Math 18.337 Project: Mapping metabolic networks in the human microbiome

Chris Smillie

12/14/2012

Background: The human body contains ten trillion human cells, comprising ~200 cell types (skin, muscle, neuronal, etc.) and ~25,000 unique human genes. However, these human cells represent only a small fraction of the total cells in the human body. For each human cell, the human body contains >10 bacterial cells, and for each human gene, the human body contains >100 distinct bacterial genes. This hidden majority of bacterial cells, collectively known as the "human microbiome", weighs only two pounds yet encodes many functions that are critical to human health, ranging from metabolism to immune regulation, and has been implicated in diseases such as obesity, diabetes, and cancer. Historically, the human microbiome has been difficult to study because the vast majority (>99%) of bacteria cannot be grown in laboratory conditions. However, recent advances in high-throughput DNA sequencing have allowed microbiologists to interrogate these complex bacterial communities at unprecedented scales, generating terabytes of DNA sequence data in a single experiment.

The goal of this project is to use parallel computation to analyze these DNA sequence data, with the goal of understanding the metabolic functions of ~1000 bacterial communities sampled from 18 body sites in the human microbiome. By integrating knowledge of community metabolism with other types of data (e.g. health status, host genetics, etc.) it may be possible to attain an ecosystems-level perspective of community functioning, which may aid the rational engineering of microbial communities on the human body.

Objectives: Although the problem of metabolic mapping is biologically interesting, my primary objectives for this project were to (1) learn to use Amazon Web Services (AWS) including EC2, EMR, and S3, and (2) run parallel code in Julia. Therefore, I decided to first use AWS to reduce my "big data" (~2.3 TB compressed) to a more manageable size (~1 GB) and to use Julia and/or Python for the remaining analysis.

Algorithm/outline: To map metabolic networks in the human microbiome I used the simple algorithm outlined below.

```
MR steps: | Algorithm steps:                                      | Tools used:
          | Upload data ---------------------------------------- | AWS/Beagle
MAP       | For each sample in data                               |
...       |     Unzip sample (.bz2)                               |
...       |     Convert sample to different format -------------- | Python
...       |     For each sequence in sample                       |
...       |         Map sequence to known database of genes ----- | BLASTX
...       |         If similarity > threshold                     |
...       |             Emit [gene ID, {sample ID: count}]        |
REDUCE    | Convert list to data matrix ------------------------- | Python/Julia
          | Analyze data matrix --------------------------------- | Python/Julia
```

Data: I used data from (1) the Human Microbiome Project (HMP) and from (2) a survey of people living in the USA, rural Malawi, and the Amazon rainforest in Venezuela. I analyzed the first dataset using Amazon Elastic MapReduce (EMR) and the second dataset using Beagle.

Methods: Here, I discuss the steps I used to analyze metabolism in the human microbiome, emphasizing computational challenges I encountered along the way.

Algorithm development: MapReduce is not strictly necessary for this project. An alternative is to perform the analysis on Amazon EC2 using EBS volumes. However, an EBS volume can be mounted to only one EC2 instance at a time. Therefore, I would need to create an Amazon Machine Image (AMI) and copy it to each instance separately, requiring costly data duplication.

Data transfer: The data from the Human Microbiome Project resides on an FTP server, and consists of 764 files of DNA sequence data, ranging from ~100 MB to ~10 GB in size (~2.3 TB total). To avoid a slow 2.3 TB transfer, I wanted to transfer multiple files in parallel from the FTP server to the S3 bucket. However, this was challenging because I could not use PUT directly from the FTP server, nor use GET directly from the S3 bucket. To solve this problem, I had to first download the data to local storage, and then upload from local storage to S3, thus requiring twice as much data transfer. An alternative, but costly, solution may be to use EC2 instances to download the data to separate EBS volumes and then transfer the data from the EBS volumes to the S3 bucket. To avoid overloading the FTP, I limited the number of simultaneous downloads. I also used parallel Python to schedule downloads and uploads to avoid using too much disk space, as downloads were faster than uploads. I encountered two problems: (1) downloads would stall for no apparent reason, and (2) some of the downloads were incomplete (in contrast, S3 uploads only succeed if 100% of the file is transferred). Overall, transferring data to S3 was a headache.

