GenomiK: A parallelized web approach to genome assembly

Blaine Muri ~ Jordan Matelsky ~ Michael Stewart

Abstract

Using Meteor, a Node.js framework for exposing public-facing websites, and Go, a modern language designed to improve concurrency support, our GenomiK framework allows the user to upload a FASTA genome, upload reads, and assemble reads with a reference genome. The algorithms output the results at each step within the process. The core of the framework is where the advantage lies, in that we are able to take advantage of Go to run concurrent operations on a multi-core server. When combined with a speedy web server[1], we lower the data-storage requirements on the user's local machine — instead, the data can be retained in cloud storage. We also increase the speed of most operations by rewriting conventionally time-complex algorithms in concurrent Go.

Introduction

A large problem with the rising genomic industry is the space requirements of many of the sequences as well as the time-complexity many of the algorithms. Users attempting to make early forays into the world of genomics frequently lack the necessary resources (e.g. processor speed, storage, RAM). We were able to solve this problem by offering large quantities of storage and processing power, made cheap by Amazon EC2 cloud architecture.

Another common concern when considering collaborative online environments is the issue of privacy. Users expect their own private machines to remain out of the public domain until publication (in the case of research) or distribution (in the case of industry). We solve this problem by allowing users to create both public and private assemblies within the application, offering both the ability to contribute to global research and the ability to keep personal research private.

[1] We use Amazon EC2's m3.medium instance, though we've designed all of our software to scale with its host hardware with no extra deploy effort.

It is true that, when running multiple assemblies on the same server, our hardware is under a tighter resource constraint than a single machine running a single assembly. To prevent this from being a roadblock to research and scientific performance, we take advantage of Meteor's asynchronous system-call functionality to distribute calls to our Go backend through time. That is, with no extra work for the user, a FASTA file that requires large amounts of resources to assemble will automatically be queued to prevent it from blocking other smaller jobs.

A large challenge we faced was that of parallelization: Genome-scale assemblies are difficult to parallelize because they fundamentally require a large amount of self-referential data-dependencies. To mitigate this effect, we found ways to parallelize some of the more time-consuming processes[2] while leaving the majority of the algorithms unchanged. For more information about this parallelization, see *Concurrent Assembly Using Go*, below.

Prior Work

To prepare for this project, we looked into current research from professionals and universities in the field. Many interesting topics came up, such as the parallel cloud architecture used by Berkeley, but it was interesting to find that no significant usage was coming from the Go programming language,[1] even though the field is well aware of the benefits as well as the easy of utilizing low-cost cloud storage and cloud computing.[2] From Michael Schatz's overview of HPC genomics, "[with] 320 computers at Amazon, we can call SNPs in a whole human genome in about four hours, for about $100." There is clearly a dramatic advantage to cloud-computing in this space.

Other systems are already utilizing HPC in the context of high-throughput computational genomics work: The Cai Lab at the University of Edinburgh has even embarked on a large-scale genomics system that runs exclusively in Meteor.[6] Although we may not be able to offer the same level of precision as Dr. Schatz's genome assembler, we hope to extend the functionality

[2] We benchmarked these processes using basic profiling techniques: We did not attempt to arrive at a high-fidelity profile of our code as we were only interested in determining, on average, which parts of our algorithms were more or less time-consuming.

by improving the modularity of the system — something that is often lacking in other cloud-compute genome-scale assembly options.

Methods and Software

a. Cloud Architecture

We chose to use Amazon's m3.medium EC2 instance family. Though it is not the lowest-cost solution, it provides the highest reliability, and it allows us to model our software on an EC2 instance that could easily be expanded to higher amounts of resources in the future. The particular instance that we have chosen also benefits from Elastic Beanstalk capability, which means that storage space can be increased with low latency and little down-time. More importantly, the server benefits from the ability to make short 'bursts' of high-efficiency computation, and almost entirely hibernate when not in use. This offers a significant cost decrease to the host admin. The EC2 instance itself is configured to run standard Linux Ubuntu 14.04 LTS, which means that any user can easily boot our system on their Ubuntu machine with little extra configuration necessary.

