# Parallel Single Chain in Markov Chain Monte Carlo Simulation

Wint Pa Pa Kyaw[*]

## Abstract

The Markov Chain Monte Carlo (MCMC) method is a statistical almost experimental approach to computing integral using random positions, called samples, whose distribution is carefully chosen. In this research, a normal distribution model with unknown mean and known variance is considered. Posterior statistics are computed using the sample mean and standard deviation, as well as the prior mean and standard deviation, instead of data input. Because this is a single-parameter model, posterior samples of mean are simulated in parallel by Monte Carlo simulation. This research also presents parallel communication schemes for simulated a single chain in Markov chain using Message Passing Interface (MPI). In this simulation, the number of simulation steps broadcast to all participating processes. Each process computes a partial sum of simulated values. All partial sums are combined into the grand sum. Finally, the root process computes posterior mean and standard variance. A major purpose of this research is to advocate the use of parallelization within a single chain, hence infusing high-performance computing technologies.

**Keywords:** parallel processing, Message Passing Interface, communication, Markov chain

## Introduction

Typically, implementation of a high-dimensional model based on Markov Chain Monte Carlo (MCMC) techniques is notoriously intensive in computing and often requires days, weeks, or even months of CPU (Central Processing Unit) time on personal computers and workstations. Therefore, in order to overcome such computational burden, parallel computing becomes appealing.

Parallel computing operates on the principle that a large problem can be split into smaller components and solved concurrently (i.e." in parallel"), each on a separate processor (or CPU core). An instance of a computer program and its activities that are taking place on each processor is referred to a process. Thus, parallel computing involves activating multiple processes that concurrently carry out related computing jobs and combining results by the main "controlling" process. Parallel computing can be achieved by programming with C, C++ e.g., using the MPI (Message Passing Interface) library to handle inter-process communication. High-performance computing communities have developed parallel programs for decades but were previously limited to programs running on expensive super-computers. In the past twenty years, interest in parallel computing has grown markedly due to physical constraints that prevent frequency scaling and to the need to handle datasets of unprecedented dimensionalities that are being generated. Parallel computing has now become a dominant paradigm in current computer architectures, mainly in the form of multi-core processors.

Parallel MCMC methods have recently been adopted in statistics and informatics and in image processing. MCMC algorithms are seemingly serial, and parallelism is not as straightforward as one would expect. Many intensive computational tasks in some applications have been handled via some simple data parallelism, implemented through the "multiple-tasking" mechanism. Multiple-tasking allows each processor to switch between tasks being executed on it, without having to wait for each task to finish, but this type of "parallel" computing is not scalable with the number of jobs. Recently, parallel MCMC algorithms and strategies have become a focal point for scientific computing. This is largely due to the need to handle datasets of unprecedented sizes. With datasets of unprecedented sizes in a model, the computing task is highly challenging, particularly with sophisticated models via MCMC

[*] Associate Professor, Department of Computer Studies, University of Yangon

implementation. This paper presents a technical description of parallel MCMC method. The algorithm typically deals with parallelization within a single chain.

## Parallel Simulation for a Single-Parameter Normal Model

Consider a normal distribution model with unknown μ and known $\sigma^2$. For a vector *y* of *n* identically independently distributed (iid) observations, the likelihood is:

$$P(y \mid \mu) = \prod_{i=1}^{n} \left(\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-1\mu)'(y-1\mu)}{2\sigma^2}\right)\right). \quad (1)$$

If a normal prior is assumed, that is, $P(\mu) \propto \exp\left(-\frac{1}{2\tau_0^2}(\mu-\mu_0)^2\right)$. (2)

where $\mu_0$ and $\tau_0^2$ are hyperparameters. It can be shown that the posterior density μ is also normal :

$$P(\mu \mid y) = N\left(\frac{\frac{1}{\tau_0^2}\mu_0 + \frac{n}{\sigma^2}\bar{y}}{\frac{1}{\tau_0^2} + \frac{n}{\sigma^2}}, \left(\frac{1}{\tau_0^2} + \frac{n}{\sigma^2}\right)^{-1}\right) \quad (3)$$

Intuitively, the posterior mean of $\theta$ is expressed as a weighted average of the prior mean ( $\mu_0$ ) and of the sample mean ( $\bar{y}$ ), with weights equal to the corresponding precisions, $\frac{1}{\tau_0^2}$ and $\frac{n}{\sigma^2}$, respectively. Because this is a single-parameter model, posterior samples of μ can be simulated in parallel by following the same algorithm as for Monte Carlo simulation.