<u>Elastic MapReduce</u>: I used Amazon EMR to map 2.3 TB of DNA sequence data to the Kyoto Encyclopedia of Genes and Genomes (KEGG), which is a databases of known enzymes and pathways. The MapReduce job was setup as follows. The *input* to the MapReduce job is the raw DNA sequence data (.bz2). The *Map* step first unzips these data, converts it to a different format (.fastq to .fasta), and maps the DNA sequences to the KEGG database, requiring a minimum similarity score of E-value < 1e-8. The *Map* step then counts local occurrences and returns a tuple, which contains the enzyme ID and an associative array of counts for each sample, e.g. (KEGG_Enzyme_X, {Sample_Y: 5}). This is an example of local aggregation and is similar to the word count problem in Homework 2. The *Reduce* step is initialized with a list of all sample IDs. It then aggregates counts across multiple samples and returns a tuple, which contains the enzyme ID and a vector of counts for all samples, e.g. (KEGG_Enzyme_X, [$C_{x,1}$, …, $C_{x,764}$]) where $C_{i,j}$ is the count of enzyme *i* in sample *j*. The *Reduce* step emits an ordered vector rather than an associative array because an associative array uses much more memory. However, I initially had difficulties in implementing this approach because each Reducer only has access to the samples it contains, and there is no guarantee that each Reducer contains all samples. I solved this problem by initializing each Reducer with a list of all samples to synchronize their lists. Overall, the MapReduce framework was relatively easy to implement. The final output of this process was a 1 GB matrix of enzyme abundances (rows) across all samples (columns).

<u>Data analysis</u>: I used (1) Python and (2) Julia to analyze the data. First, I used Julia to normalize counts in each sample and to calculate summary statistics, including Shannon's entropy, which is a measure of the metabolic diversity within a sample. I also tried to calculate the singular value decomposition of the data to get eigen-samples and eigen-genes, but this was too memory-intensive. Presumably, the eigen-samples with the largest eigenvalues would reflect the different body sites from which the microbial communities were isolated. Similarly, the eigen-genes may reflect genes that are differentially distributed among these body sites (e.g. carbohydrate metabolism in the gut). To test the hypothesis that differences in body sites explain much of the

variation in the data, I used Julia to calculate the cosine similarity between all samples. I used Python to analyze the antibiotics data, and matplotlib (a Python package) to construct all plots.

Using Julia: For serial computation, I thought Julia was intuitive and easy to use. It is also very fast, as shown by comparison to Python (see below). However, I had some trouble using Julia for parallel computation and I spent a long time trying to debug code. I realized that there must be a problem with the version of Julia that is installed on Beagle, because even examples directly from the documentation do not work, e.g.:

```
julia>      function compute_something(A::DArray)
                  B = darray(eltype(A), size(A), 3)
                  @sync begin
                        for i = 1:size(A,3)
                              @spawnat owner(B,i) B[:,:,i] = A[:,:,i]
                        end
                  end
                  B
            end
julia>      data = darray(Float32, (10,10,10), 3)
julia>      compute_something(data)
            in compute_something: B not defined
```

After installing Julia on my computer, these problems were resolved, but I encountered some other difficulties. For example, with some of my code, Julia would quit running without giving an error message, making it difficult to debug. In other cases, I noticed Julia would only give error messages in interactive mode. For example, I could get the following code to work:

```
function normalize(data::DArray)
      (nrows,ncols) = size(data)
      sum_data=darray(eltype(data),(1,ncols),2)
      @sync begin
            for i = 1:ncols
                  @spawnat owner(sum_data,i) sum_data[1,i] =
                  sum(data[:,i])
            end
      end
      @sync begin
            for i = 1:ncols
                  if sum_data[1,i] > 0
                        @spawnat owner(data,i) data[:,i] =
                        1.0*data[:,i]/sum_data[1,i]
                  end
            end
      end
      data
end
```

```
function calc_entropy(data::DArray)
      (nrows,ncols) = size(data)
      shannon_index=darray(eltype(data),(1,ncols),2)
      data += .00000000001
      @sync begin
            for i = 1:ncols
                  @spawnat owner(shannon_index,i) shannon_index[1,i] =
                  -1.0*sum([data[j,i]*log(data[j,i]) for j=1:nrows])
            end
      end
      shannon_index
end
```

However, I couldn't get very similar code for calculating cosine similarity to work, as Julia would exit without an error message. I did eventually get slightly different parallel code to work, but it was much slower than my serial code (see Results). A minor difficulty I noticed is that some functions (e.g. dot()) are only defined for Arrays and not DArrays, but using convert() is an easy solution. Of course, many of these problems are part of the learning curve for a new language. Overall, I thought Julia was an incredibly simple and elegant language for computing.

