Parallelization of Metropolis-Hastings Algorithm Using Python Multiprocessing

Sean McClure, Torin Praeger, James Stratton

May 2, 2020

1 Introduction

Markov Chain Monte Carlo methods, or MCMC, are used to estimate increasingly accurate parameter values. This is done by generating random values from uniform variables to compare parameter values against. The Metropolis-Hastings Algorithm generates multiple MCMC samples. By examining each new MCMC sampling against the last, the algorithm uses an 'accept and reject' method to walk through all possible estimates, only accepting the probable values, thus making more and more refined estimates after every, eventually converging towards the true value of the parameter in question.

We began our work by finding an implementation of the algorithm in Python that we could adapt to be parallel. After we made this implementation parallel, we ran tests on it to get timing data and extrapolate models from that data. All code referred to throughout the project can be found in the attached files or in the corresponding Github repository.

2 The Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm is used in Bayesian statistics, as it relies on the use of a prior ("best guess") distribution to make estimates of values from an unknown posterior distribution. The method is most often used to estimate parameters from distributions that cannot be measured directly through calculus.

In order to use the algorithm two components are needed:

- A prior distribution $f(x)$, the best guess for the posterior distribution.
- A proposal distribution $g(x', x)$, most often a normal distribution.

From this point, the goal is to generate estimates of the parameter and to accept or reject them based on their likelihood. To begin, a random value of x_t is chosen, and a second value x' is generated from the proposal distribution. Next, the likelihood of the estimate at x' is computed by taking the proportion of $\frac{P(x')}{P(x_t)}$, where P is the posterior distribution, which can be found using Bayes law. In order to decide whether to accept or reject the new point x', a random value between 0 and 1 is generated. If the resulting ratio is greater than this generated value, x' is accepted, and is set as the new value of x_t. This method ensures that if the new estimate is more likely than the previous estimate it is automatically accepted. In the case that the old estimate is more likely, there is still a chance that the new estimate is accepted based on the size of the $\frac{P(x')}{P(x_t)}$ proportion. The whole process is continually repeated with the updated arguments. The algorithm above uses the accept and reject method to perform a "random walk" through the sample space.

As new estimates are generated the algorithm accepts more and more probable estimates. These accepted estimates converge towards the true value of the parameter that is being estimated.

We have borrowed our implementation from (Moukarzel, 2018), which is implemented in Python 3.8. Minor changes were made to the code base in order to allow us to effectively parallelize the code.

3 Parallelization

For our parallel implementation, we used the Python Multiprocessing library for two main reasons. The first being that the multiprocessing library is native to Python 3.8. There are a few advantages of this; it requires little setup, and makes our code translatable across systems. The second reason for using the multiprocessing library is that we found there to be a problem with the threading library for Python, where it would not achieve true parallel processing due to global interpreter lock. Further, as the Metropolis-Hastings algorithm is CPU intensive, using GPU methods of parallelization would not be as effective.

After choosing our library we set about implementing a method. Due to the fact that we were working with processes and not threads, there was the issue of sharing memory to tackle as we needed a single array to store results. We solved this by using a Manager process which created an array that all of the subsequent processes could return into. This array would then return back to the main, allowing us to access the data that each process had generated. We then took the individual results of each process for the accepted and averaged them out to get a higher precision in our result.

This allowed us to run simultaneous trials for the number of processes chosen and then average the result, allowing better control of the inherent randomness that comes with Monte Carlo methods,

and a vast amount of speedup.

The pros of this method are obvious, a vast amount of speedup is gained over individual sequential runs, and the final results are more controlled as the algorithm intrinsically runs for a large sample size, allowing variance to be controlled. The cons of this method are subtle, in that each individual run will be different than its compared sequential run, and to run the "same" number of iterations, one needs to give each process a number of iterations equal to desired total iterations divided by the number of processes, losing precision in the final result. Where our implementation truly shines is in its ability to control variance, as the larger the sample the more accurate and precise the results will be. Our method gives the ability to run a large number of samples all at once, making analysis much easier.

4 Analysis

All analysis described below was performed in R, with the complete code available in the attached files and the corresponding Github repository.

