JULIA-ENABLED COMPUTATION OF MOLECULAR LIBRARY COMPLEXITY IN DNA SEQUENCING

Larson Hogstrom, Mukarram Tahir, Andres Hasfura
Massachusetts Institute of Technology, Cambridge, Massachusetts, USA

18.337/6.338 Parallel Computing
Final Report
December 13, 2015

ABSTRACT

As the throughput of DNA sequencing increases, major research facilities are performing genomic profiling on tens of thousands of samples per year with a single genome often exceeding 400GB of raw data. Julia's library of mathematical packages, its speed in numeric computing, and functionality to support operations performed in parallel are likely suited to many challenges in genomic processing and analysis. To test this hypothesis, we used Julia to implement an existing modeling strategy to predict a commonly observed sequencing artifact [Daley, 2013] in order to improve predictions for return on investment for performing new sequencing experiments for individual genomes. Our results provide insight into Julia's suitability for genomic analysis including rapid prototyping, error analysis with C++ code, and a new parallel algorithm for the identification of duplicate sequencing reads.

I. INTRODUCTION

Factors affecting computer language design and selection in computational biology have been discussed at length in recent years (Gentleman, 2004 & Nair, 2005). Historically, the field's most frequently used and computationally demanding tools, including algorithms for DNA alignment and sequence similarity queries, have been implemented in low-level languages such as C, C++, or Fortran. The ability to compile such tools into highly efficient machine code has provided obvious performance benefits, but more recent scientific and computational demands in biology have transitioned, driving many scientists to begin using high-level and dynamically typed computer languages. This transition has been driven, in part, by the flexibility offered by high-level languages in response to fast-paced changes in data collection and data structures that arise with new experimental techniques from molecular biology. Rapid

development of new tools and data types have put a heavy emphasis on prototyping, ease of development cycles, and code maintainability (Nair, 2005). For example, the Perl computer language gained popularity in bioinformatics during the 1990s for its syntactic brevity, handling of regular expressions, and support of both procedural and object-oriented programing. More recently, computational biologist have relied heavily on high-level languages including R and python to carry out cycles of analysis and processing. The community of researchers developing new capabilities in these languages often place emphasis on functionality that increases the ease of data handling and statistical modeling. The Julia language, with strength in numeric computing, is poised to build on the best parts of other high-level languages by extending performance and code interpretability when working with very large data sets observed in DNA sequencing studies. A few of Julia's most important design features include the use of multiple dispatch, just-in-time compilation, and metaprogramming.

Here we apply the Julia in extracting and analyzing frequency information from raw DNA sequencing data. Our efforts are aimed at addressing an artifact observed in popular DNA sequencing strategies that occurs when a small fraction of sequenced regions are duplicated at high frequency. Such region duplicates are problematic because they represent non-independent measurements of the underlying genome and ultimately waste storage and compute resources. Improved models to predict the impact of sequence duplicates have recently been described [Daley, 2013], but the technical infrastructure to run this computation at scale is currently lacking. Our project consists of new Julia functionality to launch and manage duplicate marking events in sequencing files. We also created Julia wrappers to manage existing C code [Daley, 2013] predicting return on investment for performing new sequencing experiments for individual genomes using methods based on the Good Toulmin model.

