Simulation of Markov Chain Monte Carlo Methods through OpenMP and MPI

Pedro H. F. dos Santos (phf254)
Noah W. Lindley (nwl268)

Abstract

In the past 20 years, due to the advance in hardware, Bayesian statistical methods have returned to focus. Usually those methods rely on Markov Chain Monte Carlo simulation techniques, since the estimation of the parameters of the models must be done through sampling in most cases. The idea is that if the chain run is long enough, the sample will be a good representation of the true distribution. However, since the distributions of the parameters may have multiple modes (especially in high dimensional settings), the sample may get stuck and fail to represent relevant parts of the parameter space. The creation of the Markov Chain, by definition, is sequential, but dealing with multiple chains to assess the results is usually time consuming. With parallel computation we have the possibility to start the chain in multiple places. Our project implemented a parallel MCMC sampling by using both the OpenMP and MPI techniques, and provided a weak scaling report of the methods to evaluate the efficiency of the code. All the code is open and available at GitLab.

Introduction

Markov Chain Monte Carlo methods are very powerful tools, and are the reason that Bayesian statistics advanced so much in the past few decades. The Bayesian approach consists of looking at the distribution of the parameters (θ in this example) given the data (y in this example), such that

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)},$$

where $p(\theta|y)$ is the posterior distribution, $p(y|\theta)$ is the likelihood, $p(\theta)$ is the prior distribution, and the marginal distribution $p(y)$ is the normalizing constant. However, usually it is not possible to calculate $p(y)$, and we only have the numerator, which is proportional to the posterior, such that

$$p(\theta|y) \propto p(y|\theta)p(\theta).$$

By using the Metropolis-Hastings (M-H) algorithm we can use transition probabilities of proposal distributions, and either accept or reject the new samples. If the MCMC runs long enough, then we start sampling from our posterior distribution, and by having large samples, we can make inference over the distribution of the parameters.

Another technique that is usually applied is the Gibbs sampler, which uses the full conditional posteriors to sample the parameters one at a time. The Gibbs sampler can either be used when the full conditional is available, or by using a M-H for the parameters.

In our project, we decided to provide an example that can be applied in most cases, since the M-H algorithm is the most widely used MCMC technique. Our project uses example 6.3 of the book Computational Bayesian Statistics (Turkman

et al., 2019). The example is a logistic regression, and the posterior distribution, in this case, has only one mode. The idea is to implement the technique by starting multiple chains at multiple places, and keep the chains running long enough such that they should converge to the posterior distribution. On the case of posteriors with multiple modes, some chains might get stuck in a mode and be unable to properly represent the posterior distribution, so running multiple chains is usually desirable in the Bayesian setting. Our example, however, is not focused on this behaviour, since the idea is to provide the tools required for the practitioners to properly implement parallel programming in their models.

Objectives

Our main objective is to parallelize a MCMC scheme in the most general way possible in C++, serving as a guide to any practitioner that might need a similar setting. Our focus in this case is the use of a Metropolis-Hastings algorithm, since this is the most common technique for MCMC sampling.

We are implementing the algorithm in both the OpenMP and MPI settings, and then analyzing the efficiency and speedup over different number of processors. Also, we provide visualization of the convergence of the multiple chains that have been created by the multiple processors.

Implementation

Our code is open and available on GitLab (https://gitlab.com/whysin/metropolis-hastings/).

Our implementation is based on example 6.3 of book Computational Bayesian Statistics (Turkman et al., 2019). The data consists of a toxicity study of some new compound that had multiple doses administered to batches of animals. In our case, we have 4 batches. Let x_i be the dose for the ith batch, n_i the number of animals in the ith batch, and y_i the number of animals in the ith batch that have shown a

response. The data for the study can be seen on Table 1.

Batch	x_i	n_i	y_i
1	-0.86	6	1
2	-0.30	5	2
3	-0.05	5	3
4	0.73	5	5

Table 1: Data - Toxicity study

Let us define that y_i follows a Binomial distribution since it is counting the number of responses in a population, such that

$$y_i \sim \mathcal{B}in(n_i, \pi_i),$$

where π_i is defined as a logistic regression, such that

$$\pi_i = \frac{1}{1 + \exp(\alpha + \beta x_i)},$$

where $\theta = (\alpha, \beta)^T$ is the vector of unknown parameters. Now let us assume that $\theta = (\alpha, \beta)^T$ follows a multivariate normal distribution, such that we have the prior

$$\theta = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 100 & 0 \\ 0 & 100 \end{pmatrix} \right),$$

which basically is a vague prior for the parameters.

