Architectures and GPU-Based Parallelization for Online Bayesian Computational Statistics and Dynamic Modeling

A thesis submitted to the
College of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Lujie Duan

Abstract

Recent work demonstrates that coupling Bayesian computational statistics methods with dynamic models can facilitate the analysis of complex systems associated with diverse time series, including those involving social and behavioural dynamics. Particle Markov Chain Monte Carlo (PMCMC) methods constitute a particularly powerful class of Bayesian methods combining aspects of batch Markov Chain Monte Carlo (MCMC) and the sequential Monte Carlo method of Particle Filtering (PF). PMCMC can flexibly combine theory-capturing dynamic models with diverse empirical data. Online machine learning is a subcategory of machine learning algorithms characterized by sequential, incremental execution as new data arrives, which can give updated results and predictions with growing sequences of available incoming data. While many machine learning and statistical methods are adapted to online algorithms, PMCMC is one example of the many methods whose compatibility with and adaption to online learning remains unclear.

In this thesis, I proposed a data-streaming solution supporting PF and PMCMC methods with dynamic epidemiological models and demonstrated several successful applications. By constructing an automated, easy-to-use streaming system, analytic applications and simulation models gain access to arriving real-time data to shorten the time gap between data and resulting model-supported insight. The well-defined architecture design emerging from the thesis would substantially expand traditional simulation models' potential by allowing such models to be offered as continually updated services. Contingent on sufficiently fast execution time, simulation models within this framework can consume the incoming empirical data in real-time and generate informative predictions on an ongoing basis as new data points arrive. In a second line of work, I investigated the platform's flexibility and capability by extending this system to support the use of a powerful class of PMCMC algorithms with dynamic models while ameliorating such algorithms' traditionally stiff performance limitations. Specifically, this work designed and implemented a GPU-enabled parallel version of a PMCMC method with dynamic simulation models. The resulting codebase readily has enabled researchers to adapt their models to the state-of-art statistical inference methods, and ensure that the computation-heavy PMCMC method can perform significant sampling between the successive arrival of each new data point. Investigating this method's impact with several realistic PMCMC application examples showed that GPU-based acceleration allows for up to 160x speedup compared to a corresponding CPU-based version not exploiting parallelism. The GPU accelerated PMCMC and the streaming processing system can complement each other, jointly providing researchers with a powerful toolset to greatly accelerate learning and securing additional insight from the high-velocity data increasingly prevalent within social and behavioural spheres.

The design philosophy applied supported a platform with broad generalizability and potential for ready future extensions. The thesis discusses common barriers and difficulties in designing and implementing such systems and offers solutions to solve or mitigate them.

Acknowledgements

First, I would like to express my sincere appreciation to my supervisor, Dr. Nathaniel Osgood, who has expertly guided me through the master's study and research and inspired me to explore innovative ideas and developing innovative methods. It has been a great pleasure and honour working with Dr. Osgood for the years.

I would like to pay my special appreciation to our lab assistant, Christine Hillis. She has been working tirelessly to ensure the efficient operation of the CEPHIL lab while paying attention and taking care of everyone in the lab for things big or small. I would also like to thank my colleague and member of the lab: Weicheng Qian, Bo Pu, Xiaoyan Li, Yang Qin, Wade McDonald, Yuan Tian and many others, with whom I worked closely every day in the lab before the pandemic, and provided support for each other when working from home. Specifically, I want to thank Weicheng and Bo for their exceptional help, resources, advising, and feedback on implementing the high-performance Bayesian methods on graphics processing units; Wade for working on and sharing the Emergency Department Waiting AnyLogic model that is used as a demonstration in this thesis; Xiaoyan and Yang provided valuable input and numerous guidance on constructing simulation models and combining with Bayesian methods; Yuan for inspiring works on social medium streaming and machine learning. I want to acknowledge the contribution of former member of the lab, Koushik Pal, for his work on GPU-based multinomial resampling that has have been applied in this thesis.

I want to thank Dr. Brian Rector, Delphine Gossner, Keira Stockdale, Dr. Raymond Spiteri, and those who work on the Saskatchewan Police Predictive Analytics Lab project; they have been providing a significant number of valuable insights and advice for the work presented in this thesis.

Also, I would like to acknowledgement Compute Canada, for providing resources, including computing clusters Cedar, Graham, Beluga, and Compute Canada Cloud, for supporting the running of experiments presented in this work. Other computing resources used include the servers provided by the Department of Computer Science, the University of Saskatchewan, for which I want to thank the IT team of the department — without whom it would not be possible to carry out all those tests and experiments smoothly on the department's servers.