4.1 Timing Data

To begin the analysis, we ran the program with iteration counts of powers of two, from 2^{10} to 2^{24}. The code was run with 1, 4, 8, 16, and 32 cores, each with one process per core. Each amount of cores was run for every number of iterations. The machine used for timing only had 16 physical cores, so the 32 core implementation was run on virtual cores. The initial results are visible in the left graph of *figure 1*. From this initial graph the speedup gained from multiprocessing is visible. The sequential code runs far slower than the multiprocessor implementations. The long term effects of overhead are visible; as the number of cores increases, the increase of performance gained appears to decrease. Further, the speedup gained from multiprocessing can actually be negative for lower iteration counts, speaking more to the overhead cost of the algorithms. It is difficult, however, to make many conclusions from this graph.

To clarify the results, the natural log of the timing results was graphed against the log base two of the number of iterations, displayed in the right of *figure 1*. This graph more effectively displays the performance over time of each implementation. Overhead at low iterations is visible, as a larger number of cores corresponds with longer completion times. Each parallel implementation reaches an inflection point at which the overhead becomes negligible, and the completion times begin to increase. This inflection point is explored in more detail later.

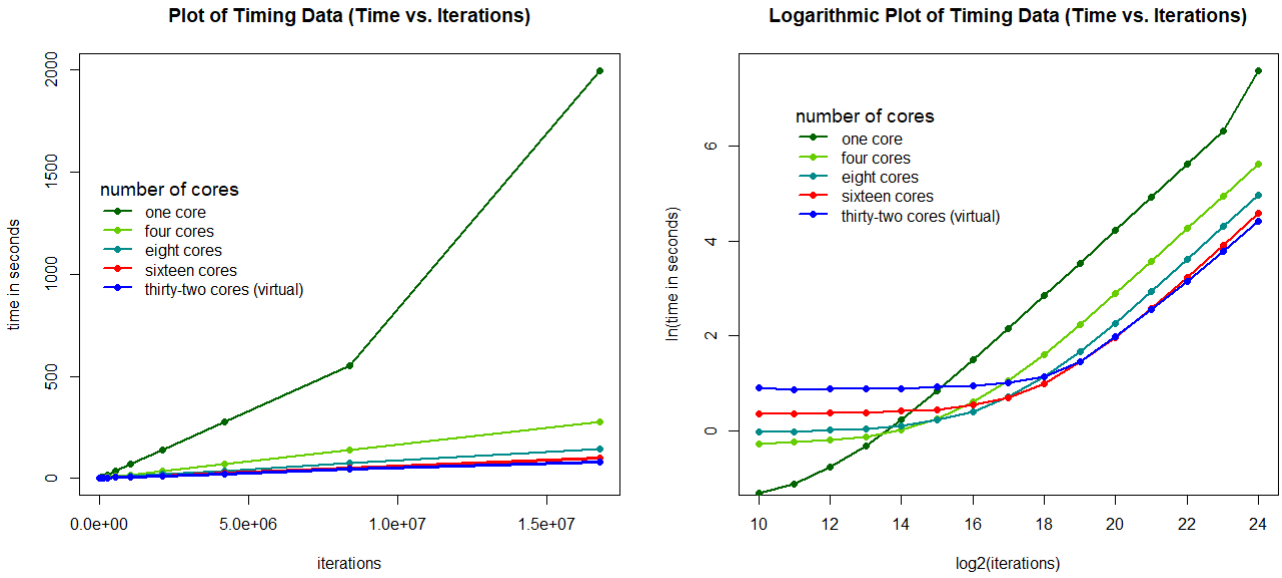

Figure 1: Timing Data

The diminishing returns from the number of cores is also visible in this graph. The performance increase from 1 to 4 cores is more noticeable than the increase between 8 and 16 cores. The 16 and 32 core implementations have similar long-run performance, which may be due to the use of virtual cores.

4.2 Speedup

The speedup gained from the multiprocessing is displayed in *figure 2* and *table 1*. We believe the last sequential data point to be an outlier, (expanded upon in the discussion section), so we took the second largest speedup for each implementation to be representative of the true speedup. The speedup for each parallel version has negative speedup at low iterations due to overhead. The breakeven point represents the number of iterations at which the parallel implementations begin to outperform the sequential version, which is reached between 2^{13} and 2^{15} iterations.

The speedup of each parallel version increases steadily, before leveling off. The percentage of potential speedup achieved decreases as the number of cores increases. For example, the potential speedup of the 4 core implementation is 4, and the speedup achieved is close to this hypothetical value at 3.95, giving us 98.75% of our potential speedup. The observed speedup for the 16 core implementation is only 10.99, which is only 68.69% of the potential speedup. The results are even worse for the 32 core implementation, although this may be partially due to the use of virtual cores. Again, this depreciation of performance over the increase of iterations is likely to due to overhead.