With this information we can calculate the proportional part of the posterior distribution. Let us define that

$$p(\alpha, \beta|y) \propto p(y|\alpha, \beta)p(\alpha, \beta) = h(\alpha, \beta|y),$$

$$h(\alpha, \beta|y) = \prod_{i=1}^{4} \left[\binom{n_i}{y_i} \pi_i^{y_i} (1-\pi_i)^{n_i - y_i} \right] \frac{1}{2\pi \times 100} \exp\left(-\frac{(\alpha^2 + \beta^2)}{200} \right),$$

then applying the log,

$$\log h(\alpha, \beta|y) = -\log(200\pi) - \frac{(\alpha^2 + \beta^2)}{200} + \\ + \sum_{i=1}^{4} \log\left(\binom{n_i}{y_i} \right) + \sum_{i=1}^{4} y_i \log(\pi_i) + \sum_{i=1}^{4} (n_i - y_i) \log(1-\pi_i).$$

We have chosen to use the following transitions:

$$q(\alpha^*|\alpha) = \mathcal{N}(\alpha, 1), \qquad q(\beta^*|\beta) = \mathcal{N}(\beta, 1).$$

They are stated this way because $q(\beta^*|\beta) = q(\beta|\beta^*)$ and $q(\alpha^*|\alpha) = q(\alpha|\alpha^*)$, leading to simplification of the M-H ratio. Also, since we do not want to only accept both parameters together, they will be implemented in a Gibbs setting. Since the parameters do not depend on each other in this specific case, they could also be sampled in parallel, however, since in most cases there is some dependence, we decided to make a general implementation of what a parallel MCMC setting would look like. Our setting can be applied to any Metropolis-Hastings and Gibbs Sampling by just changing the transition probabilities and the posterior distribution. Also, our initial guesses were sampled from the prior distribution. The MCMC scheme is described in Algorithm 1.

Algorithm 1 Sample of size m from $p(\alpha, \beta|y)$

Require: Sample size: m; Initial guess: $(\alpha^{(0)}, \beta^{(0)})$

for $i \in \{1, ...m\}$ **do**

Sample $\alpha^* \sim q\left(\alpha^*|\alpha^{(i-1)}\right)$

Accept α^* with probability $\xi = \min\left(1, \frac{h(\alpha^*, \beta^{i-1}|y)q(\alpha^{(i-1)}|\alpha^*)}{h(\alpha^{i-1}, \beta^{i-1}|y)q(\alpha^*|\alpha^{(i-1)})}\right)$

if Accepted **then**

$\alpha^{(i)} = \alpha^*$

else

$\alpha^{(i)} = \alpha^{(i-1)}$

end if

Sample $\beta^* \sim q\left(\beta^*|\beta^{(i-1)}\right)$

Accept β^* with probability $\xi = \min\left(1, \frac{h(\beta^*, \alpha^i|y)q(\beta^{(i-1)}|\beta^*)}{h(\beta^{i-1}, \alpha^i|y)q(\beta^*|\beta^{(i-1)})}\right)$

if Accepted **then**

$\beta^{(i)} = \beta^*$

else

$\beta^{(i)} = \beta^{(i-1)}$

end if

end for

return $\left(\alpha^{(1)}, ..., \alpha^{(m)}, \beta^{(1)}, ..., \beta^{(m)}\right)$

For the generation of random numbers and uniform distributions in the experiment a library called TRNG was used. Tina's Random Number Generator (TRNG) is a pseudo random number generator that allows the program to create multiple streams of random numbers for multi-threaded applications. It also doesn't depend on any specific parallel techniques meaning that we can use it with any threading library or MPI. The reasoning for using this library is that in our experiment we needed to have parameter sets that would guarantee a long period of good statistical properties, optimized speed, and it provides methods for creating random variables with different types of distributions. The random variables were implemented such that both methods would generated exactly the same numbers on each chain, which

means that the chains are directly comparable.