## Parallel Monte Carlo Methods

In practice, many statistical problems involve integrating over hundreds or even thousands of dimensions but usually these problems are not analytically tractable. Instead, Monte Carlo simulation methods can be used to tackle high-dimensional integrals. Standard Monte Carlo integration algorithms distribute the evaluation points uniformly over the integration regions.

### Parallel Computing for Evaluating Integrals

To begin, consider the following integral

$$E(P(\theta)) = \int P(\theta)\, f(\theta)\, d\theta \quad (4)$$

for some high-dimensional θ with density $f(\theta)$. Suppose the integral cannot be evaluated analytically. If *n* realizations of θ can be sampled independently from $f(\theta)$ then, according to the strong law of large numbers, the sample average $\frac{1}{n}\sum_{i=1}^{n} p(\theta)^{(i)}$ provides an approximation to $E(p(\theta))$ when $n \to \infty$.

Simple Monte Carlo algorithms proceed by averaging large numbers of values that are generated independently of each other. Obviously, Monte Carlo simulation is parallel in computing because it can be conducted concurrently. By parallel computing, the entire set of samples can be divided among the available CPU cores and then each core generates a portion and summarizes its local samples. After all processors have finished their tasks, a master program summarizes all the partial data and outputs the final result.

Suppose that there are *K* CPU cores that generate a total of *T* samples, each handling an equal portion of these samples. For simplicity, assume that *T* is divisible by *K*, such that the quotient ($m = T/K$) is an integer. Then, parallel Monte Carlo simulation proceeds as follows:

➔ Process 0 (master process):

(a) computes and passes *m* to each process.

➔ Each (slave) process ( say *j* ):

(a) simulates *m* independent realizations of $\theta$;
(b) computes $S_j = \sum_{i=1}^{m} P(\theta^{(i)})$, and passes $S_j$ back to the master program.

➔ Process 0 (master program):

(a) sums $S_j$ and generates the final sum $S = \sum_{j=1}^{K} S_j$ ;

(b) computes the Monte Carlo estimate as $E\ (p\ (\theta)) = \frac{S}{T}$.

In this example, the master process does not involve computing the sum of a portion of the data but it actually can. Also, each process is given the same number of samples. This works well if all CPU cores process the data at the same speed or approximately so. In practice, however, clock frequencies (i.e., computing speed) can vary markedly among processors. Hence, it can be more effective for each processor (or CPU core) to process a different number of samples, roughly proportional to its computing speed, and then the master program compute the weighted average of all samples obtained from the *K* cores.

**Parallel Computing of Single-Parameter Models**

A single-parameter model can serve as a building block for modeling. Consider a normal distribution with known mean $\mu$ and unknown variance $\sigma^2$ to be inferred. The data density for a vector *y* of *n* identically independently distributed (iid) observations is:

$$P(y\,|\,\sigma^2) \propto (\sigma^2)^{-\frac{n}{2}} \exp\left(-\frac{n}{2\sigma^2}\ S^2\right) \qquad (5)$$

Where $S^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i-\mu)^2$ is the sufficient statistic. Assuming an inverse- $\chi^2$ prior distribution with scale $\sigma_0^2$ and $\upsilon_0$ degrees of freedom,

$$P(\sigma^2) \propto \left(\frac{1}{\sigma 2}\right)^{\frac{\upsilon_0}{2}+1} \exp\left(-\frac{\upsilon_0\,\sigma_0^2}{2\ \sigma 2}\right) \qquad (6)$$

it can be shown that the posterior density of $\sigma^2$ is a scaled inverse- $\chi^2$ distribution with scale $\frac{\upsilon_0\,\sigma_0^2+n\,S^2}{\upsilon_0+n}$ and $\upsilon_0 + n$ degrees of freedom :

$$\sigma^2\ |\, y \sim X^{-2}\ \left(\upsilon_0 + n,\ \frac{\upsilon_0\,\sigma_0^2+n\,S^2}{\upsilon_0+n}\right) \qquad (7)$$

