1. Introduction

1.1 Parallel Computing

Parallel computing is a computational system that breaks down large, complex problems into smaller calculations to make them run simultaneously with the help of multiple processors. By leveraging multi-core processing, parallelisation can increase computing power for problem solving and improve performance. A parallel computing system (with multiple processors installed in a server rack) is generally contained in a single datacentre, where tasks are further distributed in parts and executed by each server at the same time.

Figure 1 (Parallel Computing Definition, n.d.) below provides a better illustration of how parallel computing works. From the image, note that the original problem is broken into discrete parts that can be solved synchronously. Each problem is further divided into a set of instructions, which will be executed concurrently on different processors. Consequently, each subset of the problem can be solved simultaneously rather than sequentially.

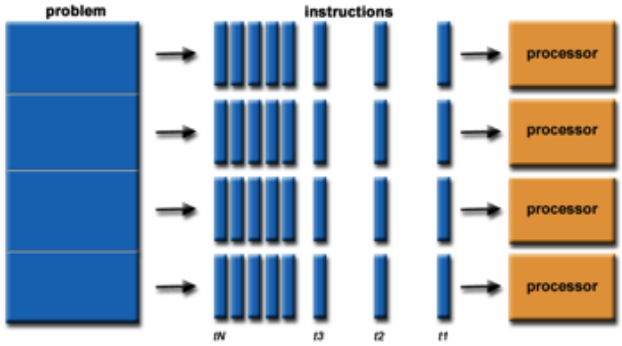


Figure 1: The decomposition of a problem through parallel computing.

Here is a quick overview of the four main kinds of parallelisation that are commonly used:

1. *Bit-level Parallelism:* This mainly focuses on increasing the word size of the processor in order to reduce the number of instructions executed to run a task. For example, a 16-bit processor will execute an operation faster as compared to an 8-bit processor.

2. *Instruction-level parallelism:* This kind of parallelisation uses 'pipelining' to overlap the execution of all independant instructions in parallel. The most common way to increase parallelisation in ILP is by exploiting each iteration in a loop, also known as 'loop-level parallelism'.

3. *Task Parallelism:* Task parallelisation breaks down a problem into smaller subproblems for simultaneous implementation of all parts of the code. For instance, a task might be divided into two threads and each thread performs a specific statistical operation.
4. *Superword-level parallelism:* Superword-level parallelism is an essential alternative for loop level parallelisation that uses a vectorization technique with the help of a vector processor to take advantage of inline code parallelism.

1.2 Relevance and Future of Parallel Computing

The emergence of parallel computing is attributed to the need for solving large problems. Unlike traditional serial computing, which solves each task in order, parallel computing provides an exclusive method to break down the problem into smaller subproblems and solve them simultaneously using different computer cores. This has the benefit of lowering the execution time and cost, especially when the problem at hand is computationally intensive.

Due to the decrease in the price of computer components such as processors, parallel computing has become more popular in practical applications, and its importance continues to grow with the increasing usage of multicore processors and Graphical Processing Units (GPUs). The growing presence of GPUs in multiprocessing systems can be attributed to their larger number of cores that are smaller and more efficient, making them better equipped to handle complex parallel architectures than traditional CPUs. As parallel computing is becoming progressively faster, problems that had previously taken too long to run are now able to be executed more feasibly. As a result, a wide range of fields that rely on solving computationally expensive problems have taken advantage of parallel computing, ranging from bioinformatics to economics.

1.3 Multiprocessing in Python

Multiprocessing is a Python module (multiprocessing — Process-based parallelism, n.d) that uses an intuitive, yet simple API to divide a task into multiple processes. The use of this API in the package helps to bypass the technical difficulties of a Global Interpreter Lock (GIL), which was initially designed to handle the memory management issue in multiple threading but limited Python to the use of a single processor. Multiprocessing allows for a block of code to be sent to multiple processors for concurrent execution.

The *multiprocessing* module is the most widely used package for the implementation of parallel computing in Python. As such, it will be our primary tool to parallelise the Metropolis-Hastings algorithm.

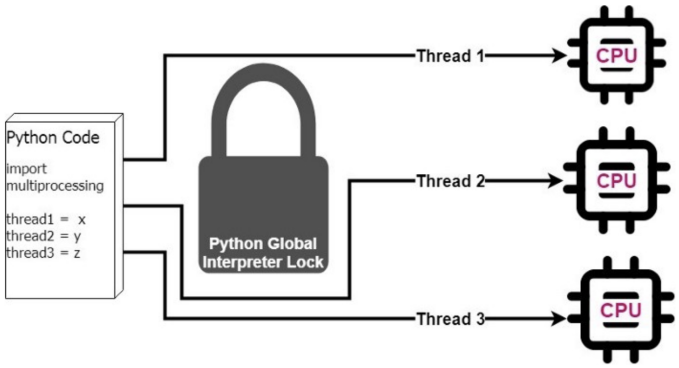


Figure 2. Implementation of multiprocessing in Python.

2. Markov chain Monte Carlo and the Metropolis-Hastings Algorithm

2.1 Markov chain Monte Carlo (MCMC)

Before reaching the Metropolis-Hastings algorithm, we will first briefly explore the rapidly developing topic of Markov chain Monte Carlo methods, also known as MCMC.

MCMC refers to a collection of algorithms that are used to obtain random samples from highly complex, potentially multivariate distributions that would otherwise be difficult to sample from directly. In most cases, these distributions will not even be known to us completely, being specified only up to proportionality. MCMC methods have recently come into prominence due to the advancements in computing power which have made MCMC more convenient to implement in practice.

