Topics Modeling

Nick Booher, Shuai Jiang, Sepehr Saroukhani

May 4, 2014

1 EG Algorithm and Serial Performance

We implemented the Exponentiated Gradient algorithm based on the flowchart provided in the problem description. The choice of the η parameter is non-trivial. Too large or too small values of η lead to serious convergence issues. A constant value of η that is not updated in each iteration also results in slow convergence. Our trial and error with the parameter did not take long though as an initial value of $\eta = 2000$ combined with the following update in each iteration

$$\eta = 0.99\eta \tag{1}$$

was suggested by Dr. Bindel on Piazza.

Per Dr.Bindel's suggestion, we set the convergence tolerance to $\epsilon = 1E-5$ and only allowed at most 500 iterations. This means that the algorithm sometimes stops after 500 iteration while the convergence tolerance has not been met yet. For the Yelp data set, for instance, the algorithm only converges 1557 times out of the 1573 times that it is called. Fortunately, this does not lead to significant errors in our problem because we obtain the same important words for each topic and only the ranking changes. An example of this can be seen in Table 1 which shows the word ranking obtained by the two algorithms for the gas topic in the Yelp data set.

The timing of our implementation of the EG algorithm, with the above-mentioned restrictions, is more or less on par with the given active set code for the Yelp data set. This can be seen in Table 2 which shows the code output with the two algorithms. However, our code is quite slower than the given active set code without the restrictions on the convergence criteria. The slower performance is even more significant for the Enron data set. This was unexpected, but it is probably because the given code is highly optimized and difficult to beat in the amount of time we had to spend on our implementation.

2 Parallel

From a basic tic/toc timing, we see that the `compute_A` function took the longest, with the for-loop taking up the majority of the time. Fortunately, that piece of for-loop in embarrassingly parallel, as the only synchronization needed is to keep track of the errors, iteration count, and our C matrix.

Using the technique described here,[1], we will assign chunks of columns of

[1] http://blog.ajdecon.org/parallel-word-count-with-julia-an-interesting/

Active set		Exponentiated gradient	
gas	(2.851e-01)	gas	(2.815e-01)
station	(2.975e-02)	station	(2.962e-02)
inside	(1.934e-02)	inside	(1.924e-02)
card	(1.714e-02)	card	(1.798e-02)
car	(1.294e-02)	car	(1.288e-02)
store	(1.233e-02)	store	(1.230e-02)
door	(1.131e-02)	door	(1.118e-02)
star	(9.495e-03)	cash	(9.399e-03)
cash	(9.021e-03)	star	(9.367e-03)
location	(8.339e-03)	location	(8.264e-03)
walk	(8.271e-03)	walk	(8.207e-03)
purchase	(7.506e-03)	purchase	(7.981e-03)
pay	(7.243e-03)	pay	(7.138e-03)
free	(7.102e-03)	give	(7.019e-03)
give	(7.070e-03)	free	(6.984e-03)
employees	(6.926e-03)	employees	(6.881e-03)
including	(6.635e-03)	including	(6.555e-03)
business	(6.613e-03)	business	(6.545e-03)
told	(6.527e-03)	told	(6.441e-03)
town	(6.458e-03)	town	(6.393e-03)

Table 1: Comparing the gas topic obtain by the two algorithms.

Active set	EG
– Compute row scaling	– Compute row scaling
elapsed time: 0.027614471 seconds	elapsed time: 0.027327602 seconds
– Timing anchor words with partial fact	– Timing anchor words with partial fact
elapsed time: 2.613210494 seconds	elapsed time: 2.521886732 seconds
– Compute intensities	– Compute intensities
Max error 2.384186e-7 8.4089623e-7	Max error 2.384186e-7 5.588801e-5
Total iterations: 25543	Total iterations: 83293
elapsed time: 5.488936963 seconds	elapsed time: 5.310499619 seconds
– Find top words per topic	– Find top words per topic
elapsed time: 0.106000463 seconds	elapsed time: 0.11604551 seconds

Table 2: The serial code output based on the two algorithms for the Yelp data set.