Contents

List of Tables

List of Figures

List of Abbreviations

SIR	Susceptible-Infected-Recovered
SEIR	Susceptible-Exposed-Infected-Recovered
GPU	Graphics Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
SSM	State-Space Model
PF	Particle filter
MCMC	Markov Chain Monte Carlo
PMCMC	Particle Markov Chain Monte Carlo
API	Application Programming Interface
REST	Representational state transfer
DSMS	Data Stream Management System
PCIe	Peripheral Component Interconnect Express

1 Introduction

1.1 Motivation

The growing availability of high-variety, high-velocity, high-veracity data sources related to human behaviour raises prospects for grounding increasingly articulated dynamic models emerging from research into human behaviour. The enormous amount of digital information available makes possible rapid advances and research, and relies more on analytics around the collected data than before. Performing fast and accurate interpretation and prediction of systems and events are now achievable. Public health and community well-being studies are no exception [3], but doing so requires effectively addressing several important challenges. The first one is that the data comes in different formats, sources, and frequencies. Often, it demands the researchers to tailor each data source's collecting process to build successful data processing pipelines. Secondly, after having the data ready, difficulties arise choosing suitable algorithms and modeling methods to get reliable results and valuable insights from the data. Some algorithms only work for specific problems with certain characteristics; some require significant efforts to preprocess, select or label the data; others impose extensive learning, implementing, and executing burden on the users. Also, even the richest single data source typically offers evidence regarding only a confined subset of intrinsically multidimensional human behaviour; grounding richer behavioural models typically requires drawing on more than one such dataset. There is a need for a principled meshing of such diverse sources of temporal behavioural data with theory captured in a model, one that takes into account the fact that often such data relates to emergent model and system behaviour. A third issue regards the delay between data availability and analysis results in such data science studies. For some methods, results and predictions may become available days or weeks later and may already be obsolete. We know that the value of the data is tied to the processing speed and the amount of valuable information represented by the data will decrease as time elapse, with that information typically becoming irrelevant after a certain period; the processing time is a critical determinant of the value of an analytic approach and often of the data that it interprets.

Public health, community safety and well-being research can take advantage of past collected data on individuals and communities to study the epidemiology of diseases and other health and social burdens, and related factors concerning behaviours and social contacts. The magnitude of data available in this domain has soared with two foremost reasons.

The first is that there are more and higher velocity channels than ever before to collect those kinds of data. The digitization of medical equipment and hospital system means that almost all the patients' vital readings and diagnoses can be stored electronically. Smartphones routinely record a wide range of sensor data, while personal wearable devices building up the body area networks have become common.

The second, social media, use of search engines and online posting and interactions have become inseparable elements of modern social life. The advancement of computer hardware and networks simplifies the storing, processing, and sharing of such data. Each of these spheres of data collection is associated with opportunities for employ the data so collected for analysis purposes.

Online configurations of algorithms – such that they can incrementally accept new data as it becomes available – offers significant benefits for machine learning and data science research, and make possible otherwise infeasible applications. Examples are Artificial Neural Networks that scan images incrementally to train and reach levels of object detection accuracy that would not otherwise be possible. Such configuration also makes applications more responsive to emerging events and enables them to revise predictions in real-time to provide the most up-to-date projection, often with great benefit in terms of projection accuracy.

Bayesian computational methods are a series of statistical methods formulated around algorithms exploiting Bayesian probability and statistics. Compartmental modeling is a common epidemiology mathematical representation where the population is divided into mutually exclusive compartments based on their health state. System Dynamics modeling, one of the dynamic modeling methods, is a quantitative method for compartmental modeling focusing on the feedbacks and accumulations within the system. When coupling Bayesian computational methods with dynamic modeling – such as those widely used in mathematical epidemiology – such computational methods can be used to create robust inference applications to explore system states and parameters shaping the dynamics. Even though the exact solutions only exist under strong assumptions for simple lower dimensional systems, we can get a good enough approximated solution for complex systems through sampling-based methods. Such methods include the sequential Monte Carlo method termed the Particle Filter (PF) [4], Markov Chain Monte Carlo (MCMC) [5], and Particle Markov Chain Monte Carlo (Particle MCMC, PMCMC) [6] that have been proven effective but usually impose a substantial computational burden.

Particle MCMC is a collection of particularly powerful Bayesian computational statistical estimation frameworks suitable for a broad range of estimation problems, including those that combine multiple time series with dynamic models. When applied to a dynamic model representing the posited structure of the underlying system, the Marginal Metropolis-Hastings (MMH) Particle MCMC method on which this work focuses explicitly supports estimating the joint posterior distribution of both static model parameters and temporal trajectories sampled from the latent state of the dynamic model. This method provides a means of arriving at a probabilistic consensus picture of the underlying system and parameter values that reflects both theory captured in a dynamic model and unfolding data. Unless otherwise stated, use of the term Particle MCMC within this thesis will henceforth refer specifically to the MMH variant of Particle MCMC. While there are many successful applications of the Particle MCMC method, the computational resource demands and prolonged running time required constitute top challenges.

While different programming languages and environments provide implementations of the Particle MCMC method, they suffer from several notable disadvantages when applied with dynamic models. Particle MCMC implementations in statistical languages and environments such as R support ready integration and interaction, but can suffer from slow execution that limits them to smaller models and simpler inference tasks. Others taking advantage of GPU or special hardware are challenging to integrate with large compartmental simulation models. Previous work from the authors includes a Particle MCMC library in the C language and offering R bindings that is designed to work specifically with compartmental System Dynamics (SD)/Ordinary differential equation (ODE) models [7]. While the high-efficiency C implementation secures notable performance gain over implementations in high-level languages, running times remain high, particularly with large particle complements, larger MCMC chains and due to the numerical integration overhead in larger SD models. Given that large compartmental models will in general require larger particle complements for effective sampling, it is not unusual for run times with such models to extend over days or weeks. This work demonstrates a high-performance graphics processing unit (GPU)-based implementation accelerating Particle MCMC use with SD simulations while retaining modularity with respect to model choice. We demonstrate a significant reduction in the execution time, resulting in a quick convergence of the algorithm and a markedly reduced interval between empirical data availability and model results.