The exact method by which a random sample is obtained depends on the variant of MCMC that is used. However, what all MCMC methods have in common is that they create a random sample in the form of a *Markov chain.* A Markov chain is a sequence of events, known as *states,* whereby the probability of the outcome of the $(t+1)^{th}$ state depends only on the t^{th} state and nothing else prior to it. Formally, a sequence of possible events $\{X_t\}$ is called a Markov chain if it exhibits the *Markov Property:*

$$P(X_{t+1}|X_t, X_{t-1}, \ldots, X_1) = P(X_{t+1}|X_t)$$

The reason for generating a Markov chain is that under certain conditions (which are outside the scope of this paper), a Markov chain will converge to a probability vector known as a *stationary distribution*, and in the case of MCMC, the stationary distribution turns out to be the target distribution from which we want to sample (Robert & Casella, 2004). Therefore, the states of the Markov chain can be used as samples from the complex distribution of interest.

It is important to note that the generation of samples from the stationary distribution relies on convergence. This means that the early states of the Markov chain, before it has converged to the target, should not be viewed as samples from the distribution of interest. These first few states, collectively referred to as the *burn-in*, should be discarded. Note that the burn-in is set subjectively but is usually no more than 20% of the entire chain. We will discuss burn-in in greater detail below.

The random samples from the complex distributions provided by MCMC algorithms allow practitioners to use Monte Carlo methods (i.e. Monte Carlo integration, sequential sampling, etc.) to provide numerical solutions to problems in order to obtain and derive characteristics from these distributions, such as the density functions and sample moments. Prior to the emergence of MCMC, such information on these distributions was not available, so the application of MCMC also provided the foundations for further progression in a wide-spanning assortment of other scientific fields.

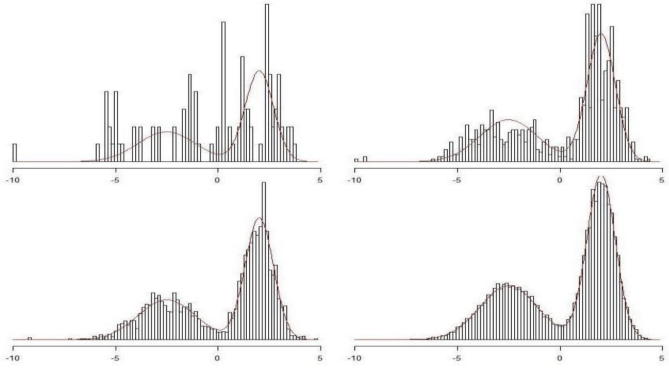


Figure 3. Graphical representation of the convergence of the relative frequencies of MCMC samples to the target distribution.

2.2 Metropolis-Hastings (MH) Algorithm

There are many different types of MCMC algorithms, but the most popular, and the method which forms the basis of this report, is the Metropolis-Hastings (MH) algorithm. The algorithm draws a *proposal* from a *proposal distribution* and then applies an acceptance/rejection criterion to decide whether the proposal should become the next state of the Markov chain or not. If the proposal is rejected, the current state of the chain simply becomes the next state.

Suppose f denotes the function proportional to the density of the distribution of interest. We now present a formal step-by-step method for implementation of the Metropolis-Hastings algorithm:

1. Initialize x_0 at $t = 0$
2. Repeat for $t = 1$ to $t = n$ (for sufficiently large n):
   a. Sample a proposal state x^* from the proposal distribution $q(x^*|x_{t-1})$
   b. Calculate the acceptance probability, $\alpha = \min(1, \frac{f(x^*)q(x_{t-1}|x^*)}{f(x_{t-1})q(x^*|x_{t-1})})$
   c. Set $x_t = x^*$ (acceptance) with probability α; otherwise, set $x_t = x_{t-1}$ (rejection)

The resulting Markov chain has converged to the target distribution, with its states providing a sample — after removing the burn-in.

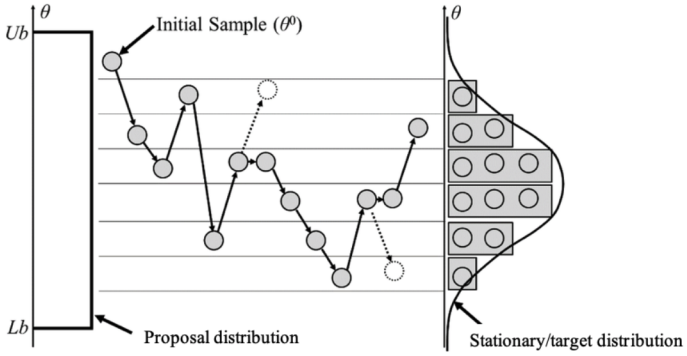


Figure 4. Trace plot and histogram of the sample generated by the MH algorithm using a uniform proposal distribution to obtain samples from a Normal distribution. The rejected proposals can also be seen on the trace plot.

2.3 Bayesian Inference and Example of Empirical Appliation (Non-Parallel)

The field of Bayesian inference has vastly benefitted from the power of MCMC. The Bayesian framework offers an alternative school of thought to the frequentist approach of statistics by incorporating beliefs held beforehand about an event, known

as *prior information*, into the probability density functions. Bayesians also consider the parameters of distributions as random variables.

For example, suppose that it was believed that the heights X of a town's residents are normally distributed with mean μ and variance σ^2, i.e. $X \sim N(\mu, \sigma^2)$. Frequentists would think of μ and σ^2 as unknown, but constant values. Bayesians instead hold that they are not fixed, but instead have their own probability distributions.

Given some data, a typical hypothesis test for both frequentists and Bayesians would involve point or interval estimation of the unknown parameters. The distinction is that frequentists would produce a *confidence interval* using the asymptotic distributions of functions of the estimators (e.g. $\bar{X} \sim N(\mu, \frac{\sigma^2}{n})$). However, Bayesians would derive *credible intervals* directly from the distributions of these unknown parameters, known as *posterior distributions*. These posterior distributions are derived through a direct application of Bayes' Theorem. For a given sample X,

$$\pi(\mu, \sigma^2|X) = \frac{f(X|\mu, \sigma^2)p(\mu, \sigma^2)}{\iint f(X|\mu, \sigma^2)p(\mu, \sigma^2)d\mu\, d\sigma^2} \propto f(X|\mu, \sigma^2)p(\mu, \sigma^2),$$

where

- f is the *likelihood* of observing X, given μ and σ^2 (assumed known here)
- p is the *joint prior density* of the parameters μ and σ^2, i.e. what we previously assumed the joint density to be (also assumed known)
- π is the *joint posterior density* of the parameters μ and σ^2, after observing X (unknown and estimated here)

Of course, Bayesians can characterise the posterior distribution of interest through the use of the MH algorithm, drawing samples from the distribution (as it is known up to proportionality). This allows practitioners to easily produce their credible intervals or otherwise use the samples for any other suitable purpose.