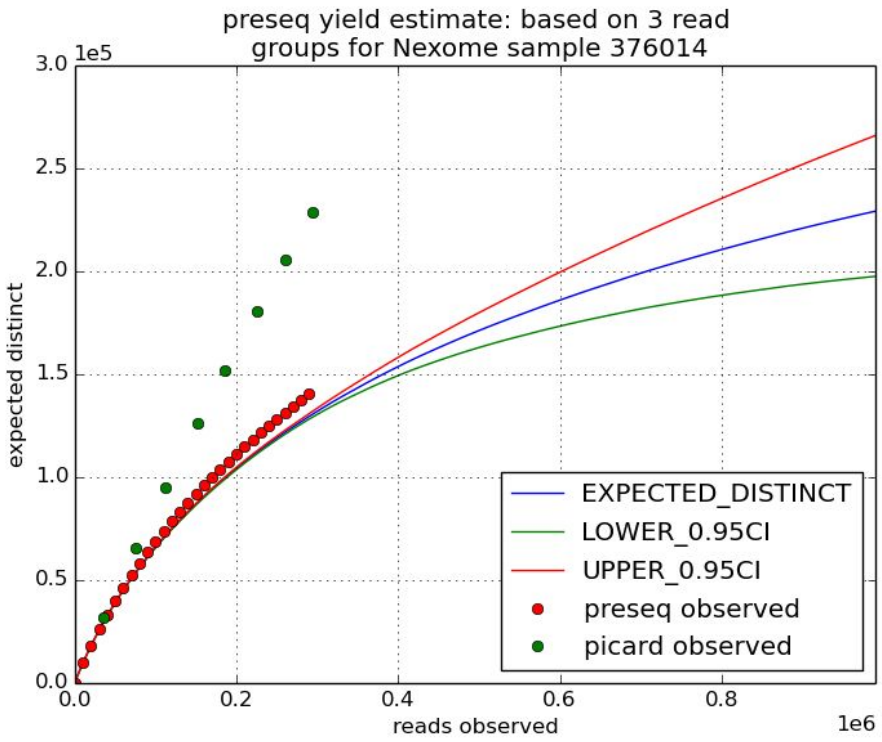

Figure 1. Representative model output for a single sample processed with the preseq.jl tool. The goal of this method was to improve predictions of information acquisition as sequencing studies proceeds.

II. APPROACH

We began by benchmarking the existing preseq C++ code by computing the complexity curve for a series of alignment data (BAM) files. This was applied to 50 exome sequencing samples, allowing us to perform timing profiles and identify bottlenecks. Through a combination of profiling efforts, we mapped the procedure through which preseq generates the complexity curve for a given BAM file input. First, POS and FLAG entries are extracted for each read from the binary BAM file, which respectively correspond to the position of a read on the reference genome and an integer value from which properties of a read (such as whether it is a primary read or if it is mapped to the reference) can be deduced through bitwise operations. Preseq then filters the reads based on their FLAG values, and then proceeds to identify duplicates among these reads. Two reads are classified by preseq as duplicate if their position on the reference genome (signified by POS) are identical, and preseq maintains a count of these duplicates. These so-called read counts are then used for calculating the complexity curve, which is the output of the code. This procedure is summarized in the schematic shown in Figure 2. Of these steps, we determined that approximately 95% of all compute time was expended in loading, filtering, and counting duplicate reads in the

BAM sequencing file. Given the near embarrassingly parallel nature of these tasks, we recognized the opportunity to re-implement them in Julia and attempt to achieve speedup through its parallel computing capabilities. Once the read counts are calculated, the C++ preseq code can be called with these pre-calculated counts (rather than the original BAM file) for the final complexity curve calculation.

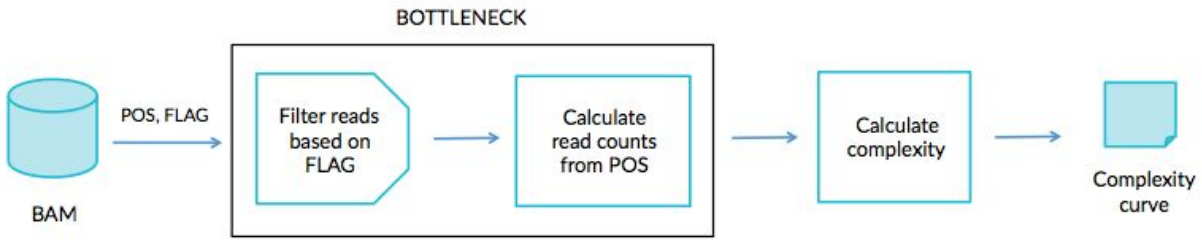

Figure 2. Overview of processing and modeling strategy used by preseq. We identified read filtering using FLAG and read counting using POS as the bottleneck best suited for optimization and parallelization in Julia.