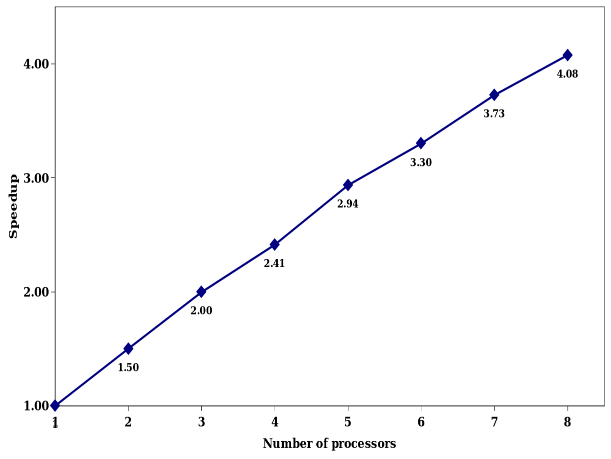

Figure 1: The speedup plot for the Enron data set.

our AtB matrix to each processor to run the EG algorithm on.

More specifically, we first define a function which each individual processor should compute and output; this is called `local_calc` in our code. The bookkeeping for the references is simply kept in a list called `rrefs` after being spawned in a for loop over the number of processors. A for loop which fetches each references and aggregates the results is called right after.

2.1 Basic Parallel Results

We run our parallel implementation on the largest Enron data to see the most varied result. The timing data is at table 3 and Figure 1 shows the associated speedups. Furthermore, the data for the unparallelized parts of the code are given in table 4.

Processors	Time for `compute_A` (seconds)
1	1327.13
2	883.63
3	664.23
4	549.80
5	452.02
6	402.02
7	356.08
8	325.47

Table 3: Time to run the Enron data set (compute intensities step only)

3 Profiling

To a get a more fine-grained view of where time was spent in the program, we used the `@profile` macro that comes with the Julia standard library to profile the parallel code on both the Yelp and Enron data sets. To ensure fair sampling,

Step	Time (seconds)
Read files	17
Form Q	72
Row Scaling	42
Anchor Words	180
Find top words	1

Table 4: Approximate time to run each non-parallelized step

we ran `mine_topics` once to ensure it was JIT'ed, then profiled a block that called `mine_topics`. This was repeated using 1-8 cores 3 different times and the results were averaged to reduce the influence of anomalous readings. For the Yelp data set samples were collected every 0.001 seconds, and the profiled block called `mine_topics` four times to ensure enough samples were available. For the Enron data set, the block called `mine_topics` once, and samples were collected every 0.1 seconds.

A summary of the results for the Yelp data set is shown in Figure 2. As expected, the portion of the runtime spent in the serial code bottleneck increases with the number of cores, and because the Yelp data set is small, this happens very quickly. At four cores, `choose_anchors_partial` overtakes `compute_A`. The performance bottleneck in `choose_anchors_partial` appears to be the accumulator in the loop. Interestingly, from four to six cores the relative time spent in these sections remains static. This could be due to thread overhead.

A summary of the results for the Enron data set is shown in Figure 3. Here to the portion of the time spent in the serial code increases with the number of cores, but because this data set is so big, it does not overtake `choose_anchors_partial` even with eight cores. This agrees with the speedup plot in Figure 1, which shows no signs of leveling off at eight cores.