2.4 Implementation of MH Algorithm in Python (Non-Parallel)

Below is our implementation of Metropolis-Hastings without any parallelisation. Our goal is to generate samples from the posterior distributions of μ and σ given data that is $N(\mu, \sigma^2)$. The algorithm follows almost exactly from the steps outlined in Section 2.2. Here is an overview of this non-parallel implementation:

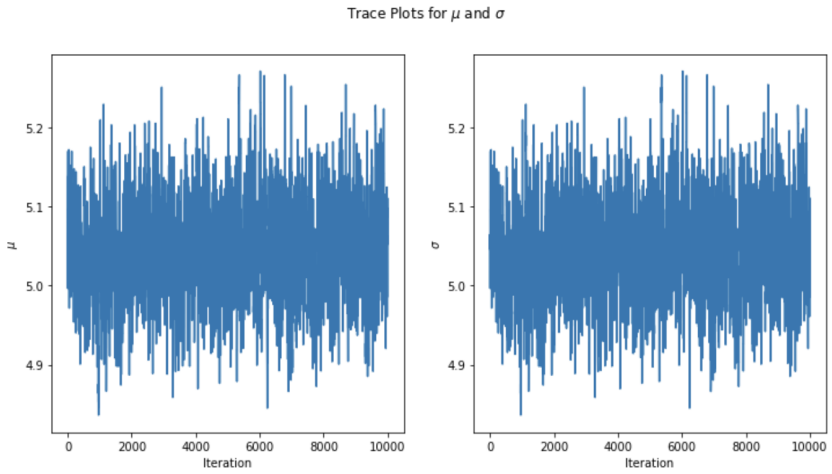
- We first create a *log_posterior* function that allows the user to flexibly define the natural logarithm of the function that is proportional to $\pi(\mu, \sigma^2|X)$. The parametrisation of the prior distributions can be changed; for the example below, as there is no prior information, we use high-variance priors of $N(0, 100)$ for μ and $IGamma(0.01, 0.01)$ for σ.
- We take logs of the distributions because taking sums is computationally easier to work with than the original product.
- Our MH sampler takes in a desired number of samples, the data, initialisations of μ and σ (we use the observed values in the data), and a proposal standard deviation array when sampling from the proposal distribution. We use a normal proposal distribution with a standard deviation of 0.05 for both μ and σ in this example.
- Because our proposal distribution is symmetric, our implementation of MH results in the variant known as *random-walk Metropolis*, and our acceptance probability simplifies to the following:

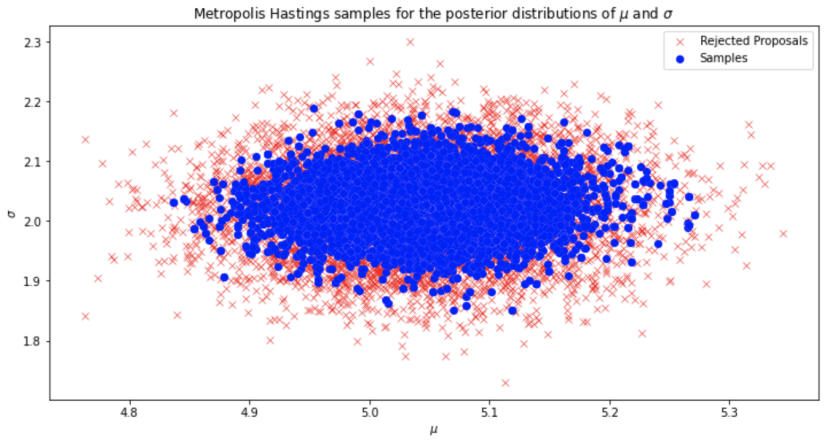
$$\alpha = \min(1, \frac{f(x^*)}{f(x_{t-1})})$$

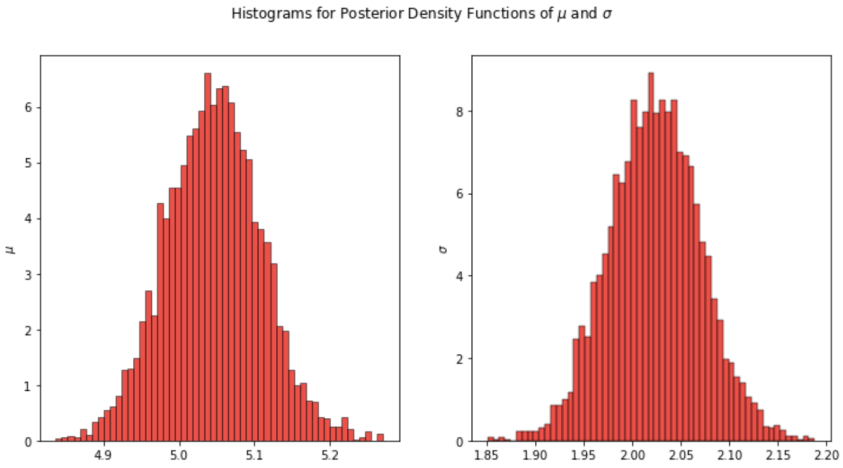
- The actual data is generated from a normal distribution with $\mu = 5$ and $\sigma = 2$, and the burn-in is set here to be zero (for illustrative purposes). Below is a summary of the results, as well as some visualisations of the output. The algorithm generates samples of μ and σ of size 10,000 concentrated around the population parameters 5 and 2. The trace plots oscillate around those values as well. From the accept/reject plot, we can get a good intuition for which points are chosen to be a part of the chain, and the resulting histograms suggest that this acceptance/rejection criterion works very well for this example.