III. PARALLEL ALGORITHM FOR READ COUNTS

The product of our efforts to parallelize the preseq.cpp code was a Julia wrapper preseq.jl, which handles all parts of the computation except the final calculation of complexity from read counts. The input to preseq.jl is a binary BAM file, but unlike preseq.cpp, this cannot be directly read for FLAG and POS values as there is currently no Julia library for interfacing with BAM alignment files. We therefore utilize an external executable known as samtools for converting the binary BAM file to a plaintext SAM file, which is then read into a DistributedArray object. When worker nodes are spawned in Julia, each node then has access to only a portion of the SAM entries for further processing. These worker nodes proceed to extracting POS and FLAG fields from their given set of SAM entries, and then apply filters based on FLAG in a manner that is identical to the original C++ code. The filtered reads are then examined for duplicates, and a count of duplicates is maintained at each worker node. Given that a duplicate may be present across multiple worker nodes, the individual count arrays from the worker nodes are examined for overlap once they are returned to the master node. The final counts are then written to a temporary file on disk, and preseq.cpp is called with this count file (rather than the BAM file) for a much faster complexity curve calculation. The overall implementation is summarized in Figure 3.

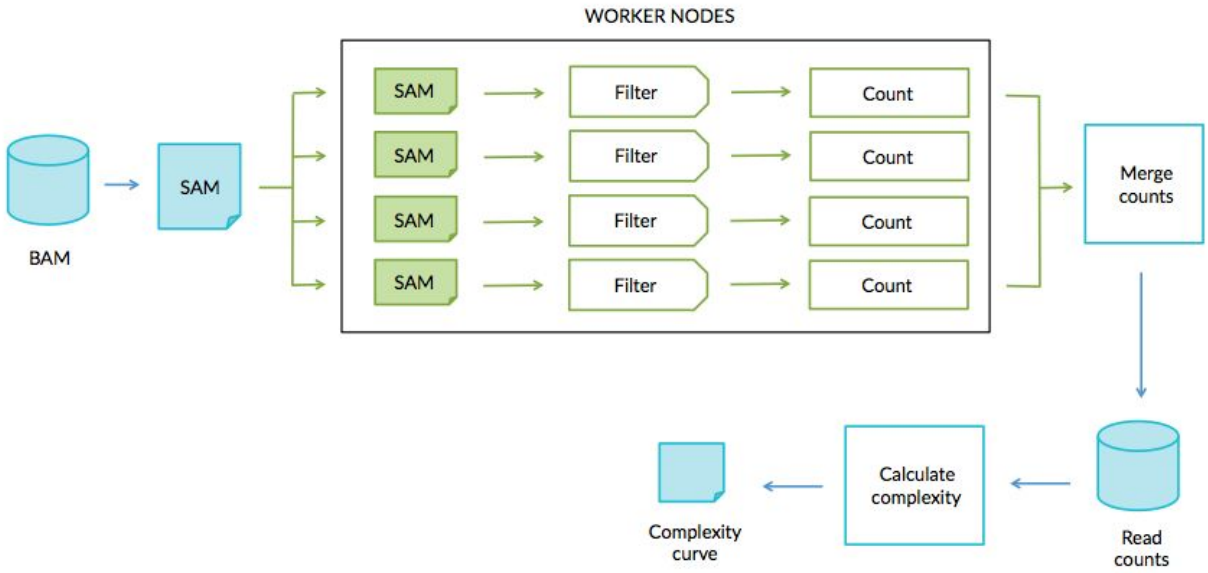

Figure 3. Schematic of the preseq.jl, a Julia wrapper that we implemented for complexity curve calculation from a BAM file input. Julia worker nodes perform the filtering and read count calculations in parallel, and the result is offloaded to the existing preseq.cpp code for the final complexity calculation.

From our earlier benchmarking of the serial preseq.cpp code, we determined execution times of approximately 5, 7, and 12 seconds respectively for 35 Mb, 67 Mb, and 135 Mb BAM files. For comparison, we ran our Julia wrapper preseq.jl on these three input file sizes, and varied the number of cores to determine reduction in execution time as the number of cores is increased. Figure 4 shows a plot of these execution times, and indicates a substantial speedup as the first few cores are added. A plateau in speedup is quickly reached, but the plateau varied according to file size. Given that production BAM files are of much larger sizes, we expect efficient consumption of a large number of cores before such a point of diminishing returns is reached. It is concerning though, that the execution time at which these curves level off is still higher than the execution time of the serial preseq.cpp code for the corresponding input file size. To investigate this, we visualized the various components of the execution time for the 135 Mb input file, as shown in Figure 4 (lower). We notice immediately that filtering and counting reads, which were subject to parallelization, contribute to the majority of the speedup as the number of cores is increased. However, there appears to be a constant and substantial overhead associated with reading the BAM files in preseq.jl that is certainly absent in the preseq.cpp code. Unlike the preseq.cpp code, which is able to directly access POS and FLAG fields for all entries using a C++ API that interfaces with BAM files, our preseq.jl code uses an external executable (samtools) to first convert the binary BAM file to a plaintext SAM file, and then extract POS and FLAG fields from each