Using Bayesian methods to estimate the state and parameters in an online configuration would mean the posterior probability of dynamic states and static parameters is calculated incrementally. This feature is essential for solving problems in robotic, localization, navigation, epidemic simulation and decision support[8]; this makes supplementing Bayesian methods with the capacity for online learning an attractive idea. In terms of non-iterative methods such as PF, converting to online is as simple as plug-and-play, as the application will use the new data right away when available and the algorithm can progress forward. For iterative methods such as MCMC and Particle MCMC, the strategy needed and the benefit of online learning is worth exploring.

In recent years, many solutions have been developed to deal with the growing needs for streaming and online processing for data science and machine learning algorithms. Those works usually include building the application or service to stream and process a fixed number of data sources for their use cases. Such pipelines would generally contain a data harvesting system that is responsible for ingesting and preprocessing the raw data. Then the data would be used in various mathematical and statistical algorithms, machine learning models or simulations, such as with a Hidden Markov model (HMM), Convolutional Neural Network (CNN), and Markov Chain Monte Carlo (MCMC) methods. Those machine learning algorithms are commonly built using Python, Apache Spark, TensorFlow and others. The intermediate and final results would be saved to files to persist on disk, and visualization can be generated and hosted as web pages. Most of such systems are static and coupled, and modifications to the system can be time-consuming and sometimes involves complete redesign and redeployment. Flexible, easy-to-manage and high-performance solutions are needed, especially for public health and social behaviour-related research and simulations.

Agencies and authorities are responsible for making judicious, expeditious and accurate decisions based on promptly collected data with up-to-date, precise information. Simulation models are powerful tools for informing and aiding decision-making. Often, a considerable amount of different structured and parameterized models run simultaneously to generate scenarios that can provide knowledge from different angles and make projections and predictions of the current situations. Manually updating the dataset used by the model each time new data points become available is infeasible, and it is currently infrequent for modeling software to support such functionality natively. There is also needs to properly design the accompanying software infrastructure for a complete solution. The Particle MCMC methods have recently been applied to public health simulations and dynamic models, and demonstrate great promise in jointly inferring states and parameters. The challenge with this combination is that the computation burden is compounded between the System Dynamics modeling and Particle MCMC algorithms. Moreover, one must carefully tune model structures and parameters before a Particle MCMC configuration is well-adapted to the situation, and then one can run the model and conclusions can be drawn.

Traditionally, when the requirement of shortening processing time arises, common solutions include optimizing the algorithm and scaling up the hardware on which the algorithm runs. Vertical and horizontal scaling up can shorten the running time by providing a more powerful single machine or distributing the program to a cluster of interconnected machines, respectively. While it can quickly reach the ceiling of scaling up a single machine, distributed programming requires the algorithm to have dividable sub-tasks and limited synchronization. The communication delay over the network can be non-trivial. Cloud computing has also become another attractive strategy, but is marked by certain risks and tradeoffs. Specialized hardware can increase the speed for certain tasks by providing appropriate instruction sets, unique memory and disk accessing strategies or fast inter-process or network communication and it is also common to design and manufacture hardware targeting the algorithms known as Application Specific Integrated Circuit (ASIC). Both of those require a thorough understanding of the algorithm, application and hardware and may place the use outside the reach of most researchers.

The main problem this work seeks to solve is to find solutions for effective and efficient streaming for computational statistics algorithms of Particle Filtering and Particle MCMC combined with System Dynamics models. This work chooses graphics processing units as the hardware accelerator for Particle MCMC algorithms combined with the models because the computation pattern of such applications is well-suited for the GPU architecture, where the particles in the form of copies of models can run in parallel on GPUs and a large count of particles are needed for the experiments. CUDA-enabled GPUs are also selected as the target accelerators because of the wide availability of the hardware and the great support of the software development kit.

This thesis's primary research goal is to design, implement and evaluate a performance-oriented system to support Particle Filtering and Particle MCMC methods with dynamic models running on incrementally delivered data. The system includes a data streaming architecture for data gathering, processing and handling interfacing with dynamic models serving as the state space models for PF and Particle MCMC methods, and the codebase that uses graphics processing units to accelerate the Particle MCMC method to achieve the speed needed for streaming processing and online learning.

The objectives of this work are to:

- Use modern software and frameworks to design and implement a data streaming system to reduce the time delay between the availability of data and resulting useful information/insights by supporting public health and social behaviour-related dynamic models employing PF and Particle MCMC methods;

- Speed up computationally intensive Particle MCMC algorithms employing dynamic models using easy-to-access graphics processing units and the CUDA programming toolkit;

- Investigate strategies for expanding Particle MCMC with dynamic models to online and incremental applications to take advantage of the dynamic streaming data.

This thesis conducted several lines of work to address the objectives. A streaming architecture aiming for extensibility, scalability, and robustness was proposed. To better take advantage of unfolding temporal data sets in real-time, this work introduced a novel GPU-based approach to accelerating conducting Particle MCMC with compartmental System Dynamics simulations to estimate parameters and states quickly. The CUDA-powered parallel Particle MCMC codebase contributed by this work uses interface-based modularity to support different simulations and equations on fast GPUs so as to make Particle MCMC easier for tuning and even adapting to streaming. Though the algorithm itself is not as intuitive as PF for the streaming process [9], this thesis designed, implemented and reported the outcomes of a basic assessment of adapting the Particle MCMC framework to support a streamed context.

