Readme

Whole Genome Shotgun Project

Max Masnick, Joel Burrill
David Bardin, Aaron Lerner

The purpose of this project was to make a simplified program that reconstructed a long sequence of DNA that had been randomly broken into overlapping pieces, as is necessary to do for whole genome shotgun sequencing.

Code Changes

We started out with a naïve (i.e. slow) program that used a regular expression pattern to perform this reconstruction. In detail, this algorithm checked to see if two strands overlapped by creating a String containing both strands with a marker in between. Because this String had to be created for every single overlap test (and a lot of overlap tests are necessary, especially in the hugefasta file), this algorithm had a very high runtime. Our first goal was to speed up the code using both regionMatches and a more intelligent algorithm.

The first change we made to our code was to rewrite it so that the threshold could be changed from the MainShotgun class. To do this, we had to add an int threshold parameter to both the merge and the doGun methods of the ShotgunReconstructor. Then, in the MainShotgun code, we created the instance variable int threshold, set it to whatever we wanted the threshold to be, and passed it into doGun.

We then modified the MainShotgun code and the ShotgunReconstructor code so that we could run various thresholds without having to reselect the file for each run. To do this, we took the code from MainShotgun that was actually reconstructing the strands and put it in a method called doStuff. We then added another ArrayList of IStrands called myBaseStrands to the ShotgunReconstructor so that we could read the strand data into myBaseStrands, copy it over MyStrands, reconstruct the DNA using myStrands, clear myStrands, then recopy the original data into myStrands using the unmodified data from myBaseStrands. This then allowed us to loop over the doStuff method in MainShotgun and change only the threshold each time we looped over the doStuff code.

Next we created the FastStrand and the FastStrandFactory classes. The FastStrandFactory class was exactly the same code as the SlowStrandFactory except it created FastStrands. In the FastStrand code, we removed the regex from the merge method, which was slowing down the reconstruction significantly. (Otherwise, we used the same code from SlowStrand). Our new merge method used regionMatches to check if two strands overlapped. This code checks for an overlap by comparing the end of one strand to the beginning of the other, starting at the threshold length and increasing until a match is found. If a match is found, the code will loop once more to see if there is a longer match. If there is not, a new IStrand is created based on the previous match. If the strands never overlapped, null is returned.

Next, we created the NoReduceFastStrand, the NonReducingReconstructor, and the NoReduceFastStrandFactory. The NoReduceFastStrand differed from the other IStrand implementations in that its merge method checked for both containment and for overlap (the other implementations checked to see if two strands could be merged or if one was contained in the other separately, which is slower). We did this by moving the contains logic from the ShotgunReconstructor and writing new code to check for containment in the merge method of NoReduceFastStrand. This code determined which of the two strands being compared was longest and used this information to test for containment with the contains method. The contains method uses regionMatches to compare the smaller strand of n length to every group of n characters in the larger strand. If a match was found, the merge method returned a new ISrand cotaining only the longest strand. If the smaller strand was not contained in the larger strand, the code ran merge in the exact same way the FastStrand code did. This allowed us to change the doGun method of the NonReducingReconstructor so that instead of iterating over myStrands twice, once for contains and once for merge, it could call the merge method of NoReduceFastStrand and iterate over myStrands once.

During this process, we found that if we checked for containment by calling indexOf, the code worked faster, but the instructions said not to use indexOf, so we did not use it in our data collection.

Data Collection

We used the modifications to MainShotgun discussed above to determine thresholds for initial testing. However, the modified code inexplicably stopped working when we began the actual data collection, so we were forced to automate our code in a new way. We were able to place the contents of main method in MainShotgun inside a separate method (doStuff), and then use the main method to loop over doStuff. Using these modifications, we were able to automatically test thresholds 1 to 100 to find the optimal threshold. We modified the printed output of the program to create a "log file" (we did not actually write this data to a separate file; we copied and pasted it out of the console). We then wrote a program in PHP to parse these log files into an HTML table to make it easier to work with the data.

After we determined the optimal thresholds, we ran each algorithm (NoReduce, FastStrand, and SlowStrand) five times on each file (huge, large, and simple) on two computers. By adding a loop to our modifications of MainShotgun, we were able to completely automate this process.