In [ ]:
```
import numpy as np
import scipy.stats as st

def log_posterior(data, mu, sigma):

    #Assumed priors are mu~N(0,10) and sigma~IGamma(0.01, 0.01)
     log_prior = np.log(st.norm(0, 10).pdf(mu)) +  \
                 np.log(st.invgamma(a=0.01, scale=1/0.01).pdf(sigma))
    #Know (or assume if not known) that the data is normally distributed
     log_lik = np.sum(np.log(st.norm(mu, sigma).pdf(data)))

    #Log-posterior density is the sum of the log-prior and log-likelihood
```

```
    log_post_density = log_prior + log_lik
    return(log_post_density)

def MH_sampler(samples, data, mu_init, sigma_init, proposal_sd=[0.05,0.05]):
   #Initialisation
   acceptances = 0
    mu_current, sigma_current = mu_init, sigma_init
    chain = [[mu_current], [sigma_current]] # First values in the sample are the initial values
    rejected = [[np.nan], [np.nan]]

    for _ in range(1, samples):
        #Draw Proposals
        mu_proposal, sigma_proposal = np.random.normal([mu_current, sigma_current], proposal_sd, (2,))

       #Calculate Acceptance Probability
        proposal_log_posterior = log_posterior(data, mu_proposal, sigma_proposal)
        current_log_posterior = log_posterior(data, mu_current, sigma_current)
        p_accept = np.min([1, np.exp(proposal_log_posterior - current_log_posterior)])

       #Accept/Reject Step
        if np.random.uniform(0, 1) < p_accept:
            acceptances += 1
            rejected[0].append(np.nan), rejected[1].append(np.nan)
            mu_current, sigma_current = mu_proposal, sigma_proposal
        else:
            rejected[0].append(mu_proposal), rejected[1].append(sigma_proposal)

        chain[0].append(mu_current), chain[1].append(sigma_current)

        return (chain, rejected, acceptances)

data = st.norm(5, 2).rvs(1000)
mu_obs = np.mean(data)
sigma_obs = np.std(data)
samples = 10000
burnin = 0

np.random.seed(444)
results = MH_sampler(samples, data, mu_obs, sigma_obs)
print("{} samples".format(samples))
print("Observed mu = {:.3f}, observed sigma = {:.3f}".format(mu_obs, sigma_obs))
print("Burn-in = {}".format(burnin))
print("Acceptance Rate = {:.2%}".format(results[2]/samples))
print("Posterior averages: mu = {:.3f}, sigma = {:.3f}".format(np.mean(results[0][0][burnin:]),
                                                               np.mean(results[0][1][burnin:])))
```







3. The Parallelisation of the Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm is primarily a sequential model since it follows the Markov property of processing only one state at a time in each step. Although this algorithm allows us to evaluate arbitrary probability distributions, the Markov property can limit computational speed, as it is an inherently serial approach. Therefore, various parallel methods have been designed to attempt to solve this issue. In this report, our main focus is the method of parallel Markov chains, though we will also discuss its merits and drawbacks by comparing it with other parallel methods.

3.1 Parallel Markov Chains

The parallel Markov chains method simulates subsamples from multiple independent chains and then combines each partition in a subtle way such as using weighted averaging (Rosenthal, 2000). The algorithm is as follows:

1. Initialise the number of cores C and total iterations/samples N
2. For $c = 1, \ldots, C$ independently generate C Markov chains, each of size n_c, that have converged to the stationary distribution of interest. Note that $N = \sum_{c=1}^{C} n_c$.
3. After removing the chosen burn-in from each chain, combine all C subsamples to produce one unified sample from the target distribution.

The basic idea behind this algorithm is the partitioning of a long chain into several independent smaller chains and running each in parallel, which corresponds to Task Parallelism (as outlined above). The results should look similar to that of the original non-parallel algorithm with the same asymptotic bias. In addition, we expect the execution time for parallel Markov chains to be approximately C times less *in the best case*, again where C is the number of cores. This requires the total iterations to be divided evenly among all cores, i.e. $n_1 = n_2 = \ \ldots \ = n_c$. However, in practical scenarios, it is difficult to achieve such ideal performance, which will be further discussed below.

This parallelisation approach has been coined as *'embarrassingly parallel'* due to the short communication time between the

different processes. This is because, aside from the initial division of the task, the only communication required among the cores is at the final step where the subsamples are combined (Neiswanger, Wang and Xing, 2013). In addition, it is worth noting that although we use this approach to parallelise the MH algorithm, it is an extremely versatile and generalisable parallel method that can be extended to other MCMC algorithms such as the Gibbs sampler. The only step that needs to be altered is how we sample for each state of the Markov chain (Step 2 above).

3.2 Merits and Drawbacks

Due to the inherent difficulty in parallelising such sequential algorithms, there are only a limited number of previous practices. More concretely, some proposed methods are problem-driven, which cannot be extended to general cases (Vanderwerken and Schmidler, 2013). An example of this is in Bayesian phylogenetic inference, where parallel MCMC is used to evaluate computationally expensive likelihood functions (Feng et al., 2003). This makes the parallel Markov chain method an attractive choice for us.

Clearly, the main advantage of the parallel Markov chains method is that it has a minimal communication cost. Communication time is a big issue for many parallel computing algorithms, which will largely slow down the runtime (Neiswanger, Wang, and Xing, 2013). Since this method does not require any between-core communication during the simulation, and since the final combination of the chains into one sample is straightforward, there is little time wasted on communication. This benefit is particularly emphasised when comparing the parallel Markov chains method to some other popular algorithms of parallelising a single chain, such as prefetching and within-draw parallel algorithms, both of which require substantially more communication between processors (Strid, 2009).