Scientific and engineering contributions of this work are as follows:

- The design and implementation of a system for applying Particle Filtering and Particle MCMC methods with streaming online processing configuration and dynamic modeling.

- Demonstrated success in production-quality deployments of variations of that platform as a centrepiece of research and reporting underway in CEPHIL. These infrastructural variations foundationally enabled delivery on daily reporting for COVID-19 reporting contracts from Saskatchewan and Canadian Federal agencies. This work has supported many thousands of Particle Filtered runs for regular reporting and research.

- Design and implementation of GPU-based optimization of Particle MCMC algorithms. This work is also in production use as a key component of CEPHIL research and contract delivery.

- Evaluation of the performance gains extending from this GPU implementation when used with different combinations of dynamic models, testing machines and environments. These experiments sought to provide an in-depth understanding of the impact of hardware configurations and models on the performance tradeoffs associated with the GPU implementations. Within these experiments, the fully implemented GPU version of Particle MCMC with dynamic models exhibited an average speedup of over 30 times, and up to 160-fold compared to the non-parallel version while maintaining the same flexibility and usability in application to different research and simulation needs;

- Articulation of detailed case studies characterizing the usage and advantages of the proposed system and codebase; this included consideration of several use cases for the streaming processing system and public health simulation model implemented with the CUDA Particle MCMC framework;
- Findings from preliminary investigations of the effectiveness of online streaming Particle MCMC with System Dynamics models. These findings demonstrate promising improvements and avenues for future extension.

1.2 Publication

A paper has been presented at and published in the Proceedings of the International Conference on Social Computing, Behavioural-Cultural modeling and Prediction and Bebavior Representation in Modeling and Simulation (SBP-BRiMS 2021) in June 2021, based on the content of chapter 4 and 5. The citation is as follows:

L. Duan and N. Osgood, “GPU Accelerated PMCMC Algorithm with System Dynamics Modelling,” in *Social, Cultural, and Behavioral Modeling*, Lecture Notes in Computer Science, pp. 101–110, Cham: Springer International Publishing, 2021

1.3 Thesis Organization

The balance of this thesis is structured as follows. Chapter 2 provides background information regarding the field we are exploring, including algorithms and intuitions behind Bayesian inference of state-space models, sampling-based estimation methods PF and Particle MCMC and System Dynamics modeling; also works and strategies of building data streaming systems as well as GPU programming using CUDA toolkit. Chapter 3 discusses the structure, design and implementation of the system for streaming analysis, sample use cases and performance evaluation in depth. Chapter 4 examines the benefits of conducting Particle MCMC with compartmental simulation models on graphics processing units. The Particle MCMC codebase migration to GPU is discussed in detail, and is followed by an evaluation of system performance tradeoffs. This work includes an examination of the system operation with a sample compartmental model to demonstrate Particle MCMC CUDA and streaming integration. Chapter 5 switches to a real-world influenza simulation model with additional texture to demonstrate the performance pattern and the usage of the system in depth. The thesis concludes in chapter 6 with the discussion of streaming analytics with Bayesian statistics methods, as are possible extensions and future research work.

2 Background and Related Work

2.1 Introduction

This chapter discusses the algorithms and technical background, mathematical notation, and previous research work related to the thesis topics. One of the thesis foci is on Bayesian statistics methods - those methods, including Particle Filter and the Particle Markov Chain Monte Carlo family, have been successfully applied to a wide range of problems. Such methods can offer notable value for public health and behavioural data science research, especially when joined with simulation models that characterize real-world events and systems. Instead of offering a point estimation, such Bayesian methods can sample from a posterior distribution as the output. This capacity to characterize a distribution over the sampled quantities – whether system state, values of static parameters, or both – offers a substantial advantage in supporting effective decision making by facilitating understanding of the current state of the system, and the resulting ability to project that state forward in baseline or "what if" questions. Particle Filtering is one application of Bayesian Recursive Estimation for statistical state inference. Particle MCMC is a powerful extension of the MCMC method to jointly sample from parameters and from trajectories involving the latent state of the system, thereby targeting complex problems in which both parameter values and system state are unknown.

This thesis specifically considers the use of these Bayesian computational statistics methods together with dynamic simulation models. This chapter first considers previous work and research on Particle Filter to Particle MCMC, from their mathematical intuitions, application to various disciplines, the strategy of combining them with empirical data and simulation models, efforts to incorporate them into a streaming context, and work towards accelerating the computations.

Research on streaming data processing, online machine learning and their software architectures will be presented here as the starting point for later discussion. The following section will present algorithmic detail, from the generic ones to specific applications and variations, discussing Bayesian inference, recursive estimation, Particle filter, Monte Carlo processes, MCMC and, finally, Particle MCMC methods. This chapter further considers literature regarding general-purpose graphic processing units (GPUs), particularly the associated programming model, best practices for heterogeneous programming on those specialty processors, and the use of GPUs to accelerate statistics algorithms.

