Sterling Baldwin
Dr. J. Challenger
CSCI 551
CSUC Spring ‘16

MultiThreaded Gaussian Elimination

Gaussian elimination allows us to solve for a square matrix of linear equations, but its computational complexity scales as a power of the size of the matrix, meaning it works great for small systems but can run very slowly on larger ones. Fortunately the algorithm can be easily parallelized, giving us the speed boost needed to work with large systems. My implementation is very straightforward in both its memory management as well as its parallelization. Im confident I could increase the speed significantly, but for a first pass I’m happy with the amount of speed up Im getting from running it on many cores.

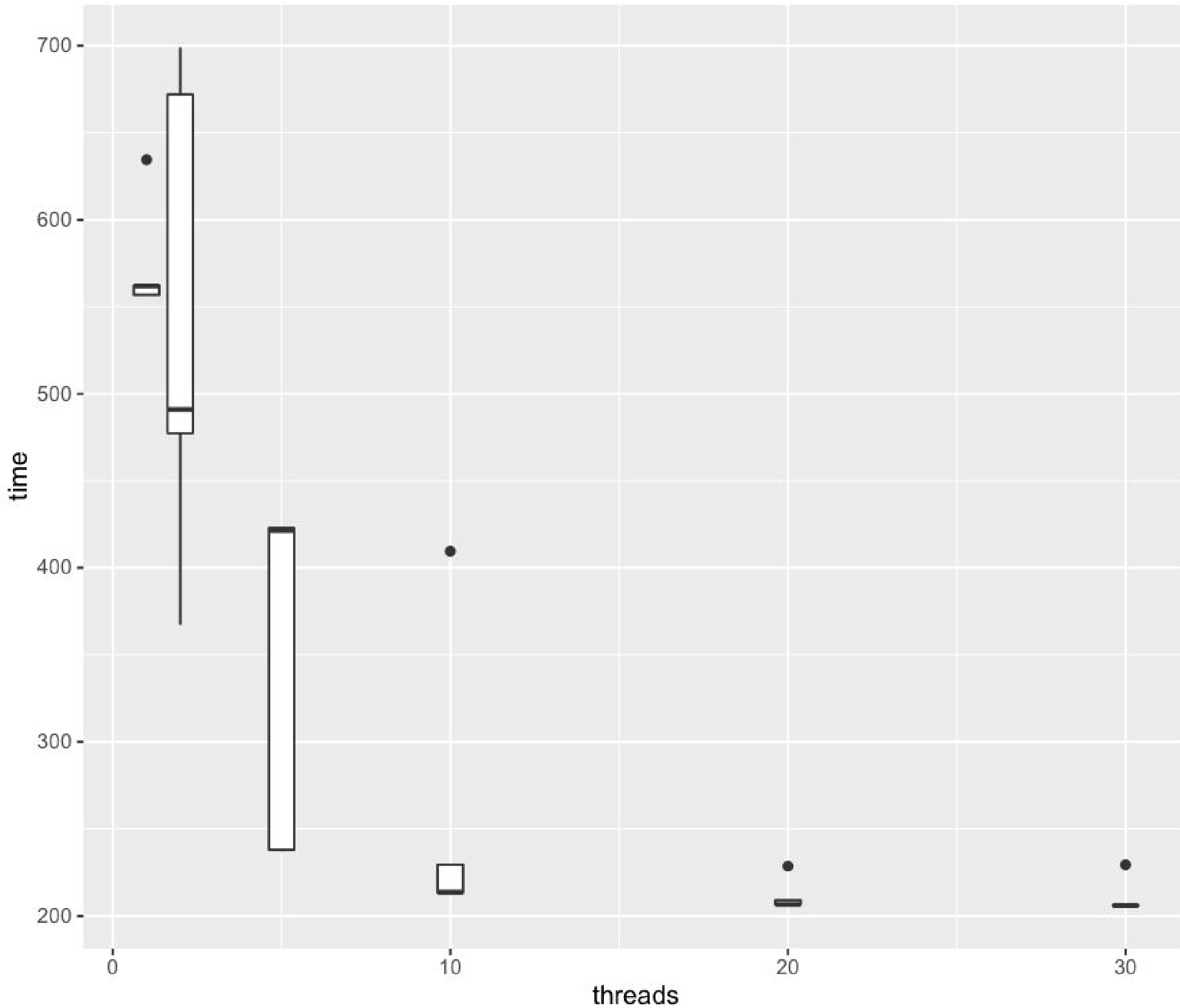
To hold the matrix I choose to use a vector<vector<int> >, which worked nicely for ease of writing and debugging the program, but turned the pivot from what should have been a single pointer swap into a O(n) operation iterating over the entire row. I choose to parallelize finding the maxElement for the pivot because it was a simple operation with no data dependencies that needed to happen in any particular order. The actual swap needed to be run in serial since it was modifying the matrix.