OpenMP

```
int metropolis_hastings(uint64_t sample_count, uint64_t burnin){
    std::vector<double> alpha(sample_count), beta(sample_count);
    int size = omp_get_max_threads();

    //Creating independent process random number streams
    trng::yarn2 stream[size];
        for (int i = 0; i < size; i++){
                stream[i].split(size,i);
        }

    //variables
    int rank;
    uint64_t accepted_count;
    double old_alpha, old_beta, new_alpha, new_beta, new_density,
    ↪  old_density, difference;

    #pragma omp parallel private(rank, old_alpha, old_beta, new_alpha,
    ↪  new_beta, new_density, old_density, difference, accepted_count,
    ↪  alpha, beta)
    {
        alpha = std::vector<double> (sample_count);
        beta = std::vector<double> (sample_count);
        rank = omp_get_thread_num();
        //Randomize initial starts of theta one and two
        alpha[0] = norm_inits(stream[rank]);
        beta[0] = norm_inits(stream[rank]);

        accepted_count = 0;
        //doing a random walk
        for(uint64_t sample_idx = 1; sample_idx <= sample_count;
        ↪  ++sample_idx){

```

```
29                //Get the old alpha and old beta
30                old_alpha = alpha[sample_idx - 1];
31                old_beta = beta[sample_idx - 1];
32
33                //Propose a new position
34                new_alpha = old_alpha + normal_dist(stream[rank]);
35                new_beta = old_beta + normal_dist(stream[rank]);
36
37                //Calculate the log-density for the new and old alpha & beta
38                new_density = log_density(new_alpha, new_beta);
39                old_density = log_density(old_alpha, old_beta);
40
41                //Calculating the log-acceptance probability
42                difference = new_density - old_density;
43
44                // Metropolis Accept/Reject
45
46                // Accept and set the proposed as the current position
47                if(log(uniform_real_dist(stream[rank])) < difference){
48                    alpha[sample_idx] = new_alpha;
49                    beta[sample_idx] = new_beta;
50                    ++accepted_count;
51                // Reject and set the old as the current position
52                }else{
53                    alpha[sample_idx] = old_alpha;
54                    beta[sample_idx] = old_beta;
55                }
56            }
57            #pragma omp barrier // synchronize threads
58        }
59        return size;
60    }
```

MPI

```
int metropolis_hastings(uint64_t sample_count, uint64_t burnin){

    int rank, size;
    MPI_Init(0, 0);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    std::vector<double> alpha(sample_count), beta(sample_count);
    uint64_t accepted_count;
    double old_alpha, old_beta, new_alpha, new_beta, new_density,
    ↪  old_density, difference;

    //Creating independent process random number streams
    trng::yarn2 stream[size];
    stream[rank].split(size,rank);

    alpha = std::vector<double> (sample_count);
    beta = std::vector<double> (sample_count);
    //Randomize initial starts of theta one and two
    alpha[0] = norm_inits(stream[rank]);
    beta[0] = norm_inits(stream[rank]);

    accepted_count = 0;
    //doing a random walk
    for(uint64_t sample_idx = 1; sample_idx <= sample_count; ++sample_idx){

        //Get the old alpha and old beta
        old_alpha = alpha[sample_idx - 1];
        old_beta = beta[sample_idx - 1];

        //Propose a new position
        new_alpha = old_alpha + normal_dist(stream[rank]);
        new_beta = old_beta + normal_dist(stream[rank]);

```

```
34              //Calculate the log-density for the new and old alpha & beta
35              new_density = log_density(new_alpha, new_beta);
36              old_density = log_density(old_alpha, old_beta);
37
38              //Calculating the log-acceptance probability
39              difference = new_density - old_density;
40
41              /*Metropolis Accept/Reject*/
42
43              // Accept and set the proposed as the current position
44              if(log(uniform_real_dist(stream[rank])) < difference){
45                  alpha[sample_idx] = new_alpha;
46                  beta[sample_idx] = new_beta;
47                  ++accepted_count;
48              // Reject and set the old as the current position
49              }else{
50                  alpha[sample_idx] = old_alpha;
51                  beta[sample_idx] = old_beta;
52              }
53          }
54          MPI_Barrier(MPI_COMM_WORLD); // wait for all threads to be done
55          MPI_Finalize();
56          return size;
57      }
```

After viewing the code snippets from our implementations in OpenMP and MPI it is possible to see that the ”random walk” for-loops are not parallelized in the traditional way. The reasoning for not doing the ”traditional” parallelization is that Markov Chains have to be done serially because each chain is dependent on the previous value sampled. So, for our experiment we wanted each processor to run starting with its own unique starting values and for-loop so that it could compute its own α and β to be compared to other samples computed in the other processors. This was done so that we could see if the samples of α and β approached similar

values in parallel.