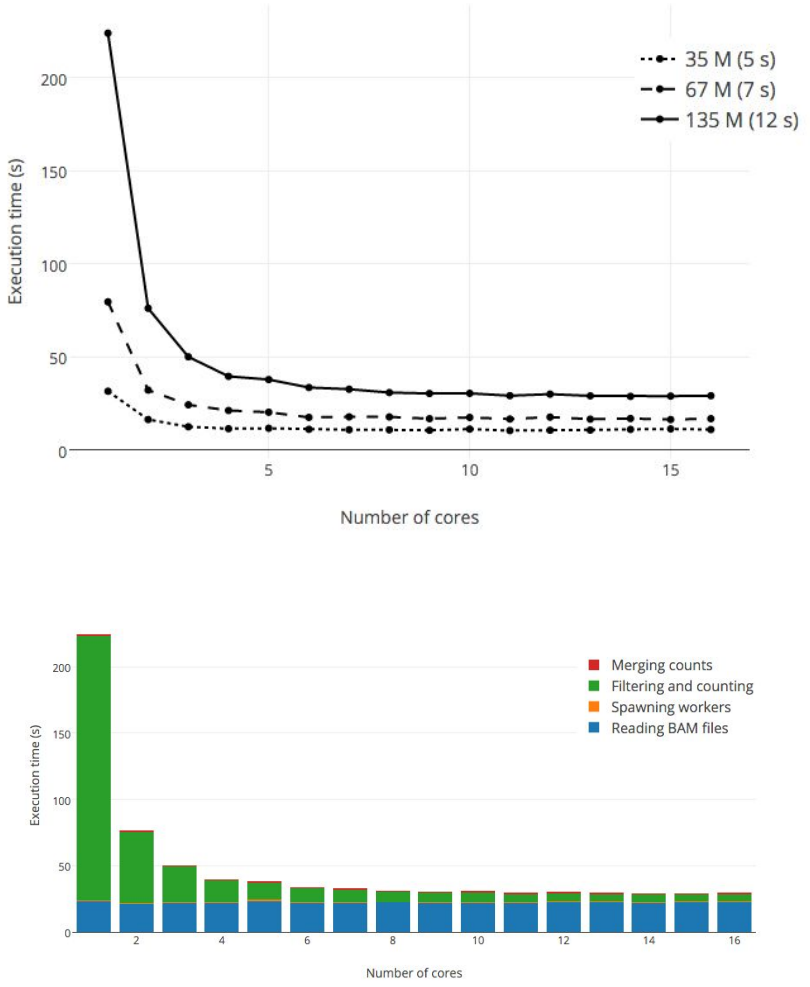

Figure 4. (upper) Execution time of preseq.jl as a function of the number of parallel cores for three input BAM file sizes. For comparison, the execution time of the serial preseq.cpp code is shown in parenthesis in the legend. (lower) Breakdown of the execution time for the 135 Mb BAM file, demonstrating excellent performance improvement in read count calculation as the number of cores is increased, but significant overhead incurred from reading the BAM file to a plaintext SAM file in memory.

plaintext entry. This leads to a massive performance loss in our Julia wrapper (compared to the original serial code). This is not especially concerning, however, since a Julia library for interfacing with BAM files currently appears to be under development, and in future iterations of this code, we hope to utilize it in our Julia wrapper to eliminate this bottleneck associated with reading BAM files.