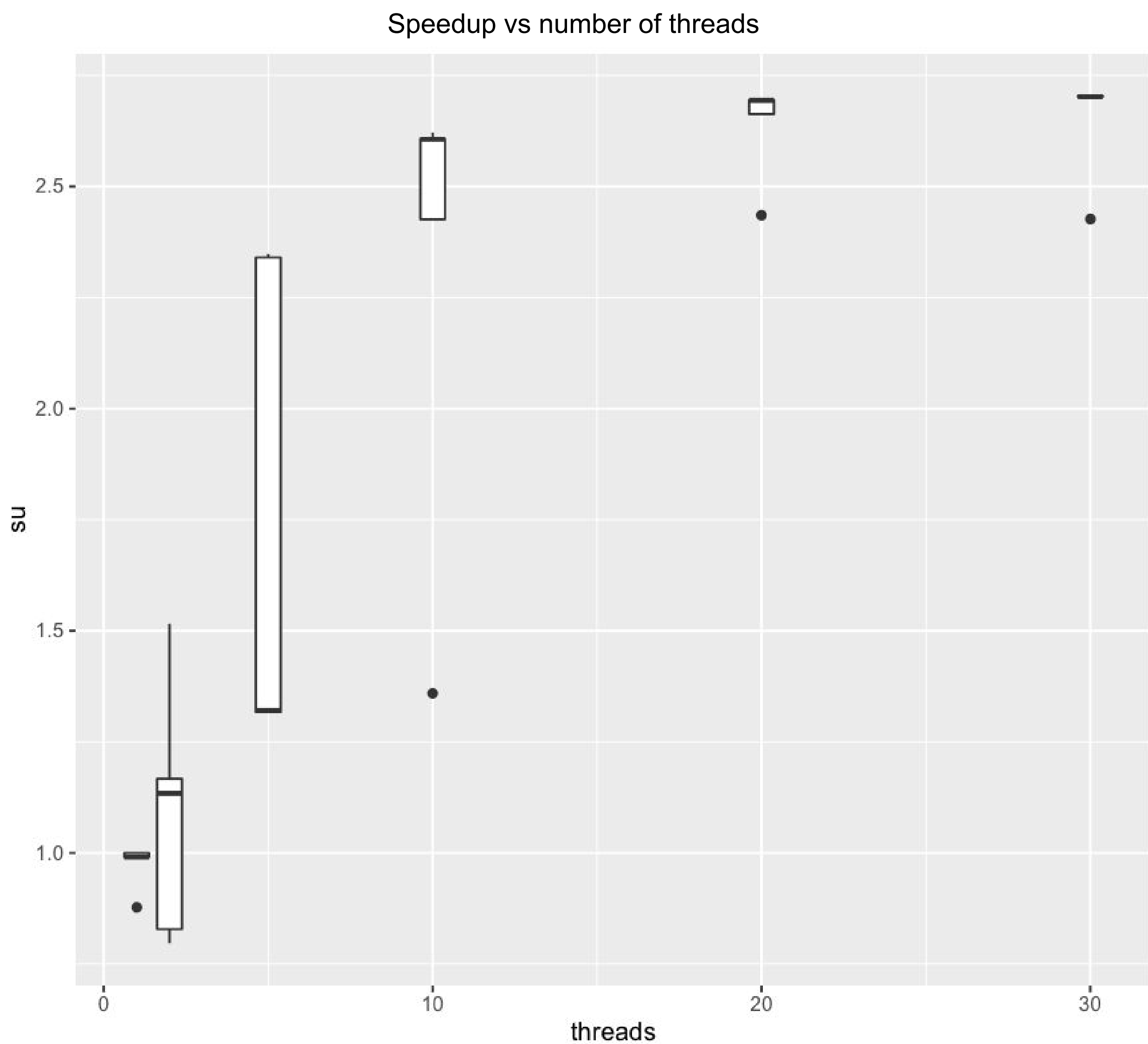
The forward elimination could also be run in parallel. I chose to explicitly set the eliminated value to 0 instead of doing the subtraction due to floating point number errors. We have a proof that the value should be zero, but when I was first testing and using the computed value instead my error was higher than it needed to be.