Data

Computer Specs

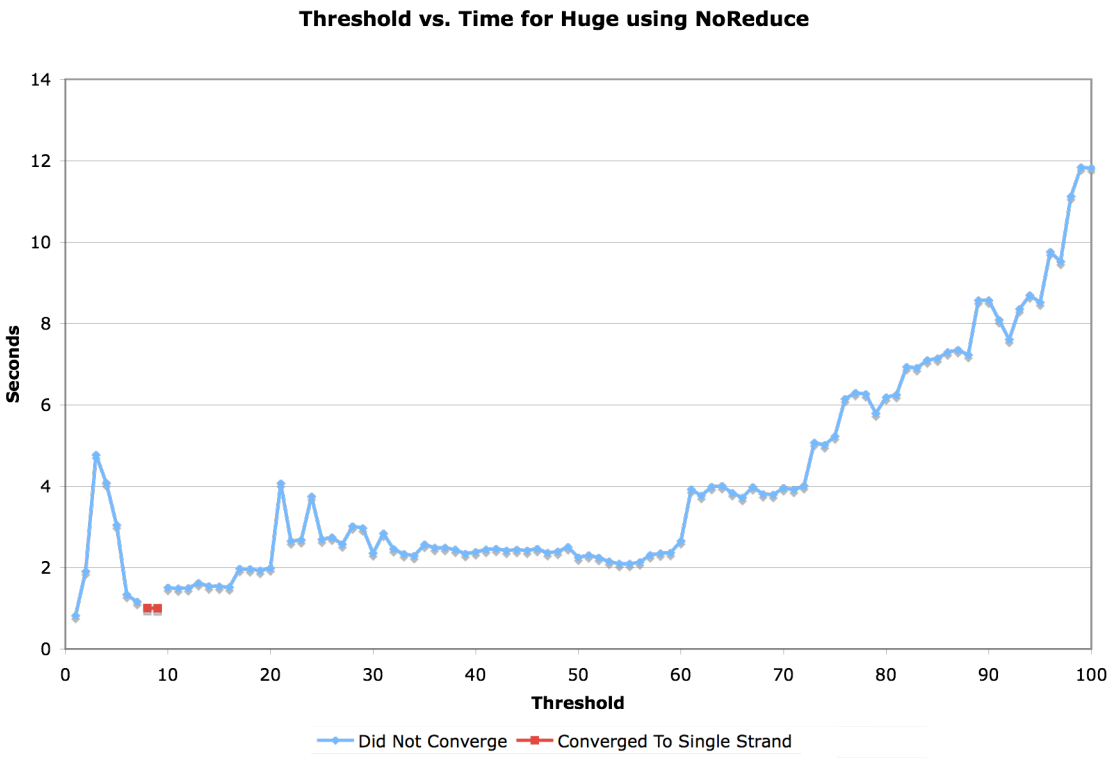
- Max's Computer:
  - 1.5GHz PowerBook G4
  - 1GB Memory
  - Mac OS X
- Aaron's Computer:
  - 2.13GHz Pentium M Processor
  - 512 MB Memory
  - Windows XP