b. Asynchronous Front-End and DDP Streaming

Our user-facing system is written in Meteor, a framework that is highly parallelizable. Though the web framework in use wasn't particularly relevant to the computational genomics work that we embarked upon, we needed a system that would allow us to stream data from the client's computer to the backend. Meteor utilizes a new method of data-transfer, called DDP (*distributed data protocol*). Using DDP, we can stream files of any size to the server, thus avoiding the common pitfall of the file-size limits incurred by conventional HTTP RESTful APIs. Unfortunately, many users' browsers still enforce the common data-cap of 2GB, though this can be disabled by an experienced user (or they can access our services via the command-line, where there is no such limit).

Building on top of this framework, we allowed for a simple web-facing user interface. For instance, a user can sign into his account, see all of his private files and the publically available files, and upload a new FASTA resource to be queued for assembly. A user has no

need to be fluent in programming or server-side technologies, and need not interact with the backend of the system itself, but rather use our non-coder-friendly website to easily upload and analyze files.

Using this system we were able to succeed in or mission to allow for cheap, user-friendly cloud storage. (Tangentially, GenomiK can even serve as a simple storage solution for arbitrarily large FASTA files, as we do not cap a user's upload size and will continue to host a FASTA file even if it cannot be assembled.)

c. Concurrent Assembly Using Go

The second part to consider within our implementation is the parallelization of tasks using Go. The currently available system by Schatz allows for several hundred computers to be hooked up in order to run available genomic algorithms, but we chose to instead run concurrent processes using a single machine.

Our approach is centered around many algorithms that are already established. When a user uploads a set of reads, the system is able to transfer the reads to the backend using Meteor, and then the GenomiK package on the server can can run a precompiled executable of a Go program. We've written each function (upload, conversion, unitigs, alignment, etc) in its own standalone module in order to keep the system as modular as possible. This allows us to easily add new algorithms to our software package, or substitute other systems for our current choice.

The first implementation that we fully ported to Go is our Smith-Waterman alignment algorithm, a derivative of the Needleman-Wunsch. (This particular feature is not supported in full through the web-interface, though it can be called from the rest of the GenomiK module.)

When working on genome assembly, our primary goal, we sought to to take existing gene assembly algorithms and modify them to reduce their data dependencies as much as possible in every step. This was very important since the only way that we could benefit from concurrency is by allowing separate computations to happen simultaneously with minimal interaction. The basis behind this is to use an approach similar to that of the initial Celera team[4]. We initially start with just the reads of a genome, uploaded through either compressed or uncompressed FASTA format (in order to make it simpler for the backend to handle data). The program will

calculate (specifically in *overlap.go*) the initial overlap of all of the reads concurrently and can then output all of the significant overlaps that are greater than k=40bp. From there, the algorithm continues to create a system of best-buddies as discussed in class, allowing for both the best buddies to the left and to the right - the reads that show the most amount of overlap to the read in question - of any given read (we throw out ties as that defeats the purpose of "best" buddies). As output of this initial program, the Go executable prints JSON to stdout so that it can be easily incorporated into a larger program as a service. We decided to use JSON output since this allows for a tradeoff between human and computer readable output, without sacrificing file-size.

It's important to note that this is where the greatest amount of speed increase occurs. Since finding overlaps between any two reads is completely independent of any other two, the task is entirely parallelizable. We were able to achieve on average an 80% decrease in the time it took to compute all of the overlaps when compared to a similar sequential implementation of the same naive algorithm. Additionally, we saw a dramatic decrease in the time it took to compare these overlaps that were greater than 40 in order to determine the largest overlap, as this could also be run concurrently.