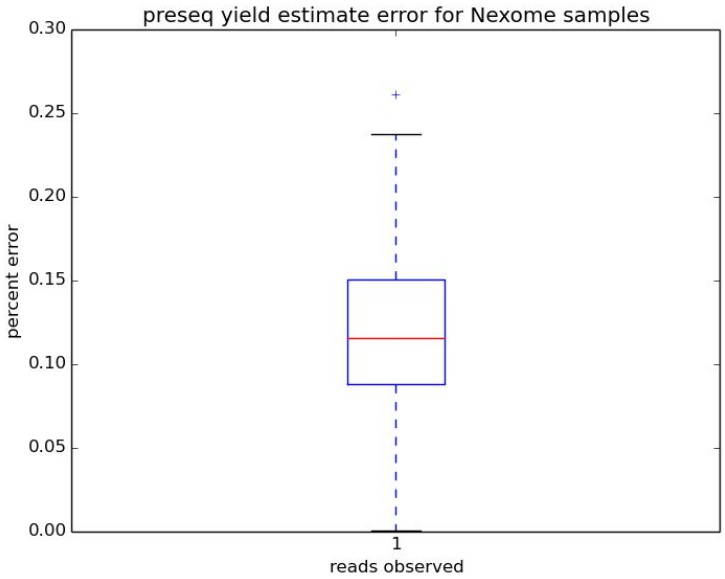

Figure 5. Rates of model prediction errors were calculated for 50 nexome samples.

IV. ERROR ANALYSIS

We evaluated model accuracy of the Julia-based preseq output. The preseq.jl modeling effort yielded a median error of 12% in predicting the number of unique molecules observed in the 50 exome sequencing studies examined (Figure 5). In this analysis, we used 3 read groups out of 16 in each sample to train the preseq model. The extrapolation of expected unique reads was calculated by preseq and compared to the true counts of unique reads observed in the full sequencing files. These results were promising as they show an improvement over existing strategies that have been applied to our working set of exome samples. Additionally, we assessed the discrepancy of read counting after implementation in Julia as compared with C++. We observed differences which were a small fraction of total counts performed and discrepancies were reduced as more reads were read in.

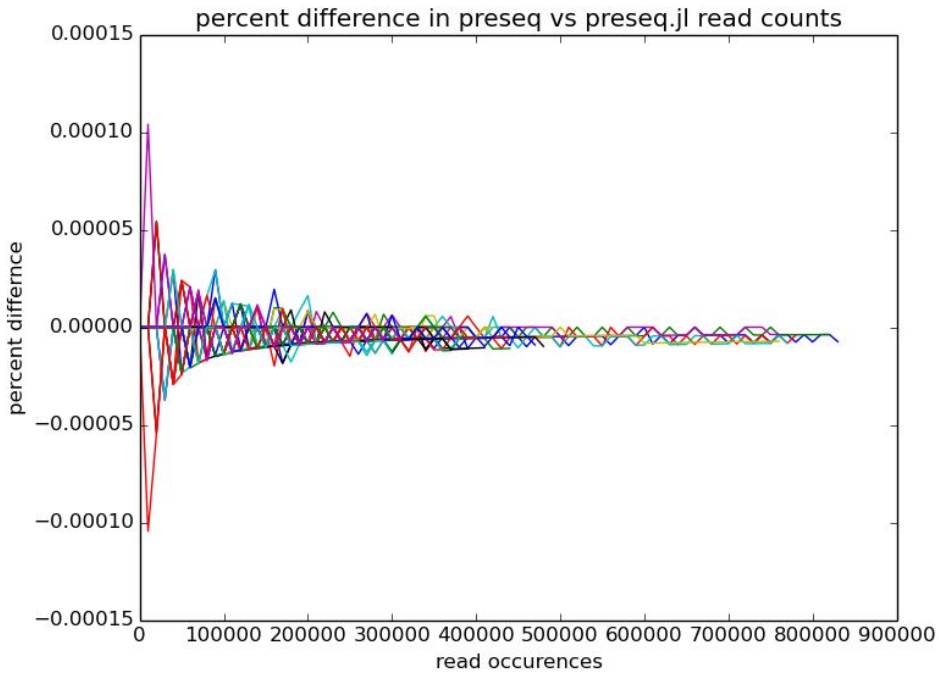

Figure 5. Small differences in read counts were observed when comparing results from Julia vs. C++ implementation. The effects of these errors were generally small and diminished as more reads were counted from each sequencing file.