The back substitution was also parallelized, as it is very straight forward. For all of my parallel sections, any private variables needed were declared inside the section making them private to their thread by default. All variables declared outside of parallel sections were assumed to be public. In all parallel sections, the only variables altered were private, or the global matrix.

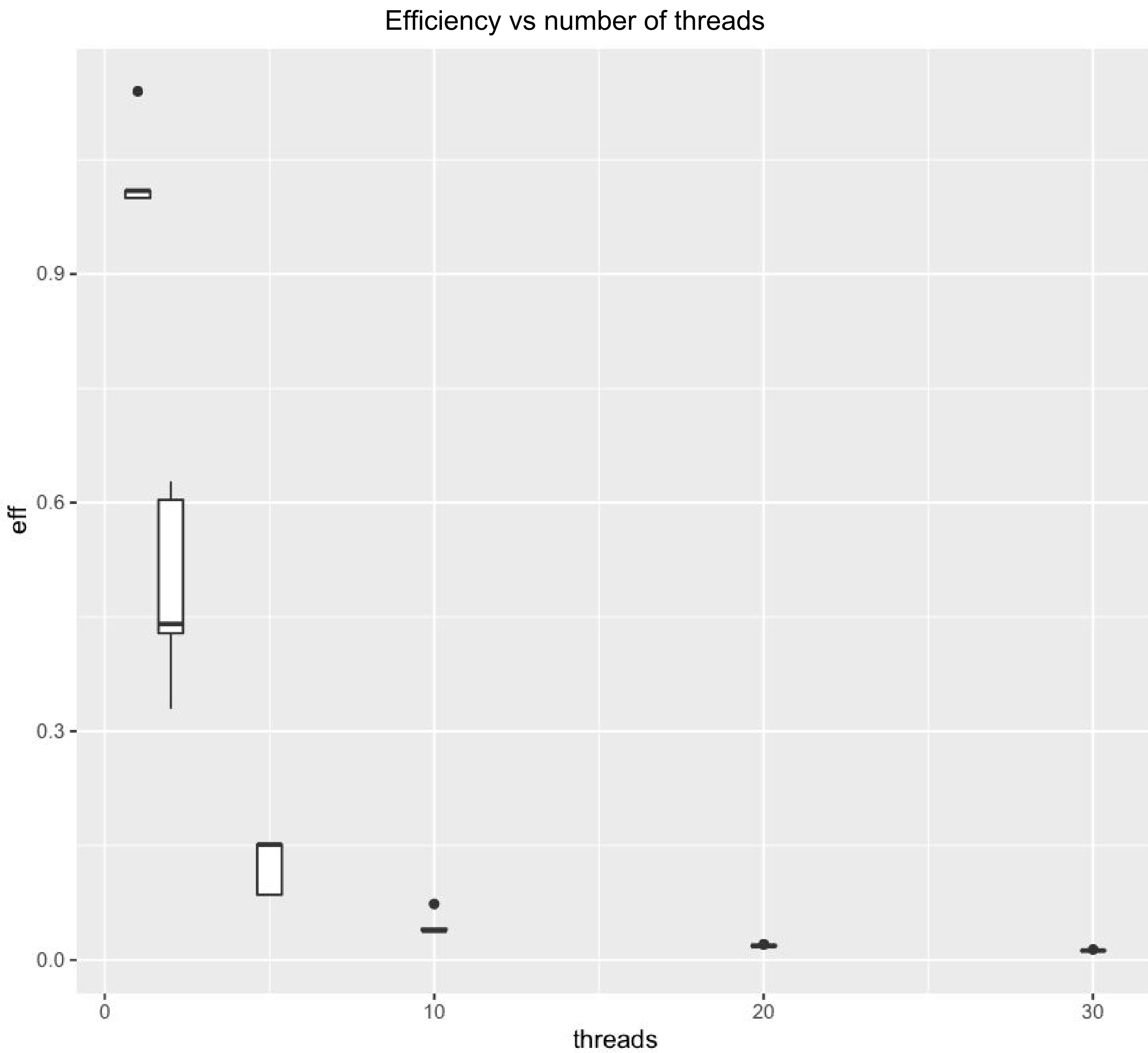
In total I have three parallel sections, one to find the maximum element for the pivot, one for forward elimination, and one for the back substitution. Between the parallel sections the program synchronizes and runs serially.