On the other hand, the parallel Markov chains method will be less effective in improving runtime for distributions with long burn-ins. This is because the subdivision into smaller chains does not change the nature of the original chain; each smaller chain still requires the same amount of burn-in to constitute a reliable sample. Therefore, with a very large burn-in, the parallel Markov chains approach will not outperform the non-parallel case. Rosenthal (2000) also illustrated this issue and mentions that if the sample size is large enough relative to the burn-in, this drawback is inconsequential. However, if the number of cores C is large, the same burn-in applied to every single subsample could pose a significant problem. This is why parallelising one single chain is increasingly becoming an alternative focus instead (Strid, 2009).

The issue of long burn-in time for parallel Markov chains has been addressed in much of the existing literature. Rosenthal (2000) proposed both examining the convergence diagnostics to determine a burn-in time and using a fixed burn-in proportional to the sample size, but cautioned that both of these methods may introduce bias in their burn-in estimates. More recently, Vanderwerken and Schmidler (2013) introduced an advanced parallel Markov chains method, which can handle the existing issues of convergence. Neiswanger, Wang and Xing (2013) also considered the problems of long burn-in and communication cost together and developed a method that deals with both issues in conjunction. The aforementioned prefetching algorithm can also provide solutions to the burn-in issue (when not implemented in a parallel Markov chains context). However, when the burn-in is not so large as to make computation prohibitive, we prefer the parallel Markov chains method.

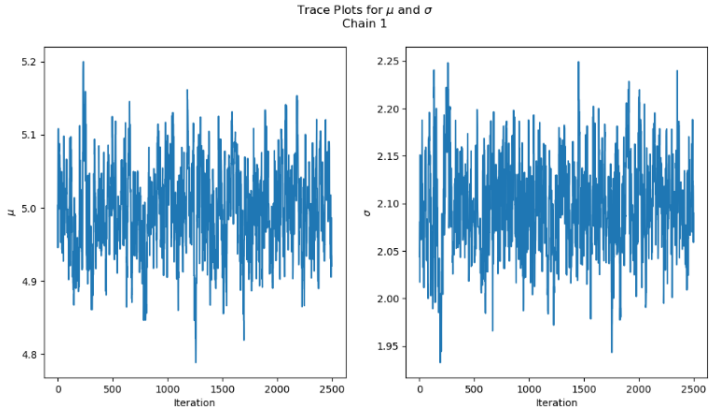
3.3 Implementation of MH Algorithm in Python (Parallel)

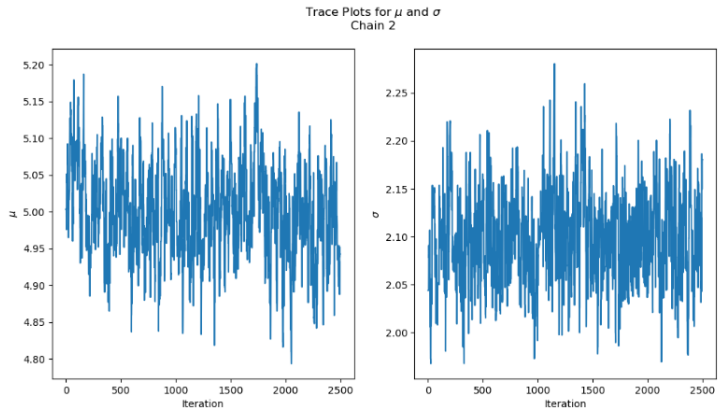
The parallel implementation utilizes the same functions defined above for the non-parallel case. First, we specify the number of cores (i.e. number of Markov chains) to use, which we choose to be 4 here. Then we split the data evenly among each chain. The Pool function from the *multiprocessing* module allows us to map the previously defined *MH_sampler* function to all 4 cores, generating 4 subsamples of size 2,500. Finally, we append the output from each chain together. **As a note, this snippet of code is designed to be illustrative of the process. To reproduce our results, please refer to the full code presented in the Appendix, which has also been attached as a separate Python .py script with the submission of this report.**

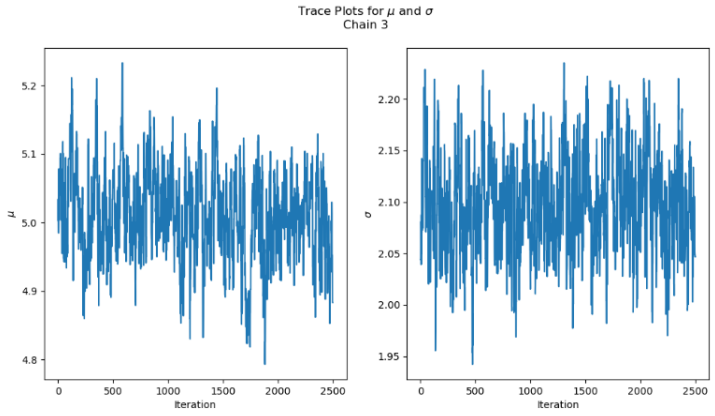
We provide the same visualisation of the output in the parallel case below. Instead of one set of trace plots, we have a set for each of the 4 chains. The trace plots, acceptance/rejection plot, and posterior histograms all look comparable to the non-parallel implementation. In the next section, we will present a comparison of runtimes to motivate the idea that our parallel Markov chains method provides a significant improvement in runtime. The example presented here and in section 2.4 is included in the table of results below.

