops. In those cases, we used 'coforall' to explicitly assign parts of the problem to specific tasks. In both cases, we found the parallelism easy to express using Chapel's language features.

We found Chapel's generic language features to be a big benefit when writing suffix array construction. We used 'param' features to optimize the code for particular difference cover sizes; created a reusable parallel partition module; and used a generic adapter to partition elements while creating them at some points in the algorithm.

B. Performance

Suffix array construction is the main performance bottleneck in the similarity and unique substrings applications. We observed it to be about 8 times faster than a C++ implementation of DC133 from [5]. Although we have not yet conducted detailed scalability analysis, the initial results showed significant speedup for parallel suffix array construction, with 18x speedup when going from 1 task to 128 tasks on a system with 64 cores and 128 hyper-threads.

IV. Conclusion and Future Work

We are taking initial steps to create fast, scalable, and maintainable software to address challenges in the field of bioinformatics. The Chapel programming language has streamlined our parallel application development. Similarly we look forward to demonstrating productive multi-node parallel computing by extending our suffix array construction algorithm to run in distributed memory with Chapel's distributed array features. Additionally, we want to explore other suffix array construction algorithms.

References

[1] B. Langmead, S. Salzberg, “Fast gapped-read alignment with Bowtie 2,” in *Nat Methods* vol 9, pp 357–359, 2012, doi: 10.1038/nmeth.1923

[2] G. Marçais, A. L. Delcher, A. M. Phillippy, R. Coston, S. L. Salzberg, and A. Zimin, “MUMmer4: a fast and versatile genome alignment system,” in *PLoS computational biology*, vol 14 no 1, 2018

[3] J. Kärkkäinen, P. Sanders, and S. Burkhardt, “Linear work suffix array construction,” in J. ACM vol. 53, no. 6, pp 918–936, Nov. 2006, doi: 10.1145/1217856.1217858

[4] G. Nong, S. Zhang and W. H. Chan, "Two efficient algorithms for linear time suffix array construction," in *IEEE Transactions on Computers*, vol. 60, no. 10, pp. 1471-1484, Oct. 2011, doi: 10.1109/TC.2010.188

[5] M. P. Ferguson, “FEMTO: fast search of large sequence collections,” CPM 2012: Combinatorial Pattern Matching 2012, in Lecture Notes in Computer Science, vol 7354, doi: 10.1007/978-3-642-31265-6_17

[6] M. P. Ferguson, “Chapel language implementation of suffix array construction within FEMTO,” [Online]. Available: https://github.com/femto-dev/femto/blob/main/src/ssort_chpl/SuffixSortImpl.chpl

[7] J. Shun, "Fast parallel computation of longest common prefixes," *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, USA, 2014, pp. 387-398, doi: 10.1109/SC.2014.37

[8] M. P. Ferguson, “Chapel language implementation of document similarity within FEMTO,” [Online]. Available: https://github.com/femto-dev/femto/blob/main/src/ssort_chpl/SuffixSimilarity.chpl

[9] M. P. Ferguson, “Chapel language implementation of finding unique substrings within FEMTO,” [Online]. Available: https://github.com/femto-dev/femto/blob/main/src/ssort_chpl/FindUnique.chpl

[10] L. Ilie and W. F. Smyth, “Minimum unique substrings and maximum repeats,” in Fundam. Inf., vol 110, pp. 183-195, January 2011