Results: In total, I mapped 2.6 x 10^8 enzymes in the HMP data, and 15,195 antibiotics resistance genes in the Malawi/Venezuela/USA dataset. In Figure 1, I plot the average functional diversity of each body site, measured using the Shannon entropy of their (normalized) metabolic profiles. It is clear that different body sites have different metabolic potentials, as demonstrated by, e.g. the gut (Shannon Index = 9.5) and the subgingival plaque (Shannon Index = 10.5). An intuitive confirmation that the metabolic mapping procedure is relatively accurate is that sites with higher species diversity (e.g. the plaque and saliva) tend to have greater functional diversity than sites with lower species diversity (e.g. the ear and cheek). To compare serial and parallel Julia, I plot the run time of this code (see 'calc_entropy' above) against the number of processors it is parallelized on (Figure 2). Surprisingly, it is slower on 2 CPUs than 1 CPU, but faster for subsequent additions of CPUs. Overall, parallelizing this code on 4 CPUs gets a 1.5-fold speedup in computation, despite the fact that the task is embarrassingly parallel. The sub-linear speedup in processing time may be the result of inefficient coding, or is due to the time spent on @spawnat statements for each column of the matrix. Next, I look at the cosine similarities I calculated in serial Julia. Whereas the serial Julia code was very fast (696.37 seconds), the Python code, which uses numpy arrays and the numpy.dot() function, was incredibly slow (> 24 hours on Beagle). This 124-fold speedup using Julia is remarkable and probably doesn't reflect differences in code efficiency because the codes are nearly identical. Using smaller datasets to avoid Beagle's 24-hour time limit, the Julia code takes 0.03 and 0.073 seconds to calculate the cosine similarities, while the Python code takes 0.34 and 6.62 seconds, respectively (Figure 4). These times represent 11-fold and 91-fold speedups, suggesting that the Python code does not scale well to larger amounts of data. Therefore, it appears that some of the Julia speedup involves its ability to handle large amounts of data efficiently. In Figure 3, I plot the cosine similarities of samples that are from the (1) same body site, and (2) different body sites. From this plot, it is clear that samples from the same body site have higher metabolic similarities ($P < 1E-10$; Mann-Whitney U test), suggesting that body site is a significant determinant of a sample's metabolic profile. In class, I also showed results from the analysis of a subset of metabolism, antibiotics resistance, in the human microbiome. Because this analysis did not use parallel computation (except in generating the necessary data), I will not discuss it in very much detail. Briefly, I find that antibiotics resistance is widespread in the human microbiome (data not shown) and that it is more widespread in Malawi/Amazonas than in the USA, but only after 2 years of age (Figure 5).

Conclusions: In conclusion, I learned to use Amazon AWS and Julia in this project for parallel computation on DNA sequence data. In general, both tools were straightforward to use, although I encountered some difficulties both in uploading data to Amazon S3 and in learning to use Julia. The data analysis I performed uncovered some trends that mostly validate the data generation step, but in the future, I will use these enzyme annotations for more biological analyses. I didn't have time to do anything really interesting with these data, except to calculate summary statistics and to look at major patterns of variation among the samples.