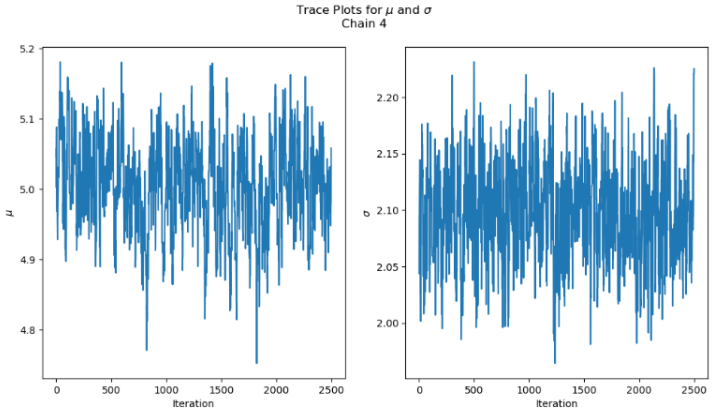
In [ ]:
```
from multiprocessing import Pool
from functools import partial

samples = 10000
nchains = 4
```

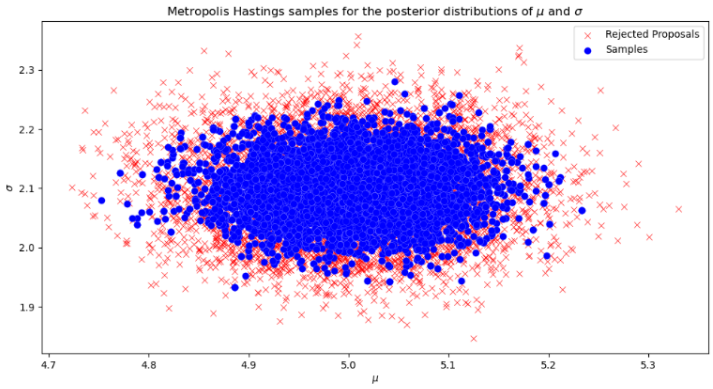
```
samples_per_chain = int(samples / nchains)
p = Pool(processes=4)
target = partial(MH_sampler, samples_per_chain, data, mu_obs, sigma_obs)
results = p.map(target, [[0.05,0.05] for _ in range(nchains)])
chain = []
burnin = 0
for i in range(nchains):
    chain += results[i][0]
    rejected = results[i][1]
    acceptances = results[i][2]
```

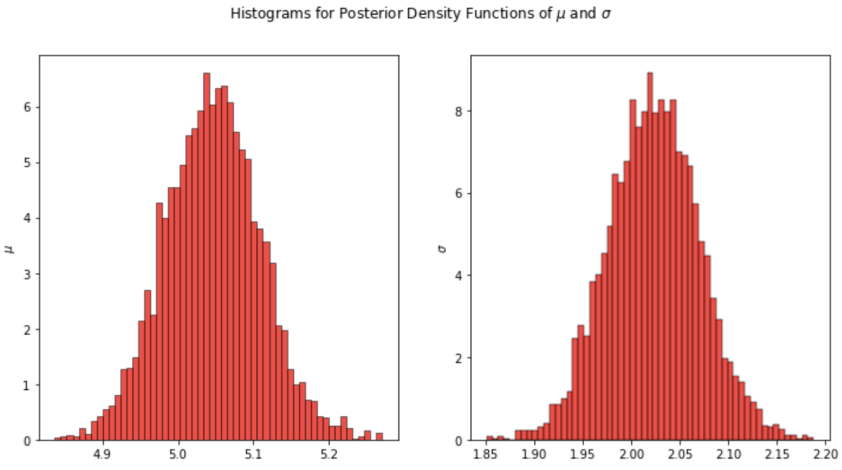







Trace Plots for μ and σ
Chain 4
μ
σ
Iteration
Iteration


Metropolis Hastings samples for the posterior distributions of μ and σ
Rejected Proposals
Samples
σ
μ


Histograms for Posterior Density Functions of μ and σ
μ
σ

4. Analysis of Parallelisation

4.1 Execution Time for Selected Runs of MH Sampler - Parallel vs. Non-Parallel

Final Sample Size	Burn-In	Samples Needed (Non-Parallel)	Samples Needed (Parallel)	Runtime (Non-Parallel)	Runtime (Parallel)
100	0	100	100	0.44 seconds	1.4 seconds
1,000	0	1,000	1,000	3.8 seconds	2.5 seconds
1,000	200	1,200	1,800	4.8 seconds	3.2 seconds
10,000	0	10,000	10,000	39.2 seconds	13.0 seconds
10,000	2,000	12,000	18,000	46.4 seconds	21.1 seconds
100,000	0	100,000	100,000	348.1 seconds	112.9 seconds
100,000	20,000	120,000	180,000	458.3 seconds	207.5 seconds

4.2 Benefits of Parallelising the Metropolis-Hastings Sampler

In this section, we will explore some of the issues discussed in section 3.2 in further detail and see if the results reflect our expectations.

We can see from the results that parallelising the MH algorithm can indeed reduce the time taken to obtain a sample from a complex distribution drastically. Our main comparison showed us that for large sample sizes and a burn-in period of 0 samples, the time taken to run the parallel algorithm is not too far away from C times faster (where $C = 4$ cores in this implementation). The non-parallel code took **39.2 seconds** to generate a sample of size 10,000, compared to the parallel code runtime of just **13.0 seconds**. This is of course extremely beneficial to practitioners who wish to employ the MH sampler to obtain very large sample sizes that allow them to characterise distributions and obtain a target density function. It is becoming increasingly common for personal computers and laptops to have quad-core or even octa-core processors, meaning that the average user of the Metropolis-Hastings algorithm could potentially reach speeds of up to eight times faster if they parallelise their code.

It is also important to appreciate that our method of choice, the parallel Markov chain method, is relatively straightforward to implement, making it quite versatile. If the function (proportional to the true posterior density of interest) from which we want to generate MH samples is already defined in the Python script, then the only steps that need changing from the non-parallel case to the parallel case are -

1. the splitting of the sample size at the beginning of the program and
2. the unification of the separate samples generated at the end.

The *'embarrassingly parallel'* nature of this method makes it such an attractive approach here.

4.3 Considerations when Parallelising the Metropolis-Hastings Sampler

We have found that the benefits of parallelising the MH algorithm can be up to C times faster *in the best case*. This best-case scenario can be split up into two factors: short burn-in and large sample size.