From there, the unitigs are computed (specifically in *unitig.go*). After finding the best buddies to the left and to the right, we are able to concurrently assemble the best buddies in order to obtain Unitigs, or high-confidence contigs. This is done by selecting a random point within the list of best buddies, and traversing from the best buddies left/right until it reaches end points. These end points are where a read no long has a best buddy to the left or right, respectively. The actual search left and right can allow for a concurrent behavior, but in practice offers little to no improvement in speed. Still this algorithm was ported into Go in order to remain consistent, and allows for one of the most efficient methods of assembling Unitigs, mapping more accurately to the human genome.

Lastly, we allow for the actual assembly of the Unitigs into the genome. To keep our algorithm simple, and to allow for enough time to increase the speed in previous steps of the genome assembly, this part was done in Python. The actual assembly takes place by referencing one of the genomes already available in the system. We chose to do reference assembly since genome references are becoming more numerous and more accurate. The algorithm focuses on

using the Z-algorithm in order to find approximate/exact matches within the reference, combined with the use of Boyer-Moore in order to be as efficient as possible [5]. This is a basic approach that allows for mismatches to occur, choosing the highest occurring read from the overlapping unitigs if they don't match. Although this is not as fast as the Go implementations (largely due to Python being slower as a whole), it's still set up in a way that can eventually be ported into Go to be parallelized. It was not done so because of time constraints, but offers the ability for multiple Unitigs to be overlapped onto the genome (within all of the positions that they match given a margin of error) all at the same time using Go concurrency.

In order to implement several concurrent algorithms in go, we created a generic framework, called a *Runner* (dynamically generated GoDoc for the Runner package is publically available). Our intent was to create a framework that would allow any function to be made concurrent with as little overhead on both the programmer and the runtime as possible. With the *Runner* package, one can easily make any function run concurrently on any desired number of goroutines (similar to threads). After being created, a *Runner* provides a concurrent safe view into the results and errors of multiple invocations of its configured function. Similar to waiting on a child process on a unix system, the *Runner* interface provides a *Wait()* method that will block until results are available. The only downside to using a generic package such as our *Runner* implementation is that the results will require casting to their original type before use which shifts go's type safety from compile-time to runtime. However, this introduces minimal overhead during runtime and is still preferable compared to a dynamic language such as python which ignores type safety all together. We were able to make use of the *Runner* framework multiple times in *genomik-cli* (for example, *overlap.go* uses this framework to compute overlaps concurrently). Besides its use in our own project, our hope is that the ease of using the *Runner* framework will help convince others to use go for algorithms that can easily benefit from concurrency.

d. Shortcomings and Future Implementations

Given more time, we would have attempted to further reduce data dependencies in order to benefit from go's easy concurrency during other steps of the unitig assembly. We also would

have liked to change our algorithms so that the backend's memory requirements could be lessened.

There were also other features we wanted to implement, and even some we tried to implement, but were not successful. One such feature was the ability to assemble a genome without a reference genome. Using only the Unitigs, in which this system was easily parallelized, the user would have to know the approximate length of the final genome solution in order for this to be successful, as is the case with many scaffolding algorithms[4]. But we posited that most users won't know the expected length of a new genome, and if it's an old genome a researcher can easily upload or use (if available) a reference genome. Thus we only included genome assembly using reference genomes. We also considered that almost *any* reference genome would do, given an adequate level of biological similarity. (For instance, if you attempted to assemble a human genome using a macaque reference genome, it would be highly unlikely that the unitigs 'split' within a region that is unrepresented in the macaque genome.) With some level of uncertainty, most genomes that share a substantial number of genes could be substituted for a same-species reference genome for our purposes.

An extension to this could have been to add in the ability to assemble the reads onto the reference genome while taking into account edit distance (i.e. insertions and deletions). This we tried at length to implement, but we were unsuccessful. The problem that we encountered was that the Unitigs themselves, if consisting of insertions or deletions, could not use the existing z-algorithm to find matches. This posed a problem in actually finding to start looking for a match for the unitigs. The end approach was to separate out the unitig into many smaller pieces, and to find matches for most of the smaller pieces in order to determine where the gaps should be, but this did not work. What ends up happening is that the smaller pieces have too many matches than what they should (due to repeats in the genome). It essentially defeats the purpose of us using the best buddies. Thus the reads were incredibly unreliable and we stuck to doing only general mismatches without taking insertions and deletions into account.