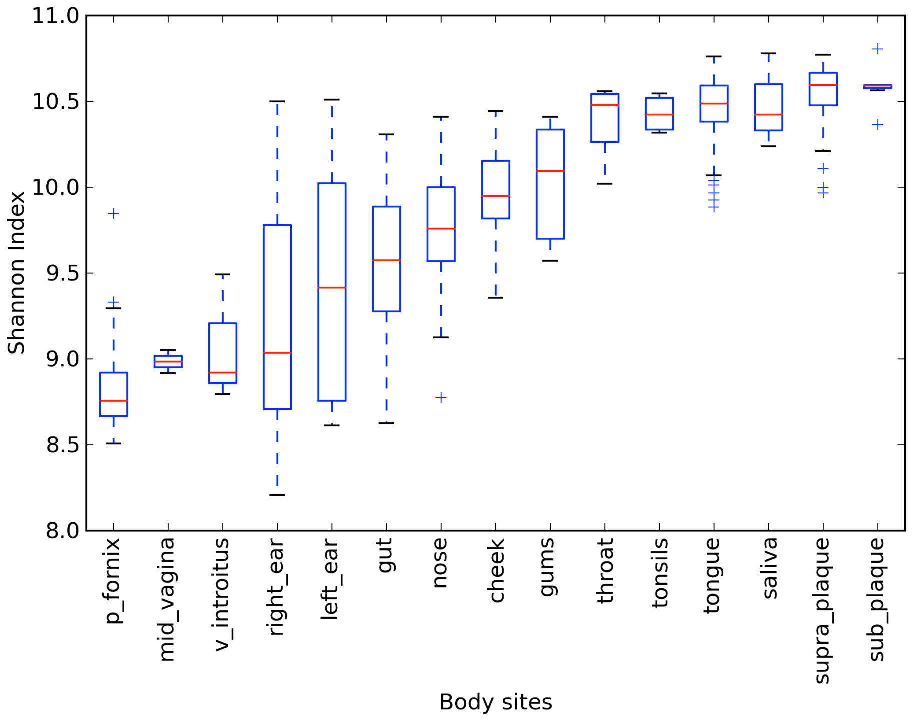

Figure 1: This shows the metabolic diversity of each body site, measured using the Shannon entropy of their normalized enzyme abundances.

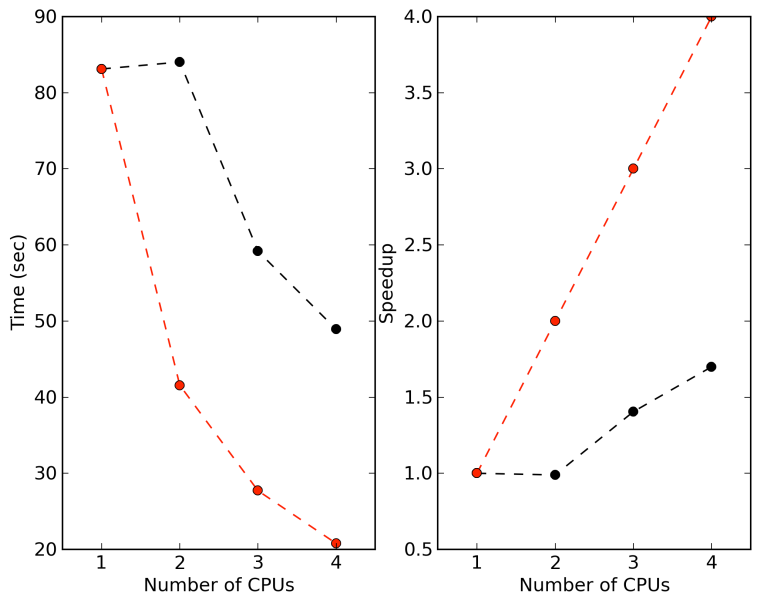

Figure 2: This shows the actual (black) and linear (red) computation times and speedups of the parallel Julia code for calculating Shannon's entropy.

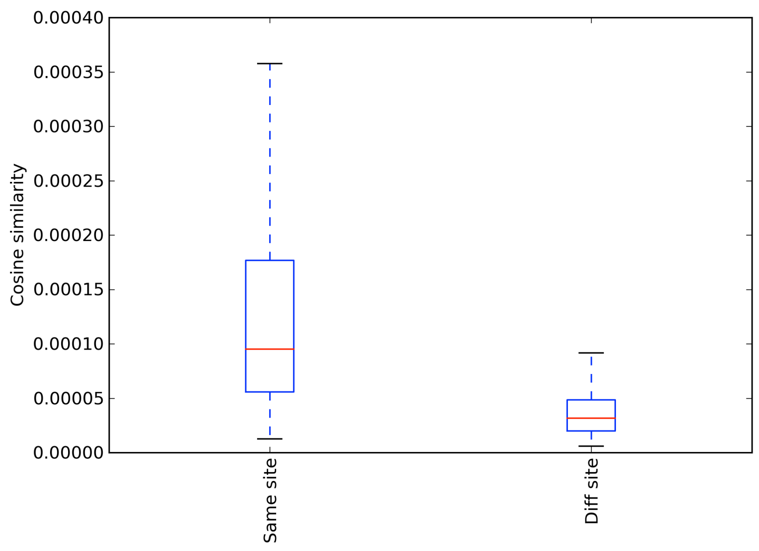

Figure 3: This shows the cosine similarities of samples from the same body site, and samples from different body sites.