We will first discuss the burn-in problem. We know that the most important property of the Markov chains that we form is convergence, which is of course not immediate. That means that it may sometimes be necessary to remove the first few hundreds or thousands of states in the chain, where convergence may not yet be reached (Kruschke, 2015). This is because these values may not represent an accurate sample of the data. Referring back to our toy example, let's assume that we wanted a sample size of 10,000 from the distribution and we knew beforehand that the chain would take 2,000 iterations to converge. This means that we would need to create a chain of size 12,000 in the non-parallel case, which took **46.4 seconds** according to our table above. However, if we wanted to run parallel Markov chains, we would need to generate a larger number of samples. This is because we are now discarding 4 * 2,000 = 8,000 samples, meaning we need a total of 18,000 to be left with a final sample size of 10,000. This scenario took **21.1 seconds** to run, meaning that by introducing a burn-in of 2,000, the parallel method went from over three times faster to just over two times faster than the non-parallel case.

However, since the runtime scales linearly with time (as clearly seen in the table), if we want to draw a sample of size 100,000 from the posterior after removing a burn-in of 20,000, the parallel and non-parallel times were found to be **458.3 seconds** and **207.5 seconds** respectively. Here, the larger sample size makes parallelisation more appealing in terms of

absolute runtime, even though the relative difference is of the same magnitude as the sample of 10,000 with a burn-in of 2,000. We maintain that the absolute improvement in running time makes parallelisation of the MH algorithm worth the effort for sufficiently large samples, although this will ultimately depend on the user's application of the sampler.

The maximal improvement in runtime is attained when sample size is also very large. This is due to communication times betwen the different processes in Python. For an extremely short program, there is no need to parallelise as the time saved from doing so is negligible compared to the communication cost. For example, if we take the same toy example but with a sample size of just 100, the non-parallel algorithm takes just **0.44** seconds to implement, while the parallel code takes nearly three times as long (**1.4 seconds**). This makes parallelisation *inefficient* for insufficiently large sample sizes. Fortunately, this is not a feasible example for a Monte Carlo simulation and by using a more realistic sample size, the code's synergies quickly come into play. For instance, for as few as 1,000 samples, the parallel runtime of **2.5 seconds** is already faster than the non-parallel runtime of **3.8 seconds**.

5. Conclusion

In this report, we have provided a brief review of parallel computing and its implementation via the Multiprocessing module in Python. We have shown the benefits of parallelisation in the context of Markov chain Monte Carlo methods, and more specifically, the Metropolis-Hastings algorithm. Our implementation provides useful intuition as to the purpose of the MH sampler for Bayesian inference, an area that has exploded in popularity thanks to recent advances in computational tools. From our results, we have found that for those who work in fields of research and/or industry who regularly use MCMC methods, it is largely much more beneficial to adopt a parallel approach. In most practical cases, the number of samples will be much larger than the 10,000 used in our toy example, at which point the runtime of single-chain sampling can be very long. This is especially the case for problems of high-dimensionality, for example in Bayesian hierarchical models with >100 parameters to estimate. In such situations, the speed of the algorithm can be increased by a factor of close to C, the number of cores of the user's machine, by implementing our relatively straightforward parallel Markov chains approach.

However, it is still important to be mindful about the size of burn-ins. If the size of the burn-in is substantially large, especially in comparison with the total chain size, then parallelisation may not be as effective as expected, and in the worst case, it can even slow down performance. Still, in most practical applications, thanks to large sample sizes and long non-parallel runtimes, the parallel approach represents a significant absolute improvement in the speed of execution.

On the basis of our results, we strongly advise those using MCMC methods to at least utilize a parallel Markov chains approach as demonstrated here (and more fully in the Appendix). For now, it is clear that this intuitive adjustment to existing code can drastically improve efficiency. In future work, we would like to explore and experiment with the implementation of some alternative parallelisation techniques, which could potentially provide even larger improvements in performance.

6. References

Calderhead, B. (2014), 'A general construction for parallelising Metropolis - Hastings algorithms', *Proceedings of the National Academy of Sciences* 111(49), 17408–17413.

Feng, X., Buell, D. A., Rose, J. R. & Waddell, P. J. (2003), 'Parallel algorithms for Bayesian phylogenetic inference', *Journal of Parallel and Distributed Computing* 63(7-8), 707–718.

Kruschke, J. (2014), 'Doing Bayesian Data Analysis: A Tutorial with R, JAGS and Stan'.

Neiswanger, W., Wang, C. & Xing, E. (2013), 'Asymptotically Exact, Embarrassingly Parallel mcmc', *arXiv preprint arXiv:1311.4780*.

Omni Sci (n.d.), 'Parallel computing', https://www.omnisci.com/technical-glossary/parallel-computing.

Robert, C. & Casella, G. (2013), *Monte Carlo Statistical Methods*, Springer Science & Business Media.

Rosenthal, J. S. (2000), 'Parallel computing and Monte Carlo algorithms', *Far east journal of theoretical statistics* 4(2), 207–236.

Stephens, M. (2018), 'The Metropolis Hastings Algorithm', https://stephens999.github.io/fiveMinuteStats/MH_intro.html.

Strid, I. (2009), 'Efficient parallelisation of Metropolis – Hastings algorithms using a prefetching approach', *Computational Statistics & Data Analysis* 54(11), 2814–2835.

The Python Standard Library (n.d.), 'Multiprocessing – process-based parallelism', https://docs.python.org/3/library/multiprocessing.html.

Universedecoder (2018), 'Introduction to Parallel Computing', https://www.geeksforgeeks.org/introduction-to-parallel-computing/.

VanDerwerken, D. N. & Schmidler, S. C. (2013), 'Parallel Markov Chain Monte Carlo', *arXiv preprint arXiv:1312.7479*.