2.2 Related Work

Speaking informally, the general idea of Bayesian recursive estimation is to use observations one by one to update prior belief regarding the distribution of system state to generate the posterior belief distribution regarding that state. Exact solutions for Bayesian recursive estimation are available from methods including Kalman filtering and extended Kalman filtering for linear systems. Such optimal methods require the likelihood functions to be available analytically and provide a maximum likelihood estimate of the system state, together with a covariance matrix expressing uncertainty in such estimates. Kalman filtering further imposes strong assumptions regarding the distributions involved, specifically in assuming that both stochastics applied to system behaviour and measurement error are normally distributed. If the distributions in question are non-normal (non-Gaussian) or impossible to derive analytically, Particle Filtering can frequently offer a good approximate solution by using weighted particles to approximate the posterior distribution. This posterior distribution is updated at each time step following Bayes' rule.

Monte Carlo is a family of repeated sampling methods for estimating the posterior. The Particle Filter method is also referred to as one realization of the sequential Monte Carlo (SMC) method. Markov Chain Monte Carlo is a technique to generate Monte Carlo samples. The word *Markov* indicates that experiment results, or samples, are independent: in a Markovian process, the next state would only depend on the previous state but not the states before. While the new sample only depends on the location of the previous state, the samples will follow the distribution used as the equilibrium distribution of the chain. Metropolis-Hastings (MH) [11, 12] and Gibbs sampler[13] are such samplers for MCMC; MH starts by generating a sample using the proposal distribution and only accept this sample when it satisfies criteria of acceptance; with many samples generated the distribution of the samples will be a close approximation of the target distribution. MCMC methods can sample from hard-to-compute distributions, such as those joint distributions of underline state and unknown parameters, where the actual analytical results are not possible to calculate. However, even with samplers such as MH, sometimes it can be hard to sample and accept efficiently, especially when the dimension of the problem is high. Thus, one can use PF to get the approximated posterior given the observations and proposed parameter value and then work with this posterior to calculate the acceptance. It has been shown that under certain assumptions, Particle MCMC, the combination of PF with MCMC is an optimal estimation. When combined with System Dynamics simulations, those can help to model and understand a wide range of real-world problems.

2.2.1 Recursive Bayesian Estimation

Recursive Bayesian estimation is a general and powerful probabilistic method for estimation in the context of sampling from joint probability density functions (PDF) for the latent state of a dynamic system in a recursive and step-by-step manner. It is also known as the Bayesian filtering, as it estimates the belief about the states recursively as the data arrive to filter out the noise and increase the confidence of the estimation[14]. This method can apply to general tracking with dynamics problems. For a dynamic system with unobservable underlying states and noisy measurements at each time step, we want to predict or track the distribution of the state over time. Tracking problems assume that the object or the state evolves gradually and smoothly rather than discontinuously. It also follows the Markovian assumption where the future state only depends on the current state but not earlier states. Approaches such as the Kalman filter (KF), extended Kalman filter (EKF), grid-based filter and Particle Filter (PF) all fall into the category of recursive Bayesian estimation, each focusing on a distinct set of problems and offering different trade-offs.

The Kalman filter is an efficient and straightforward approach but requires strong assumptions regarding the underlying system. The state evolves following a particular dynamic process with randomness, and the measurements should be related to the actual states but involve noises. The KF will be the optimal estimator [15] if the following conditions are satisfied: known linear functions characterize the evolution and measurements, and the process noises and measurement noises (sensor noises) are Gaussian distributed [16], i.e., a linear-Gaussian state-space model (LG-SSM)[14] applies. The EKF is a KF variant that can apply nonlinear Gaussian dynamics systems, where the evolution (transition model) and measurements (observation model) are nonlinear. However, both measurement and process noise still follow a Gaussian distribution. The idea behind EKF is to linearize the two models using Taylor series expansion around successive states of the model so that ordinary KF can apply. Navigation systems, robotics, image/video object tracking and GPS are sample applications of EKF. Beyond the assumptions given above, both KF and EKF are notable for their assumption that the system can be well characterized in state equations, and for their use of a formulation that quantifies uncertainty in the process's latent state according to covariance around a single maximum likelihood estimate (MLE). Grid-based methods can also find optimal solutions given that the system only has a finite set of discrete states [14]. The Hidden Markov Model (HMM) is a highly effective grid-based application on speech recognition and processing, signal processing, and pattern finding. One example is the study of human behaviours such as daily smartphone usage with HMM models [17].

For nonlinear and non-Gaussian systems with continuous states, Particle Filtering can approximate the joint probability density functions of latent states at distinct points over time using a set of importance-weighted samples known as *particles* [4]. It offers several notable advantages. First, it is similar to KF but with relaxed system assumptions. Specifically, by sampling explicitly from the PDF for the latent states, the PF dispenses with the need to rely upon a privileged MLE estimate of system state. This is a particularly valuable generalization for nonlinear models with distinct basins of attraction, for which the PDF of state variables can readily be multi-modal. Second, PF does not rely upon a specific distribution for process and measurement noise. Third, while not exploited here in this work, it bears noting PF does not assume that the underlying system evolution is amenable to state equations, thus permitting its application to a broader class of dynamic models, such as agent-based models [18]. Particle Filter methods can be found in state-space dynamic tracking applications, online parameter learning, and time-series forecasting problems and are used for navigating, location tracking, and signal processing [19].