Hence, the posterior mean of $\sigma^2$ is $\frac{\upsilon_0\,\sigma_0^2+n\,S^2}{\upsilon_0+n-2}$ for $\upsilon_0 + n > 2$. Numerically, the posterior distribution of $\sigma^2$ can be inferred based on posterior samples generated from (7). Computing for this single-parameter normal model can follow exactly the same algorithm as parallel Monte Carlo simulation. Briefly, *K* parallel processes are executed, each generating a portion of the posterior samples of $\sigma^2$. Then, the master process generates the final sum and computes the estimated posterior mean of $\sigma^2$ as a weighted average of all sample averages.

To show why the algorithm of parallel Markov chain simulation applies to parallel computing of a single parameter model, consider equation (4). For this single-parameter normal model, for example, the marginal posterior expectation of $\sigma^2$ can be expressed as:

$$E\,(\sigma^2|y) = \int \sigma^2 f \;\; (\sigma^2|y)\; d\;\; \sigma^2 \qquad (8)$$

Clearly, (8) implies a similar Monte Carlo implementation: if *n* samples of $\sigma^2$ are generated from its marginal posterior density $f(\sigma^2|y)$; then, as n =∞ ; $E(\sigma^2|y)$ can be approximated by the sample average:

$$\overline{\sigma^2} = \sum_{i=1}^{n} (\sigma^2)^{(t)} \rightarrow E\,(\sigma^2|y) \qquad (9)$$

**Parallel Markov Chain Monte Carlo Simulation**

Analytical solutions are not always available for most multiple-parameter models. Instead, MCMC simulation can be used to draw samples from the joint posterior distribution and then evaluate sampled values for the parameter(s) of interest while ignoring the values of other unknowns. MCMC methods are a variant of Monte Carlo schemes in which a Markov chain $\{X_j$ , j = 1, 2} is constructed with equilibrium distribution π equal to some distribution of interest, such as a posterior distribution in a analysis. Typically, the initial value is not a draw from the distribution π but if the chain is constructed properly, then $X_t \overset{d}{\rightarrow} \pi$ (here, *d* means convergence in distribution) and, under certain conditions, an estimator $\hat{h}$ converges to $h_\pi$ as t=∞. However, a Markov chain is sequential by nature because the distribution of $X_{t+1}$ depends on the value of $X_t$ ; where t indexes the order of MCMC iterations. This introduces a difficulty to parallelization of a Markov chain.

**Parallelization within a Single Chain**

By running multiple Markov chains, one often observe that samplers mix poorly and each chain may require a very long burn-in time. Hence, it would be preferable to develop parallelism within a single chain, instead of running multiple chains. Markov chain simulation is an iterative procedure, in the sense that simulation of the next value of the chain depends on the current value. This creates difficulty for delivering parallelism for a single Markov chain. Nevertheless, one will show that a single chain can be parallelized, subject to assumptions of conditional independence in the model. The key is to identify such steps that can be implemented in parallel.

After all parameters are given initial values, the parallel MCMC algorithm proceeds by repeatedly conducting the following steps:

➔ Master program:

(a) samples a new σ, given realization of *θ* and the data *y*, and

(b) distributes the new σ to each process.

➔ Each process (*k*):

(a) updates a subset of $\theta s$ that have been assigned to it, conditional on σ and *y*,

(b) computes summary statistics for the updated $\theta s$; and

(c) passes the summary statistics back to the master program.

Often, the above algorithm works quite well when the $\theta$ are all independent of one another, given σ and y. In practice, however, such independence may not necessarily hold and strategies must be developed to deliver efficient parallel MCMC algorithms given specific dependence between elements.