V. EVALUATING JULIA OVERHEAD AND STARTUP

Using Julia allowed for simple parallelization of the read count calculation, but it also introduced unexpected overhead issues. We found that a few basic operations of the Julia language took longer to accomplish than similar operations in more mature languages, such as python and C. In this section, these time consuming operations are discussed and benchmarked to show the effect on the preseq.jl package.

In this section, all speed tests were performed using a 13-Inch Macbook Air with a 1.3 GHz Intel i5 processor and 4GB of DDR3 RAM. One of the first things noticed when performing read counts in Julia was the Julia start time. Without user specified options, the average time on the above setup was found to be 1.015 seconds. This start up time continues to increase as users specify how many cores to start Julia with. Figure 6 shows the relationship between startup time and the number of cores specified on startup.

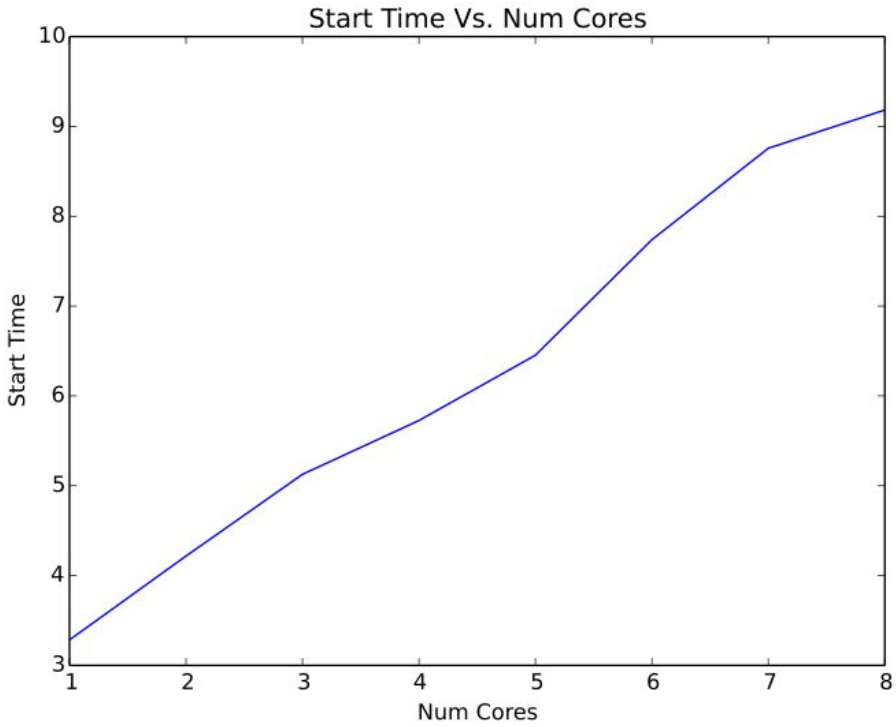

Figure 6. Julia's startup time versus the number of cores specified at startup time.

Preseq.jl also imports packages built by the Julia community, which slowed the runtime noticeably. Of the packages imported by the preseq.jl library, the most important to the parallel speedup of read counts is DistributedArrays.jl, which allows computation on arrays to be split over multiple processors and finally merged once each processor's portion of the array has been completed. To import this package alone cost over a second, which is very expensive considering given enough cores a 135MB BAM file can be processed in around 10 seconds.

Because of the cost of importing this one package, standard import times were also studied and benchmarked. Many common modules were imported and timed, and the following histogram was produced from the results.