2.2.2 Particle Filtering with Simulation Models

The approach described in [20] employs PF to recursively reground the compartmental simulation models using empirical data. Experiments were carried out using synthetic empirical data on different model networking types. The synthetic empirical observation data was created first using an agent-based model with known parameters to evaluate the models. The mean squared prediction error is used as the evaluation metric to compare this with ordinary models without Particle Filtering. Specifically, at each time step of the simulation model, the discrepancy metric draws a certain number of samples from the prior predictive distribution. It sums up the individual squared difference between the states from the samples and the states from the synthetic data. The results were normalized by the time length and sample size, and then the take root to obtain the per particle discrepancy. All results showed that, regardless of model networking type, by introducing Particle Filtering into the models, the discrepancy was reduced by at least several orders of magnitude compared to the case without it.

Several applications were developed based on this work using Particle Filter with compartmental simulation models and collected empirical data. In [21] a SEIRV (**S**usceptible, **E**xposed, **I**nfectious, **R**ecovered, **V**accinated) System Dynamics model for the H1N1 combined with PF has been tested on weekly reported cases of H1N1 influenza for 14 consecutive weeks between 2009 to 2010. The experiment results showed that when predicting one week into the future, by comparing one step ahead with empirical data, the discrepancy decreased to 1793 from 838016 by a factor of 467 for the PF model compared to the manually calibrated model. The compartmental models can be built to distinguish population groups with different sex, age, and attributes. When combining with the PF algorithm, such detailed models can have better performance in terms of predictive ability comparing to traditional calibration methods [22]. Another advantage of PF with simulation models, as presented in [21], is that they can accommodate multiple lines of data concurrently and produce more accurate and more robust results. This is an enormously powerful feature, especially considering the increasing of available data sources recently. The model does not rely solely on any single data source; multiple data sources can provide events or changes in system states missed by one but captured in another, by using all data sources when calculating the likelihoods for the PF algorithm.

While those works have been carried out with collected static datasets, dynamic models with Particle Filtering and live data streams essentially provides the model with the ability to perform online learning regarding latent state variables. The ability of integrating multiple data sources imposes additional requirements when designing streaming solutions for this kind of applications. When using PF for online real-time inference with high-velocity data sources, the algorithm might fall behind in processing the data because of the significant computational requirement for PF, as each particle in PF is a complete copy of the model representing on hypothesis. In [23] the Real-time Particle Filter (RTPF) uses a mixture of sample sets with each mixture component processing the time points. At one step, it can integrate all points accumulated in this time window to deal with the situation where the data points arrive faster than the consuming rate of the algorithm.

2.2.3 Streaming Processing

Streaming processing is the data-processing configuration focused on continuously handling of live data sources to extract information repeatedly and build knowledge incrementally. The research on streaming processing focuses on methods, algorithms and semantics, hardware infrastructure and software architecture. Applications of streaming data processing include anomaly detection [24], health care and community safety, traffic and crowd control [25], smart city [26], emergency response, natural disaster warning and management [27] and much more. Streaming processing in Big Data, also known as Fast Data or Big Data 2.0 [28] is now a fast-growing research topic. With the volume and velocity of many data sources available in recent years, batch processing methods such as Map-Reduce have clear disadvantages in many use cases compared to streaming processing. Many machine learning and computational statistic methods can naturally progress forward incrementally and are suitable for online algorithms. Those applications such as online clustering, regression and recommendation systems are now becoming standard practices to build responsive, high-performance and reliable solutions. Because of the short expected delay between when data available and when information extracted, it is also referred as real-time analytics [29, 30, 31]. Streaming processing can help us build solutions to retrieve information effectively and rapidly to take advantage of the variety and veracity of the data. The idea of streaming reasoning over the semantic web is proposed in [32], reveals that the foremost challenge is to transfer from handcrafted systems to automatic reasoning over data-streaming efficiently. It is also possible to interactively work with streaming data and do real-time analytics. Data Stream Management Systems (DSMS) [33, 34] are database systems that support executing continuous queries on stream data. A query will be installed and executed on the streaming data for the most up-to-date results until the query is uninstalled. One crucial requirement for modern healthcare is the ability to perform real-time analytics on data collected [35]. However, streaming data is considered strenuous as it is constantly flowing, loosely structured and high-cardinality, where cardinality is defined as the number of unique values [30]: the reasons that it is often loosely structured include: first, the data that is targeted by streaming processing are usually come from variety of sources such as social medias; second, streaming projects are mostly exploring orientated and trying to collect as much data as possible; last, because the data is collected in real-time, properties or values may not be available and often leave no time for manual correction. As pointed out in [36] with the development of computer software and hardware, the ability to do real-time data collection, automated analysis, epidemic recognition, and dissemination of information products for immediate action has become one of the most significant opportunities and also more challenging than before.

Streaming data processing method has been used for individual health care by continuously collecting patient physiological signals and delivering summaries to the corresponding caregivers in real-time[37]. This enables Big data analytics to apply to many categories of healthcare, such as image processing, signal processing and genomics [37]; medical signal processing and health monitoring can benefit significantly from continuous collecting, processing and predicting systems. With the advances in monitoring devices in the health care system, and the rapidly growing of personal health-related wearable and IoT (Internet of Things) devices, such as smartwatches, smart wristbands and capable mobile-phones, taking advantage of all those collected individual patient signals quickly and reliably can dramatically increase the quality-of-life, by providing information and knowledge to make better diagnoses, recommendations, event detection, notification and prediction [38, 39]. Other individual-focused applications including [40] in which widely adopted big data processing methods and machine learning models are now used to predict individual health conditions using streaming health signals. Beyond the individual level, applying streaming processing on real-time public-health data can benefit both the health system and patients, especially when combined with machine learning algorithms or simulation models. Just recently, a framework [41] that incorporates real-time emergency department waiting-time from the UK nation Health Service (NHS), data analytic methods, discrete-event modeling is made available to help patients make better decisions when seeking emergency medical help. It is also mentioned in this work that simulation software such as AnyLogic would be an excellent platform for building and visualizing the simulation part of the framework, with advantages of integrating with the rest of the system and performing GIS (geographic information system) modeling.