Results

Some experiments were performed by using the implementation discussed in the previous section. For our examples, each MCMC sampled 100,000 values for each parameter in each processor. The initial guess from each processor has been sampled from the prior distribution. The numbers generated on both methods should be the same, so our analysis will also be helpful to show these results.

First we need to evaluate if the chains are working as intended. As it is possible to see on Figures 1 and 2, the different chains converged quickly to the true posterior distribution. Each colored line is a chain, and the points are the starting values for each chain. After around 500 iterations the chains already converged. After 1000 iterations, sometimes values that are less likely to occur were accepted, but the chains would quickly converge back to the region of high probability.

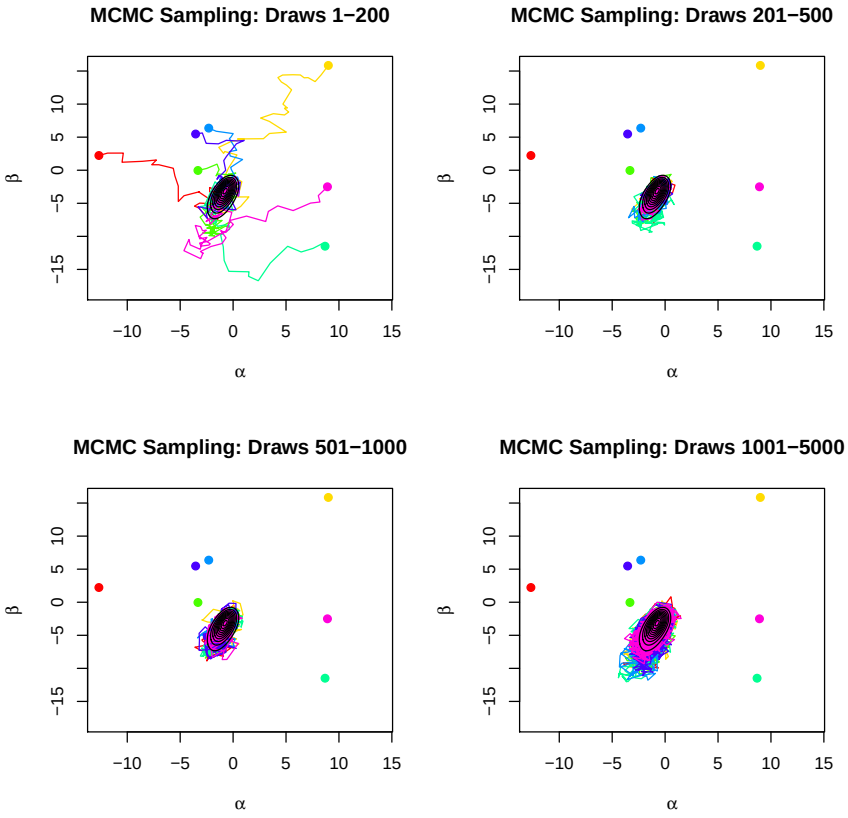

Figure 1. Convergence of Multiple Chains - MPI

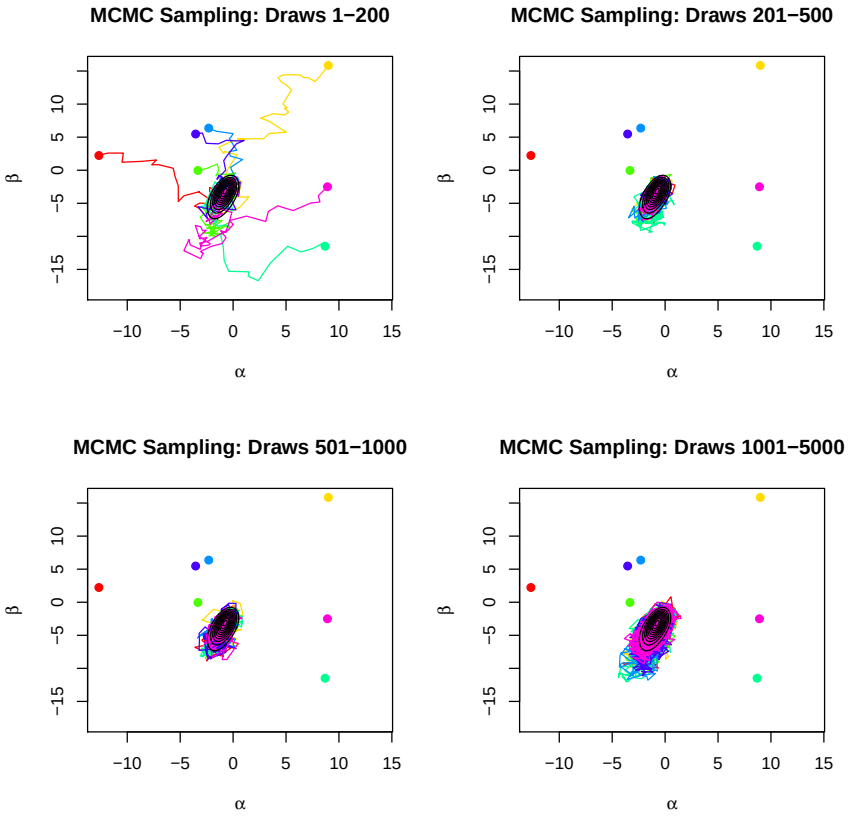

Figure 2. Convergence of Multiple Chains - OpenMP

Similarly, Figures 3 and 4 show the boxplot of the parameters. The idea of these figures is to verify if the chains are getting distributions that are different among each other. The results were similar for all 8 chains in this experiment.