Appendix

Here, we present the full implementation of our Metropolis-Hastings algorithm. The user can change the sample size, burn-in, and any other parameters throughout the code and additionally specify whether to run the non-parallel or parallel version:

In [ ]:

```
# Import required modules
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import timeit
import scipy.stats as st
from multiprocessing import Pool
from functools import partial


# Define function to calculate log-posterior density, from which we wish to sample from
def log_posterior(data, mu, sigma):
    # Assumed priors are mu~N(0,100) and sigma~IGamma(0.01, 0.01)
    log_prior = np.log(st.norm(0, 10).pdf(mu)) + np.log(st.invgamma(a=0.01, scale=1 / 0.01).pdf(sigma))

    # Know (or assume if not known) that the data comes from a normal distribution
    log_lik = np.sum(np.log(st.norm(mu, sigma).pdf(data)))

    # Log-posterior density is the sum of the log-prior and log-likelihood
    log_post_density = log_prior + log_lik
    return (log_post_density)


def MH_sampler(samples, data, mu_init, sigma_init, proposal_sd=[0.05, 0.05]):
    # Initialisation
    acceptances = 0
    mu_current, sigma_current = mu_init, sigma_init
    chain = [[mu_current], [sigma_current]]  # First value in the sample
    rejected = [[np.nan], [np.nan]]

    for _ in range(1, samples):
        # Draw Proposals
        mu_proposal, sigma_proposal = np.random.normal([mu_current, sigma_current], proposal_sd, (2,))

        # Calculate Acceptance Probability
        proposal_log_posterior = log_posterior(data, mu_proposal, sigma_proposal)
        current_log_posterior = log_posterior(data, mu_current, sigma_current)
        p_accept = np.min([1, np.exp(proposal_log_posterior - current_log_posterior)])

        # Accept/Reject Step
        if np.random.uniform(0, 1) < p_accept:
            acceptances += 1
            rejected[0].append(np.nan), rejected[1].append(np.nan)
            mu_current, sigma_current = mu_proposal, sigma_proposal
        else:
            rejected[0].append(mu_proposal), rejected[1].append(sigma_proposal)

        chain[0].append(mu_current), chain[1].append(sigma_current)

    return (chain, rejected, acceptances)


if __name__ == "__main__":

    # Generate data
    np.random.seed(444)
    data = st.norm(5, 2).rvs(1000)
    mu_obs = np.mean(data)
    sigma_obs = np.std(data)
    plt.hist(data)
    plt.title('Histogram of N(5,2)')
    plt.show()
    print('Observed sample mean = ' + str(mu_obs) + '\nObserved sample std = ' + str(sigma_obs))

    # Set up parallel/non-parallel runs
    loop = True
    while loop == True:
        toggle = input("Enter 'P' for parallel MH or enter 'X' for non-parallel MH: ")
        if toggle == 'P':
            parallel, loop = True, False
            print('Running parallel MH...')
        elif toggle == 'X':
            parallel, loop = False, False
            print('Running unparallel MH...')
```

```
    else:
        print("ERROR: Only enter 'P' or 'X'.")

# Designated number of samples to be drawn
samples = 100000

# Designated # of Burn-in steps
burnin = 0

# Start the timer (assume time to check value of 'parallel' is negligible)
start = timeit.default_timer()

# Generate MH samples (non-parallel)
if parallel == False:
    nchains = 1
    np.random.seed(444)
    results = [MH_sampler(samples, data, mu_obs, sigma_obs)]
    chain, rejected, n_accept = results[0][0], results[0][1], results[0][2]
    chain[0] = chain[0][burnin:]
    chain[1] = chain[1][burnin:]
    rejected[0] = rejected[0][burnin:]
    rejected[1] = rejected[1][burnin:]
    print("Acceptance Rate): ", 100 * n_accept / samples, "%")

# Generate MH samples (non-parallel)
elif parallel == True:
    nchains = 4
    samples_per_chain = int(samples / nchains)
    p = Pool(processes=4)
    target = partial(MH_sampler, samples_per_chain, data, mu_obs, sigma_obs)
    results = p.map(target, [[0.05, 0.05] for _ in range(nchains)])
    chain = [[], []]
    rejected = [[], []]
    for i in range(nchains):
        chain[0].extend(results[i][0][0][burnin:]), chain[1].extend(results[i][0][1][burnin:])
        rejected[0].extend(results[i][1][0][burnin:]), rejected[1].extend(results[i][1][1][burnin:])
        n_accept = results[i][2]
        print("Acceptance Rate (chain ", i + 1, "): ", 100 * n_accept / samples_per_chain, "%")

# End the time and analyse run time
end = timeit.default_timer()
print("Run Time:", str(end - start), "secs")

for i in range(nchains):
    # Trace Plots of mu and sigma for each chain
    fig, axes = plt.subplots(ncols=2, figsize=(12, 6))
    for j in range(2):
        axes[j].plot(results[i][0][j])
        axes[j].set_xlabel('Iteration')
    axes[0].set_ylabel('$\mu$')
    axes[1].set_ylabel('$\sigma$')
    if parallel:
        plt.suptitle("Trace Plots for $\mu$ and $\sigma$ \nChain " + str(i + 1))
    else:
        plt.suptitle("Trace Plots for $\mu$ and $\sigma$")
    plt.show()

# Joint Accept-Reject plot of mu-sigma
mh_data = pd.DataFrame({'mu_samples': chain[0], 'mu_rejected': rejected[0], \
                        'sigma_samples': chain[1], 'sigma_rejected': rejected[1]}).reset_index()

fig, axes = plt.subplots(figsize=(12, 6))
sns.scatterplot(x='mu_rejected', y='sigma_rejected', data=mh_data, color='red', marker='x')
sns.scatterplot(x='mu_samples', y='sigma_samples', data=mh_data, color='blue', linewidth=0.1, s=50)
axes.set_xlabel('$\mu$')
axes.set_ylabel('$\sigma$')
axes.set_title('Metropolis Hastings samples for the posterior distributions of $\mu$ and $\sigma$');
axes.legend(labels=['Rejected Proposals', 'Samples'])
plt.show()

# Traceplots and Histograms of mu and sigma individually
fig, axes = plt.subplots(ncols=2, figsize=(12, 6))
for j in range(2):
    sns.histplot(chain[j], stat='density', color='red', ax=axes[j])
    axes[j].set_ylabel('Density')

axes[0].set_ylabel('$\mu$')
axes[1].set_ylabel('$\sigma$')
plt.suptitle("Histograms for Posterior Density Functions of $\mu$ and $\sigma$")
plt.show()
```