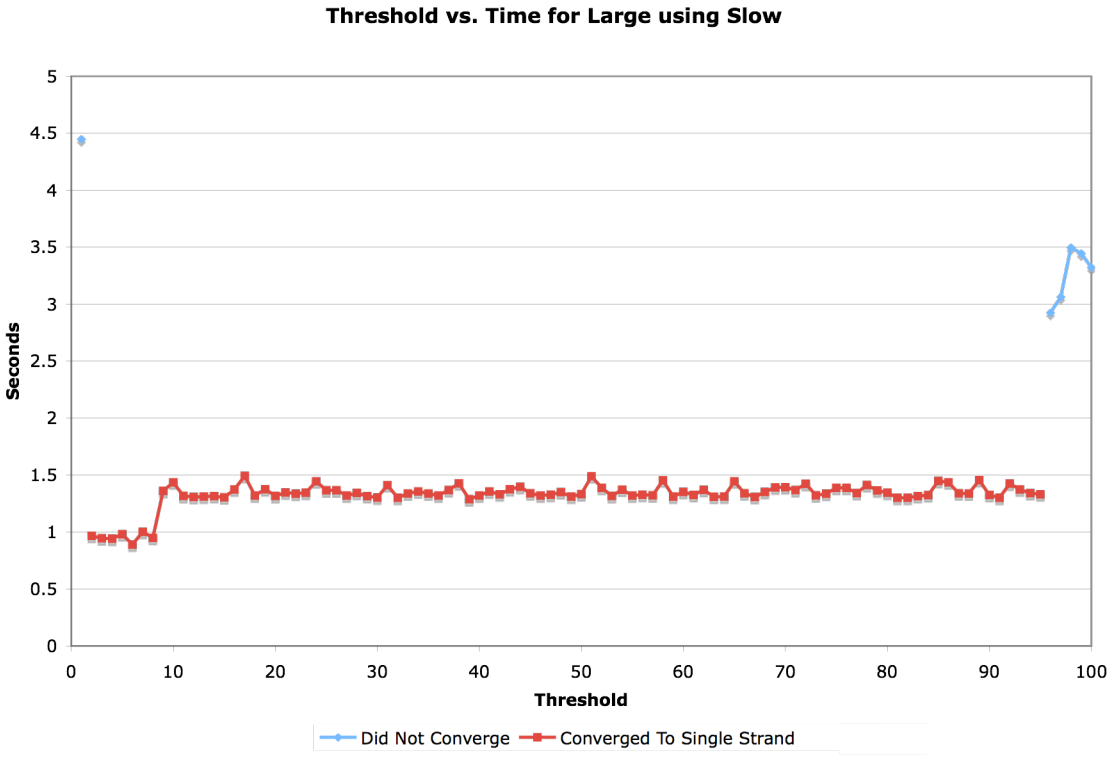
Threshold Data

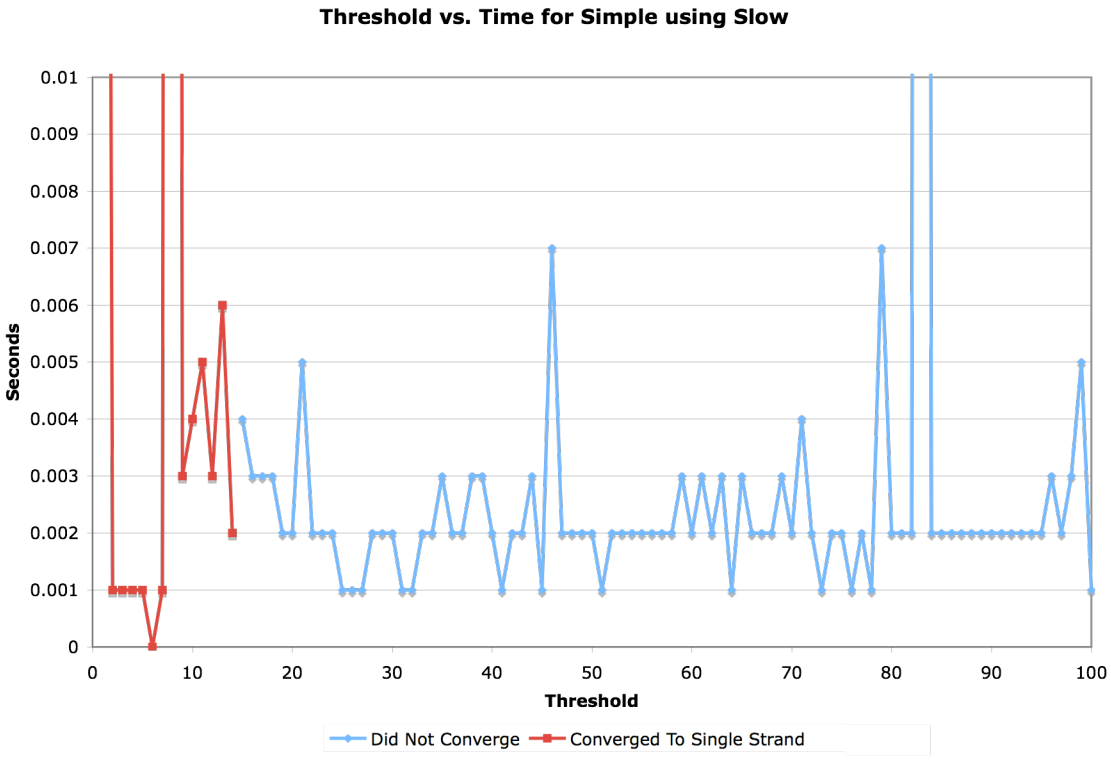
(Only the slow algorithm yielded useful data for the large and simple files. We ran a threshold test on the huge file using only NoReduce because the other algorithms took too long. All threshold data is from Max's computer)

Summary of Threshold Data

	Min	Max	Optimum
Huge	8	9	9
Large	2	95	6
Simple	1	14	6




Threshold vs. Time for Large using Slow
Seconds
Threshold
Did Not Converge
Converged To Single Strand


Threshold vs. Time for Simple using Slow
Seconds
Threshold
Did Not Converge
Converged To Single Strand

Runtime Data

Huge

Max's Laptop

Trial	NoReduce Time	FastStrand Time	SlowStrand Time
1	1.105	8.602	1168.811
2	1.142	8.489	1157.063
3	1.116	8.278	1161.505
4	1.153	8.419	1159.157
5	1.055	8.419	1173.037
Avg	1.1142	8.4414	1163.9146

Aaron's Laptop

Trial	NoReduce Time	FastStrand Time	SlowStrand Time
1	0.484	6.25	484.625
2	0.36	6.203	479.953
3	0.359	6.203	479.718
4	0.391	6.203	479.687
5	0.359	6.203	481.157
Avg	0.3906	6.2124	481.028