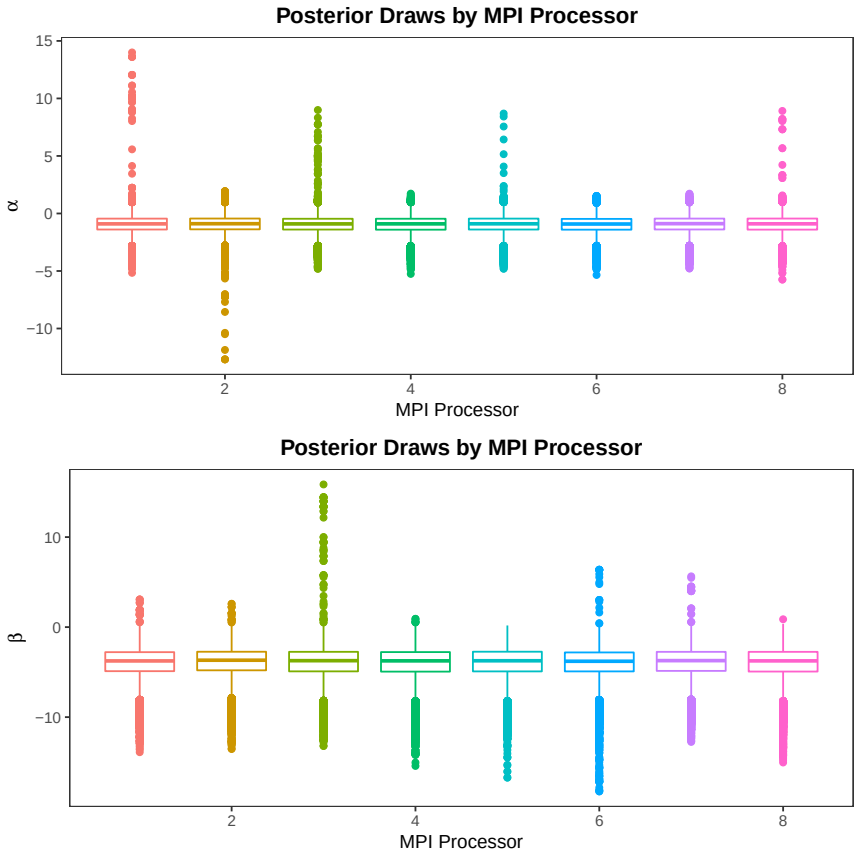

Figure 3. Boxplots of Posterior Draws - MPI

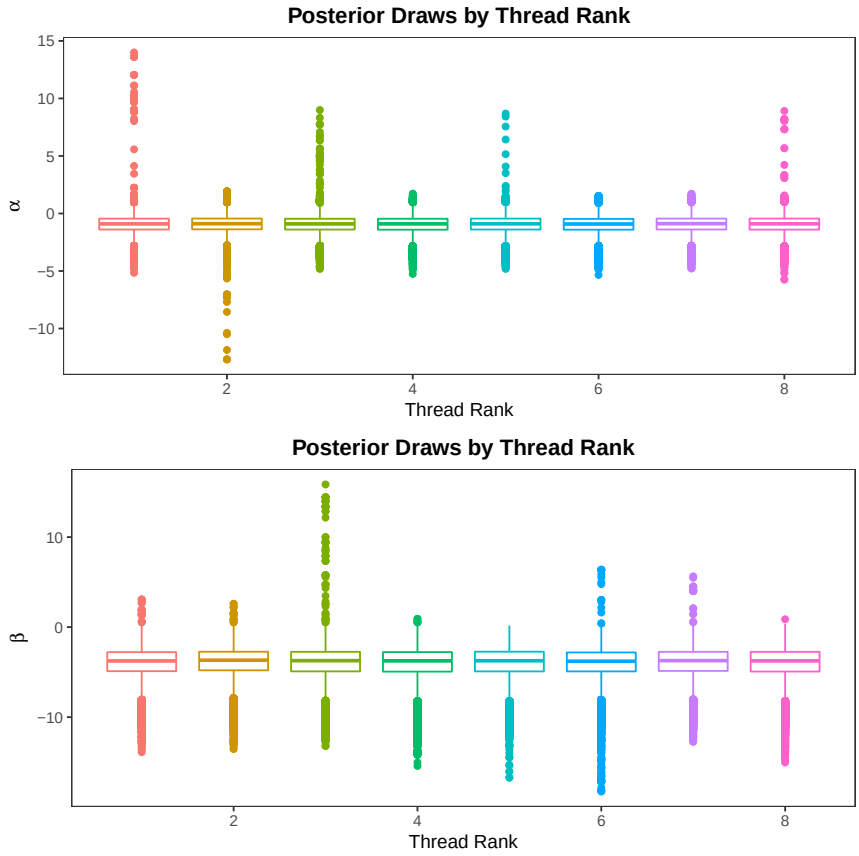

Figure 4. Boxplots of Posterior Draws - OpenMP

Furthermore, on Figures 5 and 6 we have the traceplots for both methods studied. As it can be seem on those graphs, the results were consistent among threads, and the samples seemed to properly investigate the surroundings of where each chain converged.