Table 1: Speedup from Multiprocessing

Number of Cores	Max Speedup	2nd Largest Speedup
Four Cores	7.18	3.95
Eight Cores	13.78	7.47
Sixteen Cores	20.51	10.99
Thirty-two Cores	24.18	12.42

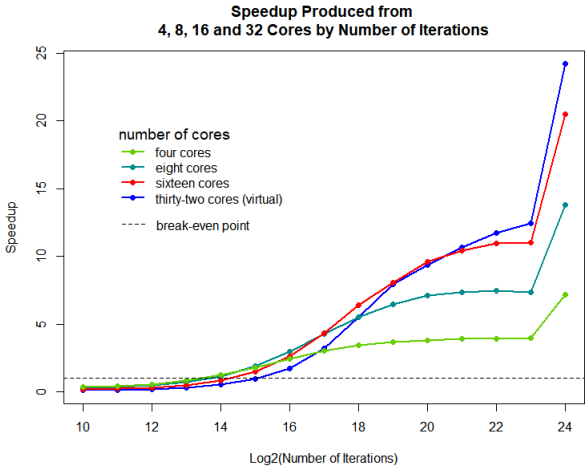

Figure 2: Speedup Graph

4.3 Regression Modeling

After collecting the timing data, two mathematical models were fit for each parallel implementation. The reason for this was twofold. First, the generated equations allow the results of the testing to be further analyzed. Secondly, the models can be used to predict the results of potential timing results without physically running the program.

As displayed in figure 2, the logarithm of the timing data trends linearly at larger numbers of iterations. This made it possible to use log-linear regression to model the results. The first model for each implementation included all the data from each respective version, and are visible in *figures 3 and 4*, (located in *appendix A*). Due to the non-linearity of the results at low iterations these models are not very accurate.

To remedy this, a second model was fit for each implementation that only includes data from the point at which the results show linearity. These models are also shown in *figures 3 and 4*, and are much more accurate, however, restricted in domain. The models can only be used for iterations beyond the inflection point of each curve (indicated for each curve in the respective graphs.) Generally, the inflection point for the graph of ln(y) appears to be located at $2^(ln2(n) + 14)$ where n is the number of processes and y is the time in seconds. More analysis will need to be conducted to determine whether this is a true trend, or a coincidence in our observed data. By averaging the constants observed in the 4 linear models, we can find our equation for the natural log of the time in seconds, that is, $ln(y) = 0.629161 * log2(x) - 10.2286845$. This equation, solved for $x's$ greater than $2^(ln2(n) + 14)$ will give a rough estimate of the time taken to run this algorithm for any combination of values for processes and iterations. The specific equations for each model are found inside the respective graphs in *figures 3 and 4*.

4.4 Application

The models allow us to further analyze the behavior of our timing data. The slope of the regression lines indicate how the completion times of each parallel implementation respond to a change in the number of iterations run. For example, according to *figure 4*, for the 32 core secondary model we are 95% confident that for every power of 2 increase in iterations the completion time will increase by between $e^{0.594}$ and $e^{0.670}$ seconds. This range is quite large, which is due to limited sample size. If we were to collect more data, these confidence intervals could be diminished.

Further, by comparing the slopes of the secondary models we can compare the growth rates of the parallel implementations. From the equations in *figures 3 and 4* we can see that the slopes are comparable, indicating that after reaching the overhead inflection point, the timing results increase at similar rates. It can be assumed that the difference in the performance of the models is due to the difference of the y-intercepts.

As previously mentioned these models can be used to predict completion times for theoretical tests. For example, we can compute the expected completion times for the 8 and 32 core implementations at 2^{30} iterations. Once again, the range is large due to the small sample size, but it is expected that the 32 core implementation will complete 1407.7 seconds faster than the 8 core version.

Table 2: Expected Results at 2^{30} Iterations

Number or Cores	**Expected Completion Time**	**95% Confidence Interval**
8 Cores	5458.13 *s*	$(3249.99, 9166.54)$ *s*
32 Cores	4145.56 *s*	$(2873.93, 5979.82)$ *s*

5 Conclusion

We found that the overhead cost of parallelizing using the Python 3.8 Multiprocessing library can be very expensive. As can be seen in our data, for iteration counts under 2^{13} sequential code will run the fastest, as the overhead cost of the processes slows down to code considerably. This shows us that the parallelization of programs is really only effective for large batches of work, and that these conditions should be analyzed before setting out to make a program run in parallel, as the effort put in may actually give negative results.