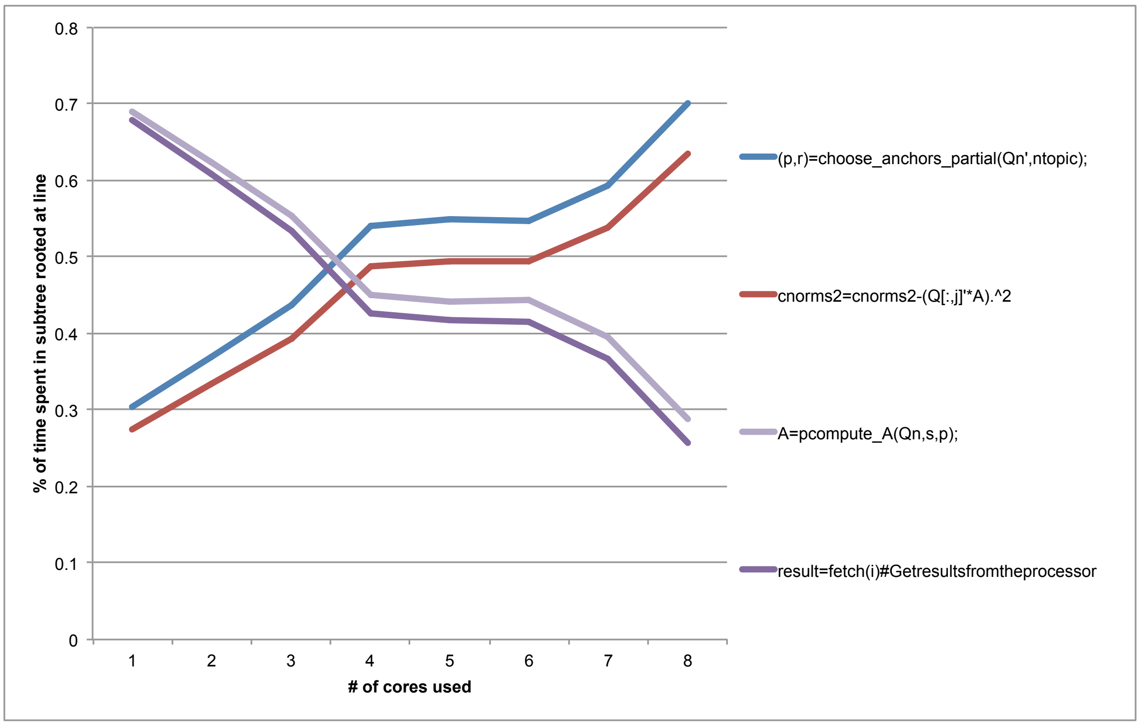

Figure 2: Compute-intensive lines of the program as reported by the Julia sampling profiler running on the Yelp data set.

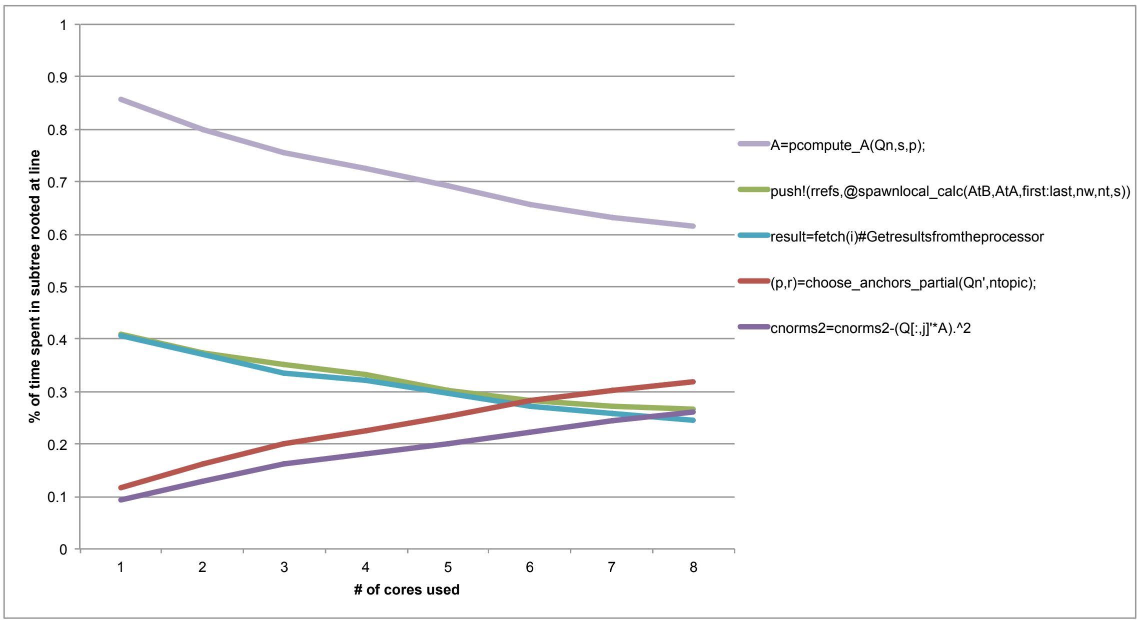

Figure 3: Compute-intensive lines of the program as reported by the Julia sampling profiler running on the Enron data set.