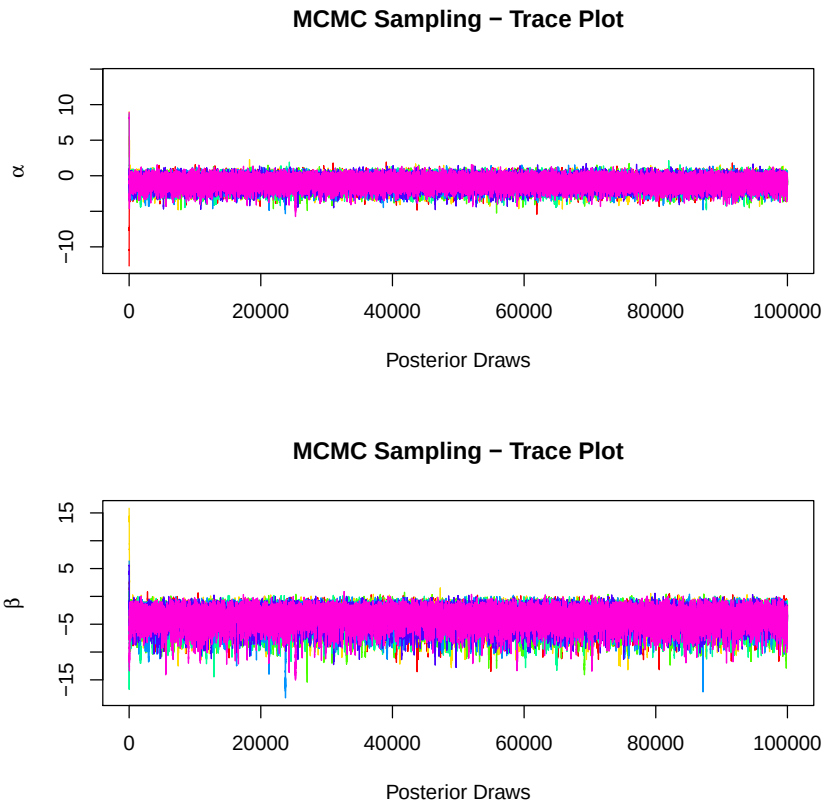

Figure 5. Traceplots of Posterior Draws - MPI

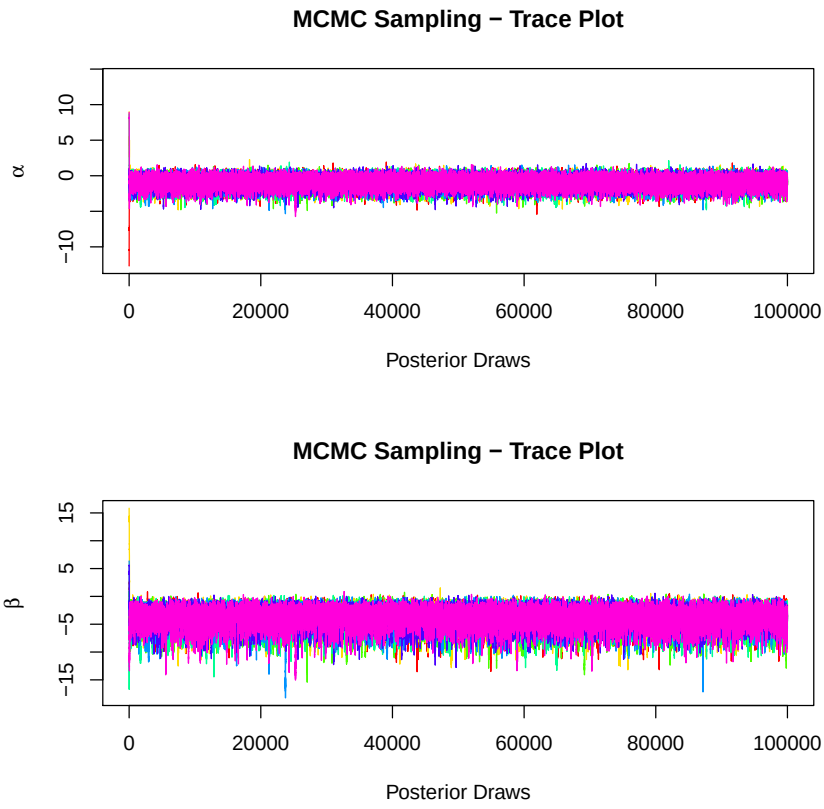

Figure 6. Traceplots of Posterior Draws - OpenMP

Now that we know that our method works as it was intended, then we can compare the efficiency of the methods on Figure 7. The red horizontal line is the perfect efficiency. As it can be seem, the efficiency of OpenMP drops very quickly, reaching values below 0.2 with around 50 threads. MPI, however, surprisingly kept the efficiency over 0.8 even with 512 processors, which means that, by using this measure on this specific problem, MPI is definitely more efficient than OpenMP.