Public health data science focuses on population health-related data to detect patterns and risks and provide input for intervention and policy-making. Other types of online data — such as user-posted social media contents, are fascinating by social behaviour data science. Data and information available on the World Wide Web, especially that generated by users of social network platforms, is considered to be well suited for streaming reasoning [42]. Examples include, but are not limited to, Twitter and Facebook; together with other data, those are great for understanding particular epidemiology among populations that traditional monitoring will not match. They are considered to be holding more information for studying public health [43, 44] and can also find the network dynamics of users and how information spread over a certain population [45, 42]. As mentioned in [46], people share symptoms and information on drugs and other medical products, sometimes substantially before they enter the health system for help. That Twitter data can be potentially combined with other sources such as emergency department visiting data and weather data for prediction and resource allocation. The size of those data is increasing quickly, and the amount of information enclosed is invaluable. Those data sources can aid research on disease controlling, intervention designing, new drug experiments, social pattern recognition and more.

Automation is the key for large-scale analytics together with artificial intelligence and machine learning algorithms. The solution we are looking for should be easy to use, ready for different scenarios, and high-performance.

The design of such systems requires many careful considerations. The system needs to be robust, responsive and adapt to new patterns fast. It is a substantial challenge to develop tools that are powerful yet with a highly user-friendly interface[35]. A selected set of existing solutions are compared in table 2.1; some solutions aim for specific problems in their domains while others are flexible to fit many use cases. Usually, a streaming solution would include a range of components to fulfill its functionalities. The components-based solution can offer advantages such that the behaviour and the properties of the entire system can be inferred from the behaviour and properties of the components. As pointed out by [30], the real-time architecture includes, from source to end, components for collection, data flow, processing, storage, and delivery. All components should fulfill three fundamental properties: high availability, low latency, and horizontal scalability. Most solutions use multiple smaller components to handle individual tasks with an API-centric approach [47] for effective communication and integration, and introduce new components quickly to the existing system for maximum flexibility. Common data-mining preprocessing includes data cleaning, transformation, integration, reduction and discretization [48] that should be handled on a per-data-source base, and thus requires specific components to be made. Some configuration files are chosen for easy deployment.

The real-time health care analytic framework presented in [49] is also constructed with individual components: SOMA and Spark streaming libraries are the data harvesting component that gathers and ship the data; Spark ML (machine learning) library acts as the analytic components that support standard machine learning algorithms such as clustering and SVM (support vector machines); Spark GraphX and Cassandra handle live visualization and persistent storage, and Apache Mesos is the cluster manager. Such a setup can overcome many flaws associated with traditional problem-specific solutions, such as hard to scale, long processing delays and security concerns. Though this framework also suffers from several notable limitations such as the overhead of the linked data platform settings and available machine learning algorithms. In [40] they are using the Spark ML decision-tree model to take in user's tweets in real-time, then analyze it and send back a Twitter message to the user using Twitter's API regarding their health status. In [50] ED visiting time data were brought in using open source software, such as FTP (File Transfer Protocol) and Windows Task Scheduler, and then feed into a set of naive Bayesian classifiers to classify them. In [51] a near real-time influenza (H1N1-2009) monitoring and forecasting system was set up during the 2009 epidemic period for Singapore with a public available websites front showing the predicted trends. The system was built using a shell script with *cron* jobs, and the predictions were produced with a stochastic compartmental model. It provided valuable information to the public and institutions. However, the quickly deployed system lacks extensibility, scalability and maintainability. Another similar example is presented in [52]. A highly scalable real-time disease surveillance website system has been built to visually display the harvested and analyzed Twitter tweets related to ILI (Influenza-like Illness) cancer. It shows that those numbers inferred from Twitter match well to later CDC-reported official counts — but the Twitter counts have the advantage of the near real-time speed, which is much valuable sometimes than the CDC reports. EMBERS (Early Model Based Event Recognition using Surrogates) presented in [53] is a large-scale streaming architecture for predictive analytics for societal events and disease outbreaks. The data sources targeted by this system include those that are publicly available such as Twitter, Google Trends, government reports, weather information and HealthMap. The system is divided into component sets, including: data ingest, such as read feeds and convert to fixed format message such as JSON; enrichment, such as tokenization, entity extraction, date normalization and geocoding; analytic modeling, includes surrogate and prediction generation; and finally, selection/delivery.

Each set has many simple independent components, such as those responsible for ingesting individual data sources, tokenizing and tagging texts, or an analytic model; the components are communicated using JSON message over a message queue built on ZeroMQ sockets. It is also notable that the system and all the components and services can be deployed quickly to IaaS (Infrastructure as a Service) platform such as AWS using a single configuration file. A visualization tool enables the users to track the data flow to monitor and make sense of the prediction results. Comparing to their works, ours focuses on time-series time for Bayesian inference instead of natural language processing.