A

Log-linear Regression Equations and Figures

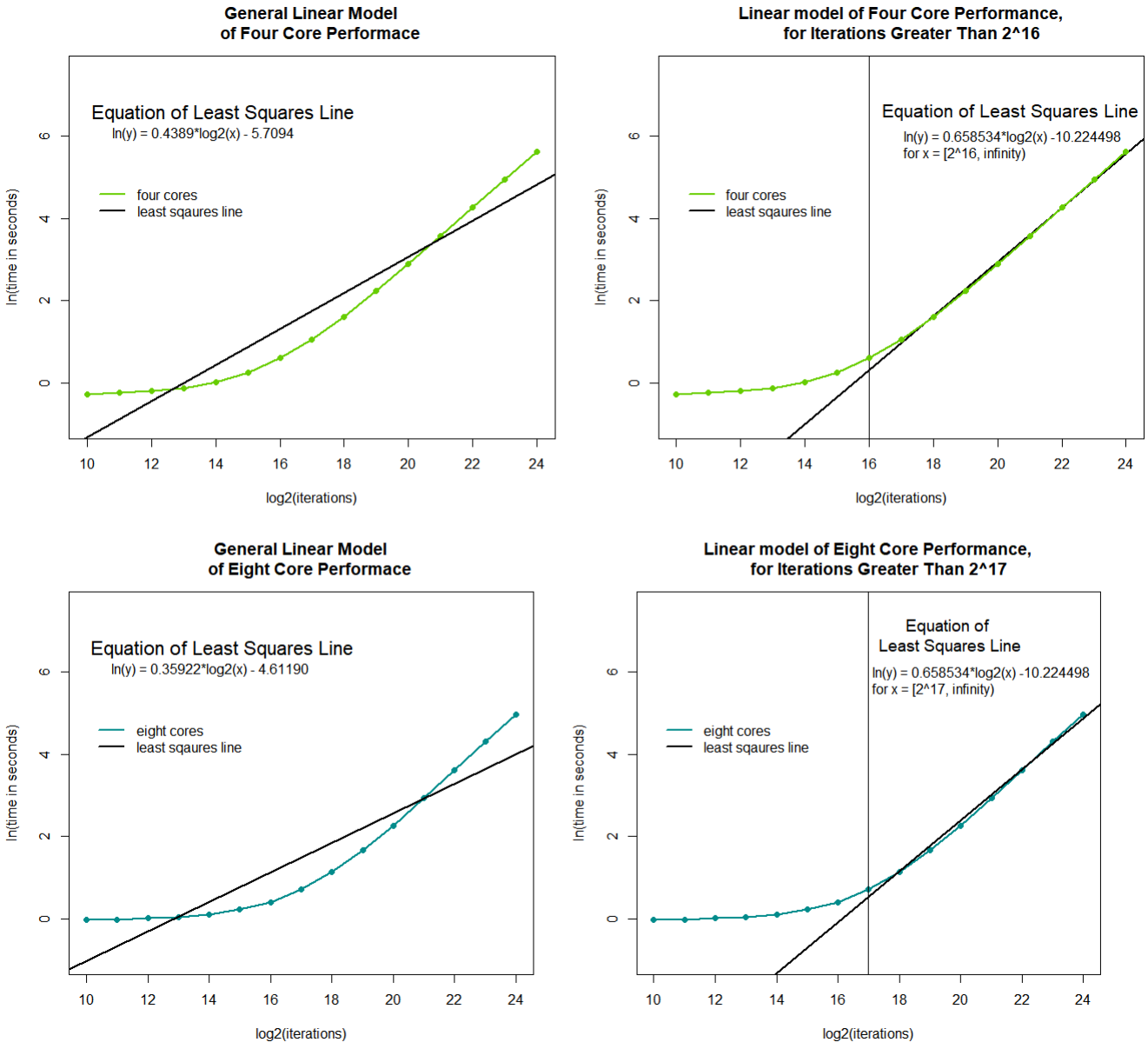

Figure 3: Regression models of 4 and 8 core implementations

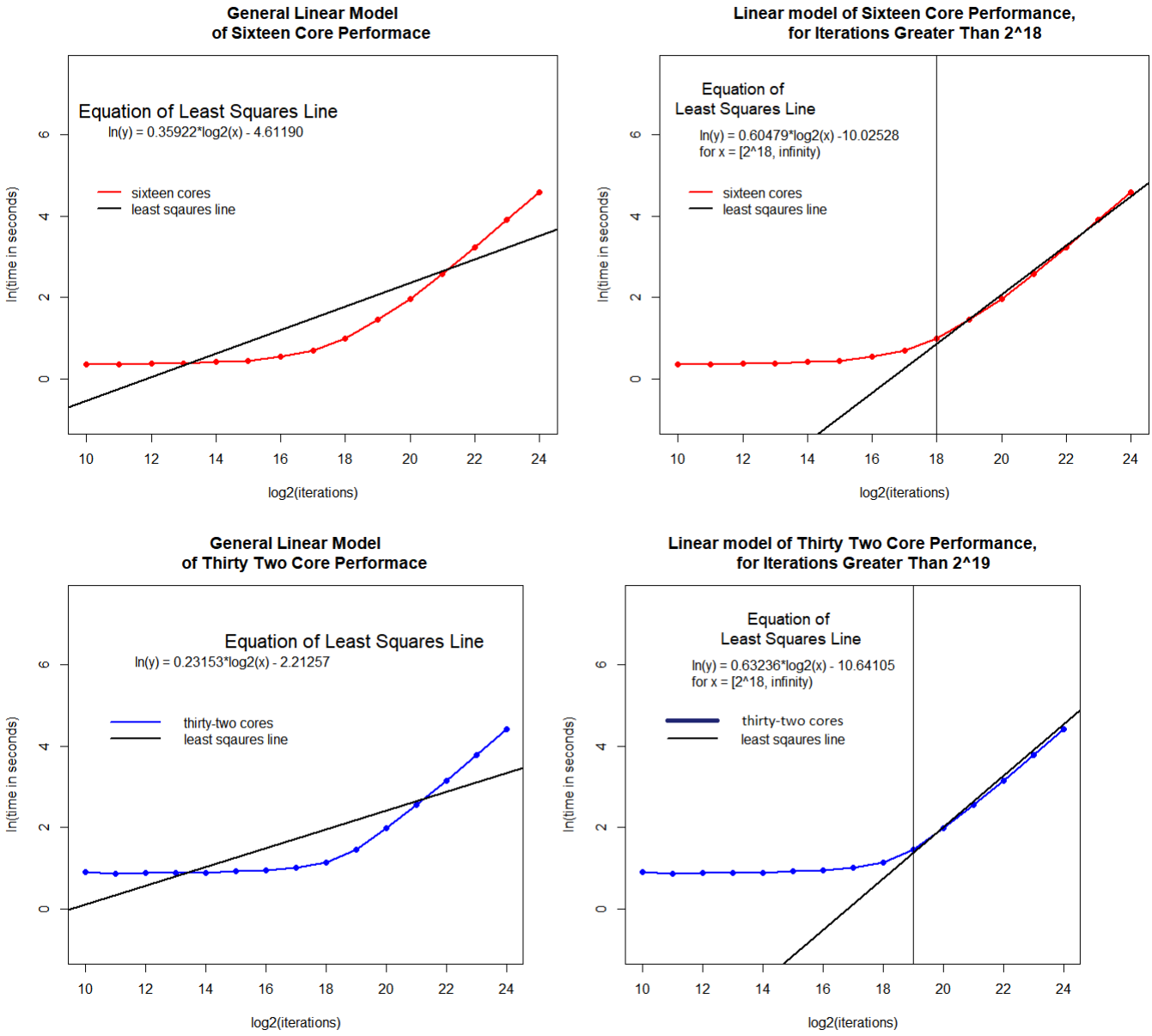

Figure 4: Regression models of 4 and 8 core implementations

References

Hastings, W. K. 1970. “Monte Carlo sampling methods using Markov chains and their applications.” *Biometrika* 57(1):97–109.
URL: *https://doi.org/10.1093/biomet/57.1.97*

Moukarzel, Joseph. 2018. “From Scratch: Bayesian Inference, Markov Chain Monte Carlo and Metropolis Hastings, in python.”.
URL: *https://towardsdatascience.com/from-scratch-bayesian-inference-markov-chain-monte-carlo-and-metropolis-hastings-in-python-ef21a29e25a*

Stephens, Matthew. 2017. “Simple Examples of Metropolis–Hastings Algorithm.”.
URL: *https://stephens999.github.io/fiveMinuteStats/MH-examples1.html*

Yildirim, Ilker. 2012. “Bayesian Inference: Metropolis-Hastings Sampling.”.
URL: *http://www.mit.edu/ ilkery/papers/MetropolisHastingsSampling.pdf*