BIOINFORMATICS

Vol. 00 no. 00 2015
Pages 1–3

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

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

[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
Summary:
Availability and implementation: khmer is freely available from https://github.com/ged-lab/khmer and under a BSD License. The features presented here are included since version 1.4.
Contact: irberlui@msu.edu

1 INTRODUCTION

Next-Generation Sequencing (NGS) technologies generate data at increasing rates, and for almost fifteen years at an even faster rate than Moore's Law. Algorithms need to be adapted or even developed to deal with these data rates, and depending on which analysis is intended probabilistic data structures can be used to manage the complexity.

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 is space efficiency, since a very small amount of memory can be used for crude results but better estimatives are reported as memory increases. They are usually specialized, performing one kind of operation well but missing other functionality available on generalized data structures.

An example of a probabilistic data structure is a Bloom Filter, which is useful for set membership operations. They basically work by creating a large bit array of m bits and using k hash functions to update each position of the array. To insert a new value to a Bloom filter the hash functions are calculated and the hashed values are used to set positions on the bit array to 1.

After an element is inserted it can be queried for membership on the dataset, which means calculating the hash for a value and checking if all the 1-bit are set in the bit array.

To avoid saturation and returning false positives more memory can be allocated (by increasing m), but ultimately the amount of memory needed to avoid false positives depend on the cardinality of the set (how many unique elements are present). Usually this is not known beforehand, and since there is no way to remove elements or extend a already started Bloom filter the computation needs to be restarted if the false positive rate gets too high.

One way to avoid this is by estimating the cardinality of the set, and then initializing a Bloom filter with the proper size. The HyperLogLog counter is a cardinality estimation data structure with constant (and low) memory footprint, also based on hashing and probabilistic estimation. It works by a similar process to Bloom filters, but with a different way to update a bit array. See Figure 2 for an example.

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 such as Bloom Filter and Count-Min Sketch. I have a sequential HyperLogLog counter implementation in a khmer branch, and it was used as baseline. This also allowed leveraging the existing infrastructure (read parsers, package installation, API signatures and some k-mer hashing methods).

khmer is implemented both in Python and C++. Data structures and performance-critical sections are implemented in C++ and this low level API is exposed to Python, with scripts providing the functionality available to users. The Python interpreter is implemented using a global interpreter lock (GIL), which makes multithreading virtually impossible with pure Python. Usually multiprocessing is used instead, but there is a communication overhead (since memory is not shared between processes). C/C++ extensions are not affected, and so OpenMP is viable.

There is (Georganas *et al.*, 2014), an article accepted for Supercomputing 2014 describing the implementation of a parallel HyperLogLog counter. They use C++ and MPI, and also MurmurHash as hash function, the same used for my sequential implementation. They use cardinality estimation to set Bloom filters, which are used to build de Bruijn assembly graphs.

2 APPROACH

3 METHODS

3.1 Input parsing

The most popular data format for sequences is FASTA, a human readable text-based representation. It is very easy to parse, but also very inneficient

*to whom correspondence should be addressed since it doesn't allow any direct form of indexing or random access. Usually files are processed into more efficient formats before being used, although it might not be a viable option: sequence datasets tend to be very large, and many systems don't have enough disk space to hold both original data and the processed file. I used the read parser available in khmer, and it can process both FASTA and FASTQ (an extended FASTA file with quality scores), either compressed or uncompressed.

3.2 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.

3.3 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).

3.4 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.

4 DISCUSSION

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.partial.fasta

- 112,455 basepairs - 47 seqs - 2392.7 average length - 111 KB

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):

```
  $ export OMP_NUM_THREADS=1
  $ time ./hll ../../Gallus_3.longest25.fasta
  129,388,424
  real    0m52.660s user    0m52.505s sys     0m0.062s

  $ export OMP_NUM_THREADS=2
  $ time ./hll ../../Gallus_3.longest25.fasta
  129,388,424
  real    0m27.193s user    0m54.142s sys     0m0.082s

  $ export OMP_NUM_THREADS=4
  $ time ./hll ../../Gallus_3.longest25.fasta
  129,388,424
  real    0m14.042s user    0m55.790s sys     0m0.093s

  $ export OMP_NUM_THREADS=8
  $ time ./hll ../../Gallus_3.longest25.fasta
  129,388,424
  real    0m7.370s user    0m58.318s sys     0m0.084s

  $ export OMP_NUM_THREADS=16
  $ time ./hll ../../Gallus_3.longest25.fasta
  129,388,424
  real    0m3.773s user    0m59.251s sys     0m0.094s


$ time ./just_io ../Gallus_3.longest25.fasta.bak
real    0m4.149s
user    0m4.076s
sys     0m0.065s

$ time OMP_NUM_THREADS=16 ./hll ../Gallus_3.longest25.
129388424
real    0m4.722s
user    1m15.223s
sys     0m0.109s
```

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).

5 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.