Time vs number of threads

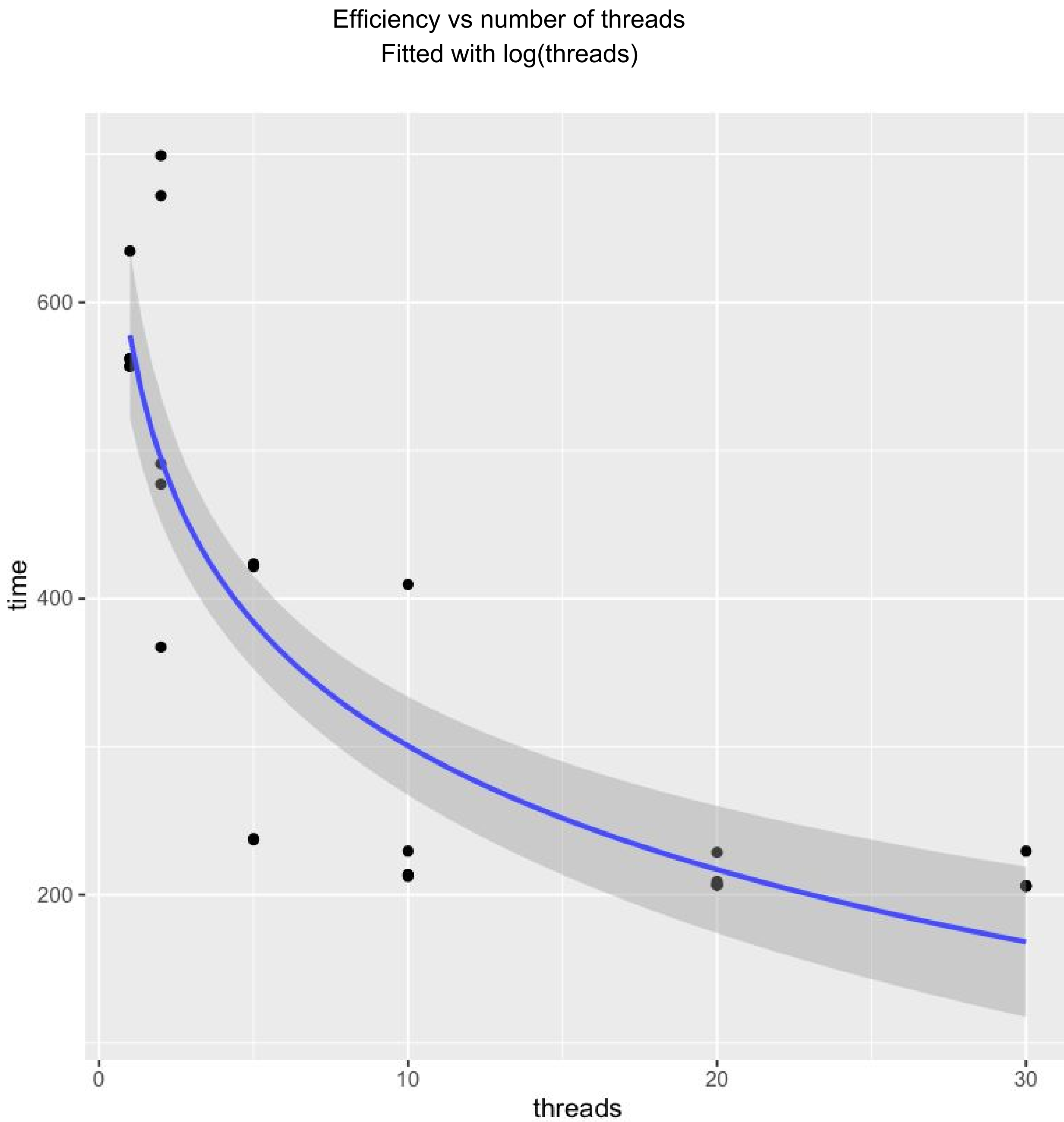




Speedup vs number of threads
su
2.5
2.0
1.5
1.0
0
10
20
30
threads



Efficiency vs number of threads
eff
0.0
0.3
0.6
0.9
threads
0
10
20
30



Efficiency vs number of threads
Fitted with log(threads)
time
600
400
200
0
10
20
30
threads

Run data

run	threads	time	su	eff	l^2norm
1	1	562.04	0.9907302	0.99073020	0.00148092
2	1	556.63	1.0003593	1.00035931	0.0620897
3	1	634.54	0.8775333	0.87753333	0.00123439
4	1	556.83	1.0000000	1.00000000	0.00387337
5	1	561.99	0.9908183	0.99081834	0.00185559
6	2	699.08	0.7965183	0.39825914	0.000824216
7	2	672.06	0.8285421	0.41427105	0.000387558
8	2	491.00	1.1340733	0.56703666	0.00160626
9	2	367.27	1.5161325	0.75806627	0.0026571
10	2	477.33	1.1665514	0.58327572	0.00349673
11	5	421.57	1.3208483	0.26416965	0.00306324
12	5	237.93	2.3403102	0.46806204	0.00234666
13	5	423.31	1.3154190	0.26308379	0.000465094
14	5	237.11	2.3484037	0.46968074	0.000325477
15	5	422.81	1.3169745	0.26339491	0.00306324
16	10	409.59	1.3594814	0.13594814	0.00234666
17	10	212.40	2.6216102	0.26216102	0.000465094
18	10	213.64	2.6063939	0.26063939	0.000325477
19	10	229.51	2.4261688	0.24261688	0.000585629
20	10	213.50	2.6081030	0.26081030	0.000687656
21	20	206.69	2.6940345	0.13470173	0.00145342
22	20	228.63	2.4355072	0.12177536	0.000413767
23	20	206.67	2.6942953	0.13471476	0.000404553
24	20	206.63	2.6948168	0.13474084	0.00618676
25	20	209.06	2.6634937	0.13317469	0.00067923
26	30	206.07	2.7021400	0.09007133	0.00173497
27	30	206.09	2.7018778	0.09006259	0.000646146
28	30	229.46	2.4266975	0.08088992	0.000422524
29	30	205.93	2.7039771	0.09013257	0.000217088
30	30	205.92	2.7041084	0.09013695	0.0239922