Large

Max's Laptop

Trial	NoReduce Time	FastStrand Time	SlowStrand Time
1	0.037	0.017	1.679
2	0.006	0.007	1.035
3	0.011	0.004	1.018
4	0.008	0.183	1.079
5	0.011	0.002	1.022
Avg	0.0146	0.0426	1.1666

Aaron's Laptop

Trial	NoReduce Time	FastStrand Time	SlowStrand Time
1	0	0	0.391
2	0.047	0	0.375
3	0	0	0.359
4	0	0	0.359
5	0.015	0	0.344
Avg	0.0124	0	0.3656

Simple

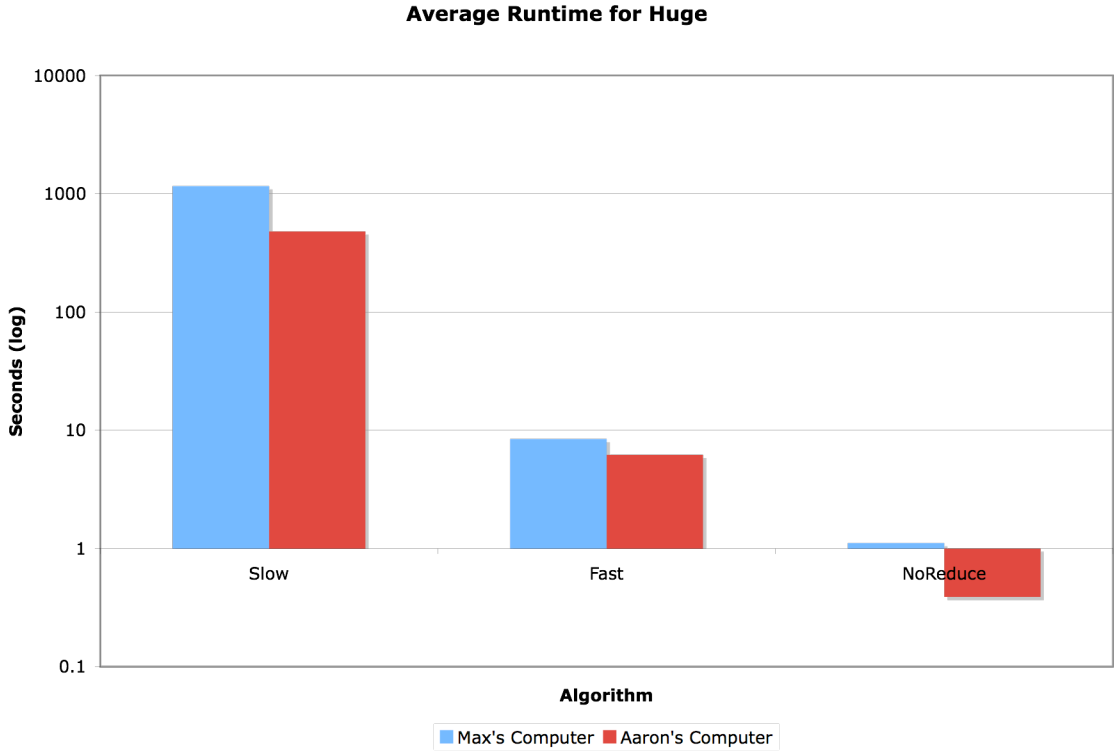
Max's Laptop

Trial	NoReduce Time	FastStrand Time	SlowStrand Time
1	0.014	0.001	0.013
2	0.002	0	0.001

3	0	0	0
4	0.002	0	0.001
5	0.001	0	0.001
Avg	0.0038	0.0002	0.0032

Aaron's Laptop

Trial	NoReduce Time	FastStrand Time	SlowStrand Time
1	0	0	0.016
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
Avg	0	0	0.0032



Analysis & Conclusions

We were able to significantly improve the speed of the program (our data shows a 99% speed increase on Huge between the Slow and NoReduce algorithms). This demonstrates that creating String is a comparatively time-intensive process, visible when processing large amounts of data. Additionally, we noticed that it is always better to iterate once instead of twice, as the time difference between Fast and NoReduce showed.

We also noticed that the method this program uses for determining runtime is not accurate for times less than a second, as shown by the an anomalies on in our data for the faster algorithms on the smaller files. We talked in class about how C programs can give actual processor time instead of "wall clock" time, which would yield much more accurate results.

The runtimes on Aaron's laptop were much smaller than the runtimes on Max's laptop. In general, we have noticed that Java runs much slower on Macs than on PCs, so our results support this observation.

One final note: We found that far fewer thresholds worked with our NoReduce algorithm than with the other group's NoReduce algorithm. We compared our code with the other group's, but were unable to track down the source of this discrepancy. However, our code was performing faster than theirs (at least while we were doing the comparison).