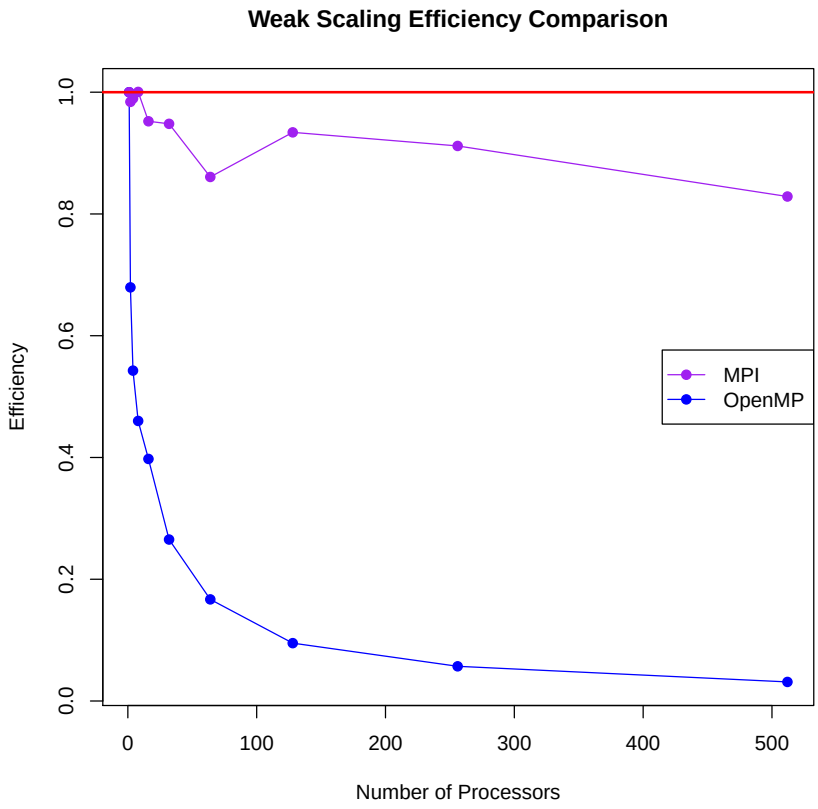

Figure 7. Efficiency Comparison

Finally, let us compare the speedup of both methods on Figure 8. The red diagonal line is the perfect speedup. While OpenMP showed an increase at the speedup, this increase was small, so the scaling was poor. MPI, on the other hand, showed almost perfect scalability until 256 processors, and gave incredible results even for 512 processors, which means that the method scales almost perfectly with MPI.

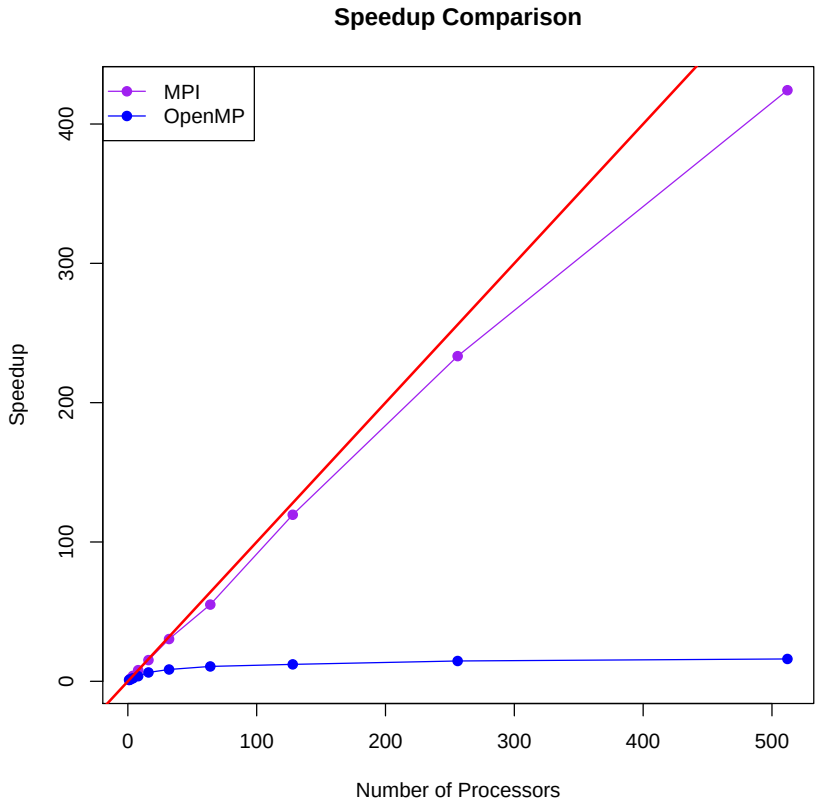

Figure 8. Speedup Comparison

The results were surprising. For the nature of the problem, we thought that the thread approach using OpenMP would be the most efficient way to deal with MCMC, however, since each chain is separated and they do not interact with each other, using MPI ended up being natural, since it is basically the same code running multiple times in each processor. Other thing that might have happened is that since every MPI processor has its own memory, then the data is stored locally and has easy access. OpenMP has private variables for every thread, but maybe it is not as optimized as MPI and there might be some time loss in this communication.

MPI seemed the best way to parallelize MCMC, due to the near perfect weak

scalability results that were found in our experiments. For more specific problems, Gibbs without dependence may be computed completely in parallel, which leads to even higher efficiency, but our method can be applied in most Bayesian models used nowadays, serving as a guide on how to make parallel MCMC efficiently.

References

Turkman, M. A. A., Paulino, C. D., and Müller, P. (2019). *Computational Bayesian Statistics*. Cambridge University Press.