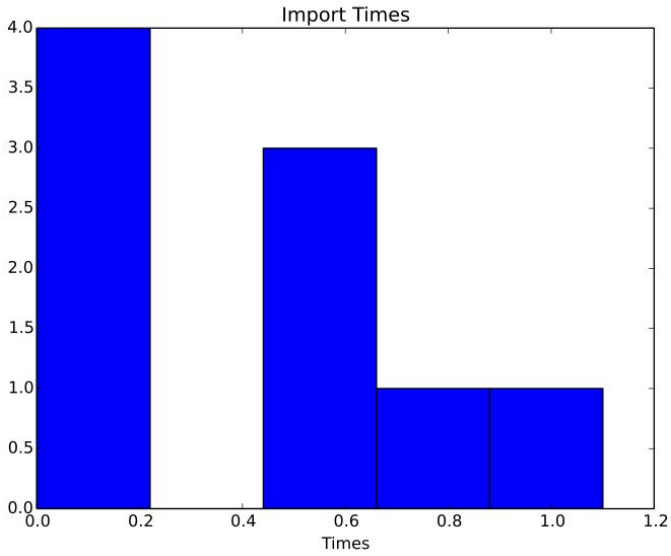

Figure 7. Histogram of import times of different common modules, including DistributedArrays.jl, a crucial package for the parallelization of the read count computation.

We also examined I/O costs as an important function of Julia's performance in the preseq.jl tool. Our algorithm relied on the writing of an intermediate read counts file. We benchmarked the time required by Julia to perform I/O as compared to C++, in order to determine whether the additional I/O would prove costly in our efforts to create a speed up. Below are read and write costs between C++ and Julia for different sized reads and writes. Figure 8 shows the I/O time between Julia and C++ is quite similar. This shows that there is no major cost of using Julia to perform the writing of an intermediate read count file.

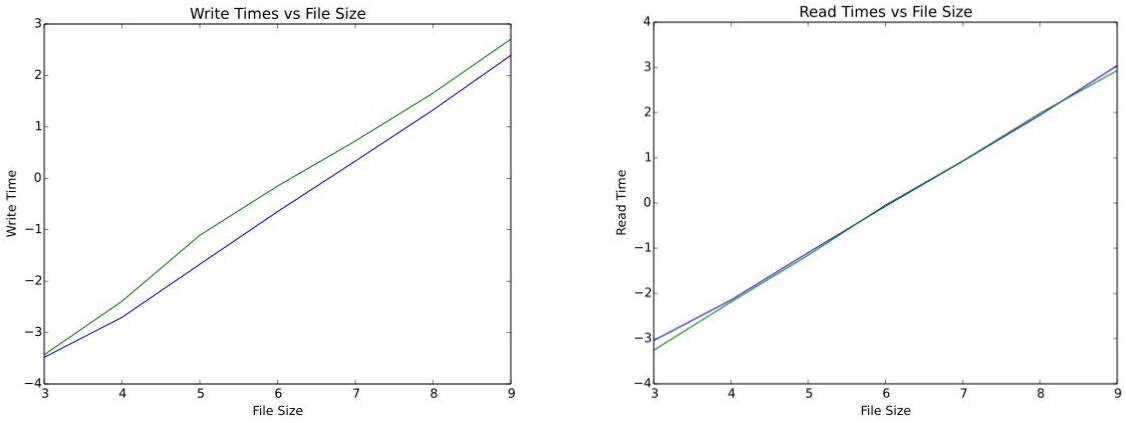

Figure 8. Log log (base 10 bytes) plot of the time to write different file sizes between Julia and C++ (left). Log log (base 10 bytes) plot of the time to read different file sizes between Julia and C++ (right).

VI. CONCLUSION

Predicting the molecular complexity of a genomic sequence library is a time consuming part of genomic analysis. This project tackled this issue by parallelizing a popular molecular complexity prediction tool, preseq.cpp. We were able to benchmark the parallelized preseq on production servers, and display a greater than 10X speedup. One limitation of the current project was the lack of a Julia-specific tool for interfacing with BAM alignment files. Our results show that reading of the binary sequencing file is now a bottleneck in the Julia-based read counting algorithm. The construction of a file reader was beyond the scope of our project, but efforts are ongoing in the Julia community to enable Julia to read and filter binary files directly. This would eliminate unnecessary BAM files handling and greatly speedup preseq.jl.

VII. REFERENCES

Daley, T., and Smith, A.D. "Predicting the molecular complexity of sequencing libraries." Nature methods 10.4 (2013): 325-327.

R. Gentleman, et al. “Bioconductor: open software development for computational biology and bioinformatics” (2004)

Nair, M. Niedner, R.H, Grbbskov, M. “Perl in bioinformatics”. (2005) DOI: 10.1002/047001153X.g409321