## Implementation of Communication Protocol

The purpose of this example is to show parallel computing using the MPI (Message Passing Interface) library. The change in computing time for this example is, however, almost insignificant because sampling from a normal distribution is very quick. In addition, with parallel simulation, inter-process communication requires some extra time as overhead, which offset gains from parallel computing.

MPI is a language-independent communication protocol used to program parallel computers that is extensively used for high-performance computing. More specifically, MPI is a library of routines for creating parallel programs e.g., in C, that allow users to create programs that can run on most parallel computer architectures. In the code, the MPI library is used to handle inter-process communications in the C program. With MPI, each task can have its own local memory during computation (but multiple tasks can reside in the same physical machine and/or an arbitrary number of machines). Typically, tasks exchange data by sending and receiving messages but data transfer usually requires cooperation among processors, that is, a "send" operation must have a matching "receive" operation.

MPI_Comm_rank() is used to find out the ID of all participating processors and MPI_Comm_size() is used to get the number of participating processors. MPI_Bcast() is used to send common parameter values (e.g., number of simulation steps) to all participating processors. Then, after each processor has finished its work, the subroutine MPI_Reduction() is used to sum up the posterior values from all processors. Subroutine MPI_Reduction() collects data from all processors, reduces the data to a single value (e.g., by summation), and then stores the results on the master process (and on all processes as well). There are several predefined operations that MPI_Reduction() can provide. In addition to summation, it can also conduct multiplication, and find minimum or maximum values. Finally, the master processor computes the means and standard deviation (and other posterior statistics, when relevant) for the mean of the normal model. Sequential functions are used to generate random numbers, with process ID used as the random number seed.

## Implementation of Parallel Simulation Program for a Single Chain

The process computes posterior statistics with the following code segments.

```
tar0 = 1.0/(sd0*sd0);
tar1 = (1.0*nind)/(sd1*sd1);
varn = 1.0/(tar0 + tar1);
sdn = sqrt(varn);
mun = varn * (tar0*mu0 + tar1*mu1);
```

The process allocates memory for random seed variable with the following code segment.

```
MPI_Alloc_mem(sizeof(long), MPI_INFO_NULL, &idum);
```

The process broadcasts the number of simulation steps to all participating processes with the following code segment.

```
ierr = MPI_Bcast(&niters, 1, MPI_INT, root_process, MPI_COMM_WORLD);
```

Each process computes a partial sum of simulated values with the following code segment.

```
for (i = proc_id + 1; i < niters + 1; i +=
nprocs)
{
  xi = mun + sdn * randomnormal(idum);
  xi2 = xi * xi;
  psum = psum + xi;
  psum_xi2 = psum_xi2 + xi2;
}
```

```
ierr = MPI_Reduce(&psum, &sum, 1, MPI_DOUBLE, MPI_SUM, root_process, MPI_COMM_WORLD);
ierr = MPI_Reduce(&psum_xi2, &sum_xi2, 1, MPI_DOUBLE, MPI_SUM, root_process,
          MPI_COMM_WORLD);
```

The process does a reduction in which all partial sums are combined into the grand sum with the following code segment.

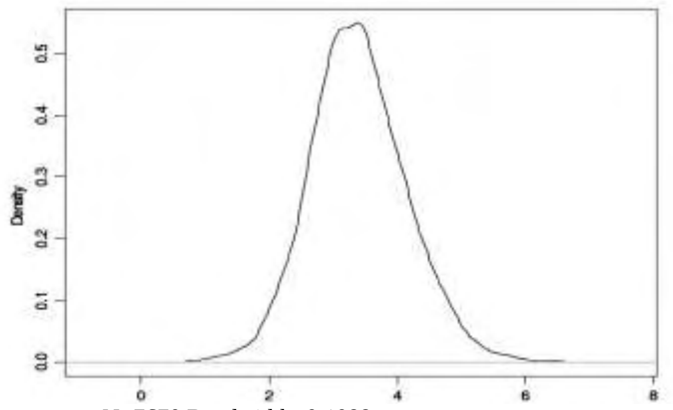
Finally, the root process prints posterior mean and standard error of mu with the following code segment.

```
if(proc_id == root_process)
{
mumu = sum / niters;
sdmu = sqrt((sum_xi2 - niters*mumu*mumu)/(niters-1));
 }
```

