BIOINFORMATICS

Vol. 00 no. 00 2015
Pages 1–2

Probabilistic cardinality estimation for fun and ~~profit~~ SCIENCE!

Luiz C. Irber Jr. [1,*], C. Titus Brown [2,1,3]

[1]Department of Computer Science and Engineering, Michigan State University, East Lansing 48823,
[2]Department of Microbiology and Molecular Genetics, Michigan State University, East Lansing 48823,
[3]School of Veterinary Medicine, UC Davis, Davis 95616, United States of America

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: XXXXXXX

ABSTRACT

We present an open implementation of the sublinear memory HyperLogLog cardinality estimation algorithm for counting fixed-length subsequences of DNA ("k-mers").

Summary:

Availability and implementation:

The HyperLogLog implementation is in C++ with a Python interface, and is distributed as part of the khmer software package. khmer is freely available from https://github.com/ged-lab/khmer under a BSD License. The features presented here are included in version 1.4 and later.

Contact: irberlui@msu.edu

1 INTRODUCTION

Next-Generation DNA Sequencing (NGS) technologies generate data at increasing rates, and for almost 10 years have been increasing data generation capacity at rates faster than Moore's Law. In the last five years, several probabilistic data structures and algorithms have been developed and applied to this increasing volume of data.

Probabilistic data structures are useful when the cost for a exact answer is prohibitive, but an approximate answer is acceptable. The approximation is usually attained through reproducible randomness (hashing, for example) and average case analysis. The main benefit of probabilistic data structures is a tradeoff between space and accuracy, since often a very small amount of memory can be used for crude results but better estimatives are reported as memory increases. These data structures are usually specialized, performing one kind of operation well but missing other functionality available on generalized data structures.

Here we present an open implementation of the HyperLogLog cardinality counting algorithm, specialized for k-mers, fixed-length subsequences of DNA strings. The HyperLogLog counter is a cardinality estimation data structure with constant (and low) memory footprint, based on hashing and probabilistic estimation. (@Brief description.) See Figure 2 for an example.

This functionality is useful for a variety of purposes, including estimating the minimum required memory allocation for fixed-size data structures such as Bloom filters and Count-Min Sketches (ref (Pell *et al.*, 2012), (Georganas *et al.*, 2014), (Zhang *et al.*, 2014)).

*to whom correspondence should be addressed

2 METHODS

We implemented HyperLogLog for k-mers on top of the khmer library. khmer is a library and suite of command line tools for working with DNA sequence. It implements k-mer counting, filtering and graph traversal using probabilistic data structures that include Bloom Filters and Count-Min Sketch. Building on top of khmer leveraged leveraged the existing infrastructure (read parsers, package installation, API signatures and some k-mer hashing methods).

(Briefly describe HLL implementation, plus problem decomposition)

2.1 The "add element" operation

This operation involves the calculation of a hash value for the input, some bitwise operations to determine special properties (longest run on 0-value bits, for example) and a memory update to one position of the bit array. This is the CPU-intensive part, depending heavily on the hash function used. I chose MurmurHash3 because it is one of the fastest non-cryptographic hash function available and it has a reasonably uniform hash space distribution. This is executed $len(read) - (k - 1)$ times for each read on the dataset, were k is the desired k-mer size.

2.2 HyperLogLog merge operation

The merge operation is a elementwise max-reduction between two bit arrays. Because the bit arrays are relatively small (about 128 KB) the merge operation is an excellent way to avoid resource sharing and synchronization during the update operations, at the cost of instantiating additional temporary HyperLogLog structures. Since their sizes are small, this is a viable tradeoff. This operation is not necessary when adding elements in parallel and can be executed optimally after all elements were consumed (i.e. once at the end).

2.3 Problem decomposition

One way to divide the problem is by instantiating multiple HLL counters and distribute reads between them while there are reads

available. After all the reads are consumed the counters are merged and the final counter can be used for cardinality estimation.