Additionally, the user is able to assemble a genome, but we the last part of the assembly is still in Python. This was a shortcoming due to time constraints. We set it up in a way that can be converted to concurrent goroutines, but were unable to actually convert this. Most of the code

was already written in Python so the implementation stayed in Python. And we were not able to add an extension onto this where a user can query reads onto a genome. This could possibly be pulled out of the current assembly algorithm, but the addition itself would require too much of a change on the front end of the application to be added into this iteration (though we look forward to continuing to iterate in the future.)

Lastly, we want to add even more modularity to the website. Right now, a user is able to upload files and run our algorithms, but the website is set up in a way that a user can do almost anything. We want to add in the future the ability for a user to not only select our algorithms, but select other algorithms out there that are tried and true. Additionally, a user should be able to upload his own algorithms for testing purposes. This would help better open source the development process of web-based assembly architectures, hopefully accelerating performance and growth.

Results

The most significant result comes from the speed increase. By allowing the use of the Go programming language on the backend, overlaps can be achieved up to ~80% faster. Considering this is one of the most processor intensive steps in assembly, this allows for a significant decrease in payload on the server, allowing for the cost of assembly to be decreased even more. Also, a major speed increase occurred in querying reads. On any sort of data, test or actual, the speed increase is obvious, also keeping around 80% faster. This is simply because the features of Go allow existing algorithms to better use the multi-core processors in today's computers.

Additionally the use of the modular Go helpers to handle the concurrency is a great addition to the field of computational genomics. And they had great success within the different Go programs on the backend of the server. Using Node.js to call a simple Go executable allowed for the Go files to also have access to other files within the directory. Thus both the overlap.go and the unitig.go files could make use of the same helper methods. This can then be extended by others later on to help improve algorithms through concurrency.

Compared to other methods, such as the full genome assembly of Celera, our solution offers significant speed increases at the beginning of the algorithm. This is a result of running on

both test and actual data sets. The actual speed at later steps is harder to determine, but it's likely that since Celera uses full scaffolding it will end up being slightly faster for the entire assembly. Luckily, our solution offers a modular approach that contains results from each steps in assembly. During later implementation we should be able to combine the concurrency of overlaps in Go with the full Celera suite (as it can be run from command line on the backend of the server). Or even better, we, or users, can reimplement the Celera suite in order to allow for faster assembly with concurrent scaffolding using Go.

Lastly, our front end interface was able to achieve the level of simplicity that we wanted a user to have. The user can easily interact with the backend without having to actually know too much about genomics. It's interactions peak the interest of new users and offer enough features for seasoned professionals to be able to use it as a professional tool for genome assembly. And again, it offers the ability of mass storage of information for extremely cheap. The development of cloud architectures in the future should only make the fact more true. So given the current situation, our model offers the same level as computation available from other options, but with a higher level of modularity.

Conclusions

All in all, our algorithm adds a significant speed increase over others. Although not every algorithm within genomics has been converted into Go, our solution offers the initial steps needed to make that easier, allowing for actions to become parallelized with minimal effort. With the combination of mass data storage and convenient helpers, our algorithms are able to easily be expanded. This offers a new level of research within genomics centered on community involvement as a method of improvement.

Resources

1. http://www.cs.berkeley.edu/~kubitron/courses/cs252-S12/projects/reports/project1_report.pdf
2. https://www.genomeweb.com/informatics/michael-schatz-genome-assembly-and-cloud
3. http://vlab.amrita.edu/?sub=3&brch=274&sim=1433&cnt=1

4. http://www.sciencemag.org/content/291/5507/1304.short
5. Haubold, Bernhard, and Thomas Wiehe. *Introduction to Computational Biology: An Evolutionary Approach*. Basel: Birkhäuser Verlag, 2006. Print.
6. http://genomecarver.cailab.org/