## Results and Discussion

The example data are physical quantities measured on 7670 samples. The kernel density of average quantity is shown in Figure 1, which approximately suggests a normal distribution. Assume that the population variance of average quantity is 0.58. In this example, the prior distribution is assumed to be normal with mean equal to 4.0 and variance equal to 1.0 (these are just guesses of the parameter values in the distribution of average quantity). A parallel C program is used in this analysis. To estimate μ; a total of 1 000 000 values is simulated for μ; which are handled by K = 10 processes, each generating 100 000 values and computing the partial sum. Then, the K partial sums are transferred back to process 0, where the Monte Carlo estimate is computed. The program only outputs the posterior mean and the standard deviation. The program also outputs minimum and maximum values.

The posterior mean is estimated to be 3.394, which corresponded very well to the sample mean of 7670 samples (Table 1) because the impact of the prior on the posterior could be ignored given the very large sample size. The median and mean agreed well with each other (Table 1). These are indications that the posterior distribution of the mean of samples is symmetric.



N=7670 Bandwidth=0.1099

Figure 1 Kernel density of physical quantity measured in 7670.

**Table 1 Posterior summary statistics of physical quantity based on a single-parameter normal model.**

| Sample set | Min | Median | Mean | Max |
|---|---|---|---|---|
| 1 | 3.357 | 3.394 | 3.394 | 3.431 |
| 2 | 3.356 | 3.394 | 3.394 | 3.429 |
| 3 | 3.352 | 3.394 | 3.394 | 3.430 |
| 4 | 3.357 | 3.394 | 3.394 | 3.436 |
| 5 | 3.353 | 3.394 | 3.394 | 3.432 |
| 6 | 3.355 | 3.394 | 3.394 | 3.430 |
| 7 | 3.355 | 3.394 | 3.394 | 3.431 |
| 8 | 3.354 | 3.394 | 3.394 | 3.428 |
| 9 | 3.356 | 3.394 | 3.394 | 3.431 |
| 10 | 3.358 | 3.394 | 3.394 | 3.433 |
| Pooled | 3.352 | 3.394 | 3.394 | 3.436 |

Min = minimum value; max = maximum value.

The above results were obtained from parallel computing on ten CPU cores. The run times in seconds using simulation steps (s = 1 000 000) with the various number of processors in MPI version. Speedups of various processors are listed in Table 2.

**Table 2 The Speedup obtained after four consecutive trial runs with simulation steps** (s=1 000 000).

| No. of Processors | Speedup |
|---|---|
| **8** | 7.33 |
| **10** | 10.04 |
| **16** | 16.10 |
| **24** | 23.11 |