I chose a shared memory implementation for this problem decomposition, since only a small amount of memory is needed and this is also the architecture most khmer users have available for use. OpenMP was chosen because it doesn't demand code changes (meaning the program will still work if OpenMP is not available, although slower). This is also important because it is not available currently on the default OSX compiler (clang), but as soon clang implements OpenMP support it will work for khmer users. This would be harder (but not impossible) to implement with MPI.

The only shared resource during updates is the bit array. In order to avoid any synchronization instead of having one HyperLogLog shared between threads (and so a critical section or atomic update of the bit array position) I opted for creating one HyperLogLog data structure for each thread, this way no resources need to be shared.

OpenMP tasks maps well to this decomposition: one thread (using a single pragma) get reads from the read parser and for each read a task is spawned, using firstprivate(read) to guarantee the read won't be overwritten. The task thread does the "add element" operation in the counter assigned to its thread ID. After all reads are parsed and the tasks finish one thread does the merge operation over all counters.

Since the sequential HyperLogLog implementation is CPU-bound (limited by hashing), optimized input reading is not a priority.

3 DISCUSSION

Running with only one CPU is suboptimal because the problem can be easily parallelized.

For comparison we ran a simple benchmark using the same input infrastructure, but without performing any kind of processing to check what is the I/O lower bound. This baseline can be used to verify how many threads are needed before I/O becomes the bottleneck.

Figure **??** shows $t = 32$ threads are needed to saturate I/O.

Memory consumption increases proportionally to the number of threads used, since one HyperLogLog is instantiated for each thread. At the end of input processing this memory is released and only one is needed, being the result of merging every HyperLogLog holding partial values from each thread. This temporary increase in memory consumption is not a practical problem because each HyperLogLog is small: Figure **??** shows the maximum resident memory for pure I/O is about 25 MB,

Another memory overhead associated with increasing the number of threads is the memory OpenMP allocations for its task queue, which depends on the compilers' implementation. Usually this is not so drastic because the thread doing input parsing can switch to sequence processing if the queue is full, and in I/O saturation conditions the queue will be close to empty anyway.

I implemented parallelization in shared memory using OpenMP. Shared memory parallelization is useful because the most time consuming step is calculating the hash and this doesn't share state among other calculations. The critical operation is updating the bit arrays, which is fast and just modify a small amount of memory.

The target architecture is multicore CPUs. The primary users of khmer are biologists and one of the project goals is easy installation and a low cognitive barrier for new users. Compiling in a consistent way for Xeon Phi or GPUs is non-trivial in most systems, and even OpenMP is not supported in some popular platforms (OSX + clang, for example). Nonetheless, adapting the code to use either Xeon Phi or GPUs could lead to even better results, since the hashing process is mostly CPU-bound.

I used two different datasets during development, one being a subset 3 orders of magnitude smaller than the other:

Gallus_3.longest25.fasta

- 149,943,923 bp - 44,336 seqs - 3,382.0 average length - 144 MB

Despite being smaller, the average length of each sequence is about the same for both datasets. I used the partial dataset just to make quick tests (specially when segfaults were envolved).

The true cardinality of the complete dataset for $k = 32$ is 129,196,601. This is the timing data for the complete dataset (all tests with k=32):

Speedup times are close to linear, which is best seem on Figure 3. We can also notice the estimation is within error bounds, being smaller than 1

For stress testing I used a larger dataset. Although it's 4 times smaller than the one I proposed to use, it is a typical dataset found by users.

Chicken_10Kb20Kb_40X_Filtered_Subreads.fastq

- 43,076,933,303 bp - 9,006,923 seqs - 4,782.6 average length - 81 GB

Using the Python API, with 16 threads and k = 32, the running time is close to what is expected when extrapolating the results on smaller datasets, (about 36 minutes).