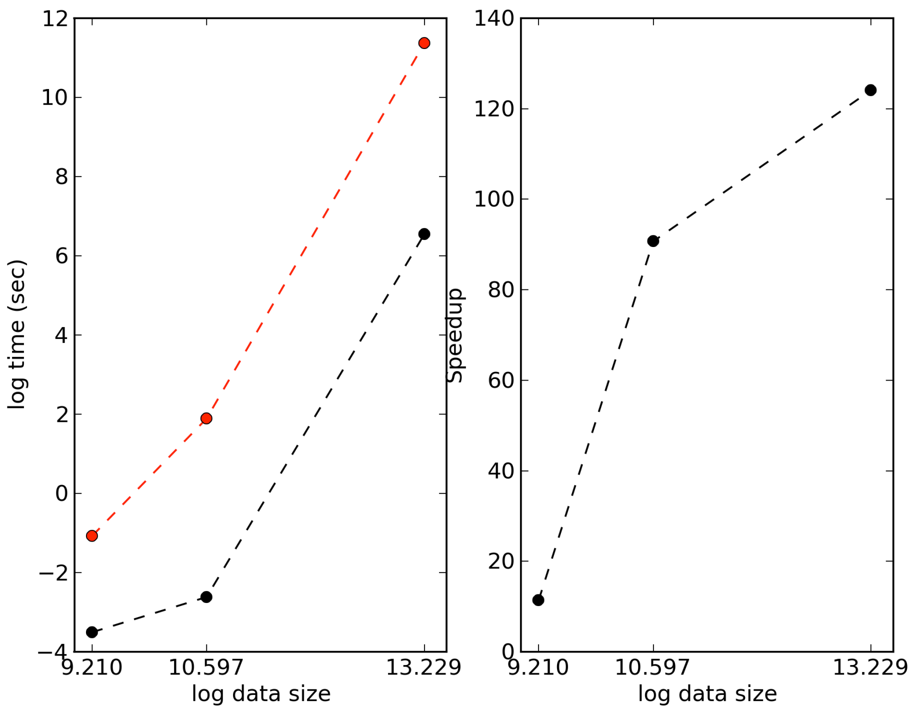

Figure 4: This shows the time it took to calculate the cosine similarities using both Julia (black) and Python numpy (red). On the right, it shows the percent speedup of Julia over Python.

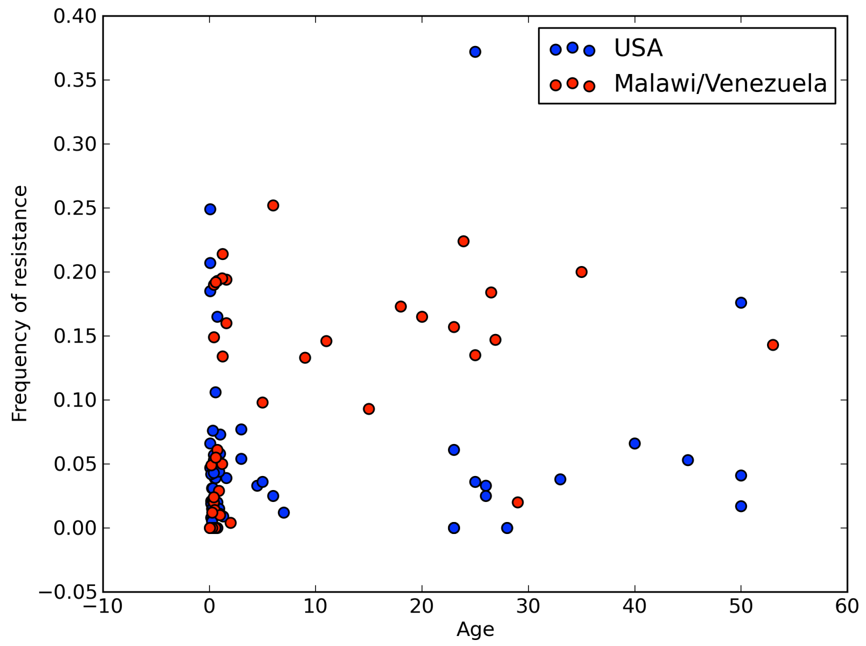

Figure 6: This shows the frequency of antibiotics resistance (normalized to the amount of non-human DNA sequenced) for subjects from the USA (blue) and Malawi/Venezuela (red).