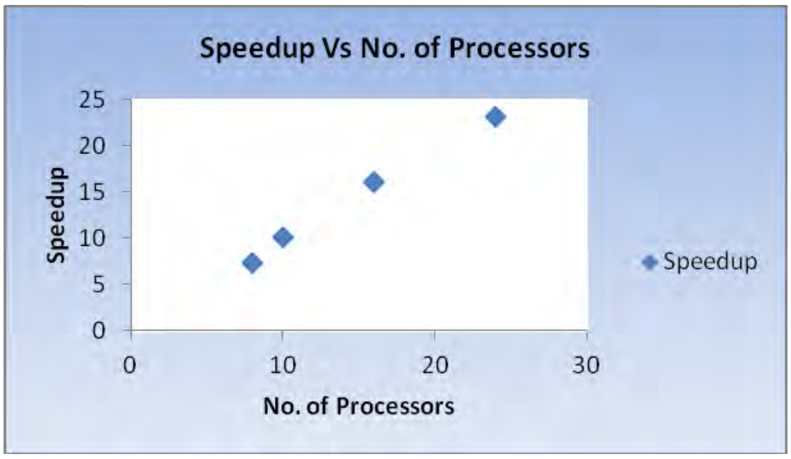


Figure 2: The relation of Speedup and Number of processors

When the program is executed with multiprocessor, the execution time for serial processing and the execution time for parallel processing are obtained for different processor. Then speedup are calculated. From Table 2 and Figure.2, it can be observed that parallel computing is more suitable for enormous data. The experimental results show that parallel algorithms using MPI achieve comparable accuracy and almost linear speedups over the traditional serial version. The computational time of using the MCMC method on a data set is shown in Table 3.

**Table 3 The results obtained after four consecutive trial runs with S = 1 000 000 000.**

| Number of Processors | Execution Time (seconds) |
|---|---|
| **4** | 10.502036 |
| **8** | 6.714650 |
| **16** | 5.794617 |
| **24** | 5.624466 |

When the program is executed with multiprocessor, the execution time for parallel processing are obtained for different processor. The results in Figure.3 show that parallel execution time goes faster with increasing the number of processors.

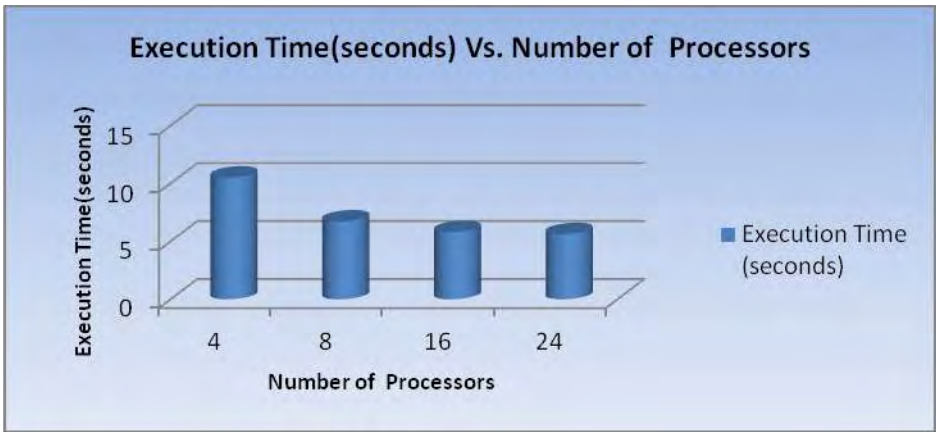


Figure 3: The relation of No. of Processors and Execution Time

## Conclusion

In this regard, high-performance computing offers a markedly competitive edge, not only in reducing computing time but also in tuning optimal models for prediction. A single chain MCMC algorithm tackles a large range of complex inferential problems that were previously not considered possible, tractable. In the meantime, statisticians are becoming ever more ambitious in the range (complexity) of models they consider and the algorithms for large complex models often require enormous amounts of computing power.

## Acknowledgements

I would like to express my sincere grateful very much to Associate Professor Dr Soe Mya Mya Aye, Head of Department of Computer Studies, University of Yangon, for her kind permission to carry out this research. I am greatly indebted to Dr Pho Kaung, Rector, and University of Yangon for his excellent help and creative ideas that have assisted me in broadening my research skills and his continuous guidance, devotion and perpetual encouragement.

## References

Edelman,A., (2004), *Applied Parallel Computing*, Massachusetts Institute of Technology press.

Grama,A., Gupta,A., Karypis,G. and Kumar,V., (2003), *Introduction to Parallel Computing*, 2nd Edition, Pearson Education, The Benjamin/Cummings, ISBN: 7-111-12512-6.

Karniadakis,G.E. and Kirby,R.M. , (2007), *Parallel Scientific Computing in C++ and MPI*, Cambridge University Press, London, ISBN:9780521520805.

Sasikumar,M., Shikhare,D. and Prakash,P.,R.,(2000), *Introduction to Parallel Processing*, Prentice-Hall,New Delhi, India, ISBN-81-203-1619-3.