4 CONCLUSION

ACKNOWLEDGEMENT

Funding: This work was supported by FUNDING-AGENCY [GRANT-NUMBER].

REFERENCES

Georganas, E., Buluç, A., Chapman, J., Oliker, L., Rokhsar, D., and Yelick, K. (2014). Parallel de bruijn graph construction and traversal for de novo genome assembly. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 437–448. IEEE.

Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J. M., and Brown, C. T. (2012). Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, **109**(33), 13272–13277.

Zhang, Q., Pell, J., Canino-Koning, R., Howe, A. C., and Brown, C. T. (2014). These are not the k-mers you are looking for: Efficient online k-mer counting using a probabilistic data structure. *PLoS ONE*, **9**(7), e101271.

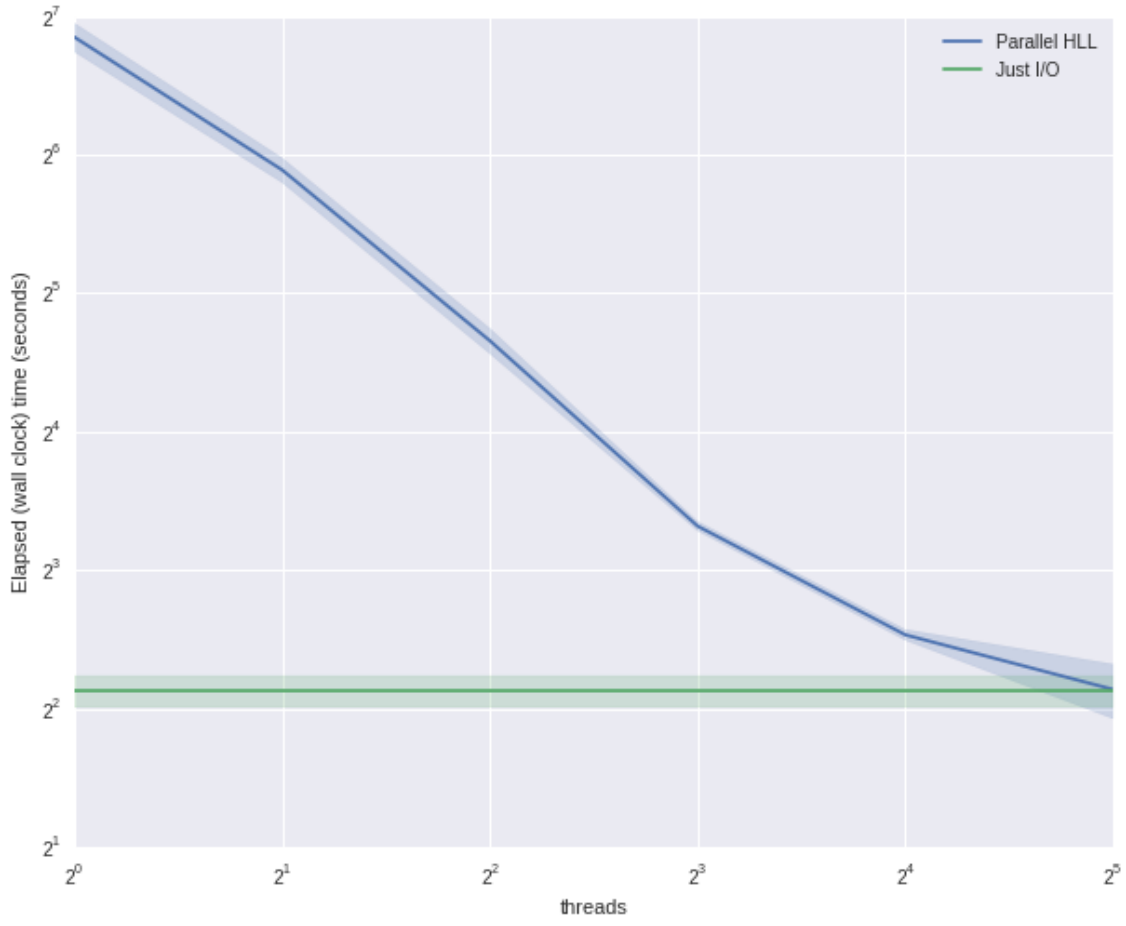

Fig. 1. Walltime and lower bound(I/O)

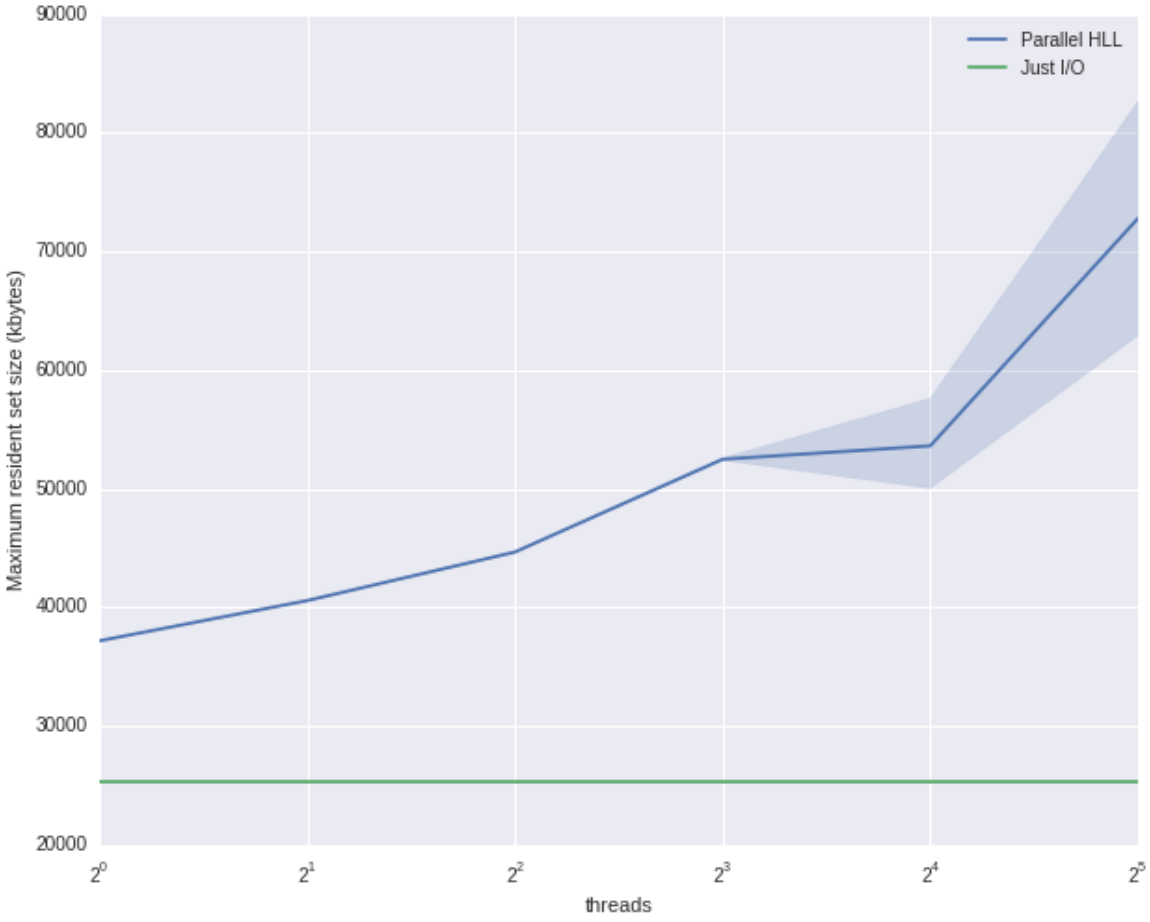

Fig. 2. Memory consumption