Table 2.1: Data Streaming System Designs

Title	Field and Area	Software and Languages	Visualization Components/Components oriented	High Availability	Containerization	modeling Methods and Machine Learning method
	Architecture Overview					
Big Data Stream Computing in Healthcare Real-Time Analytics [31]	Public Health and Health Care	Apache Storm, Kafka, ZooKeeper, Hadoop, Cassandra	Yes/Yes	Taking advantage of ZooKeeper and built-in HA of Kafka, Hadoop, and other components	Not mentioned	None
	Producer connect to Kafka broker, the use either Storm for streaming processing or Hadoop for batch operations, save to NoSQL Cassandra or HBase, then carry out the analytics from the database, including visualization, decision making, prediction, LOAD, and recommendation					
A Scalable Streaming Big Data Architecture for Real-Time Sentiment Analysis [28]	Sentiment analysis for social media, public opinions about cryptocurrencies and their price	Apache Spark with Spark Streaming, Apache Hive, Apache Hadoop HDFS, Zeppelin	Yes/Yes	Built-in HA of HDFS	Not mentioned	Lexicon, AFINN and Bing as sentiment analysis lexicon

	Spark Streaming connect to Twitter Streaming API as the Data Streaming Layer, input data streaming to Data Processing Layer which consist of Spark and Hive; Data Storage Layer and View Layer are built with HDFS and Zeppelin respectively					
A real-time architecture for detection of diseases using social networks: Design, implementation and evaluation [54]	Public Health	Java, Apache Lucene, MySQL	No/Yes	None	Not men-tioned	SVM
	The System is built with independent Java packages as components; connect to Twitter Streaming API, index, clean, extract keywords and then use classifier to classify the posts					
Streaming Social Media Data Analysis for Events Extraction and Warehousing using Hadoop and Storm: Drug Abuse Case Study [55]	Drug Abuse	Apache Storm, Twitter Streaming API, Hadoop and HDFS	No/Yes	Note mentioned	Not men-tioned	None
	Using Apache Storm to connect to Twitter Streaming API for gathering tweets of the same user, and then send to Hadoop for further linguistic processing and information extraction					
Towards Reliable, Performant Workflows for Streaming-Applications on Cloud Platforms [56]	Scientific Streaming Workflow	RestFlow, Azure Cloud	No/Yes	Yes, custom built VM management	Yes	None
	Using RestFlow to gather workflow requests from end users, and implemented a streaming management system that is responsible for running tasks, maintain and discover available streams and management VMs.					

The simulation models' output can contain a large quantity of hidden information that will not be captured once the models are finished running. Valuable insights can be discovered when looking across many model executions, especially when stochastic values are presented in the models. Also, simulations can be terminated early without waiting for the full results in certain circumstances, but the partial results can be collected and used later. This makes data mining on the output of the simulation model considered equal important [57].

When conducting research using simulation models, it is usually the case that many experiments are needed with a large amount of output from each experiment and then performing systematic analyses of the results after all the output being collected. In [57] the authors believe that streaming systems can act as a feedback loop by integrating online stream mining to simulation results. The online analyzing results can help identify the patterns and schedule additional experiments on the fly, further uncovering the hidden relations, and shortening the iterating times of knowledge discovery.

Another design consideration for building a streaming system is infrastructure support. First, the system and its components need to be able to run on workstations, high-performance clusters and supercomputers, private servers, and cloud providers' infrastructures. The datasets such as social media collections, multimedia resources, genotype data, and data collected from wearable devices can grow beyond the capacity of a single machine and the design needs to be scalable to handle those kinds of workloads potentially when needed. Cloud computing provides a simplified experience for researchers and developers to set up servers, deploy experiments and services, and store and share datasets. Especially, cloud providers make deploying and scaling the system easier by hiding the infrastructure details from users while providing all the core features such as quick scaling, low cost, pay-per-use, high availability. When designing the system, we will need to keep in mind the cloud infrastructure and networking implications.

In [58], the authors describe a cloud computing infrastructure design that can provide the required security, resilience, and data protection for health-related applications. They have built the middleware to support many standard functions such as logging, authentication and offer them as SaaS (Software-as-a-Service). They also employ virtual machines (VM) technologies to isolate individual applications. One notable feature of their platform is that it provides API for the 3rd party apps to access the patient health data while honouring the privacy-preserving requirement. Cloud-native is a set of guidelines and best practices for designing and operating large software applications systems on the cloud; a cloud-native application (CNA) is designed with cloud-first in mind to best profit from the cloud computing infrastructure [59]. A microservices-oriented architecture is an architecture design pattern closely related to cloud-native philosophy. It is the opposite of traditional one-for-all monolithic design and emphasizes quick evolving, self-contained code and library. It compiles with the cloud-native pattern of decomposition of functionality while each component can operate and scale independently and quickly [60].

Virtualization and containerization, such as Docker and Singularity, have gained a tremendous amount of attention and have been widely-used throughout the software development cycles. Containers can offer a unified development environment; they enable easy testing, where error states can be preserved, and spawn up a testing environment quickly. Containerized service deployments are scalable, robust, easy to monitor and upgrade. ZeroVM presented in [61] is a sample solution build on top of container technology that provides a secure workflow for distributed big data analytics that moves isolated containers to data instead of moving data into applications. Containerization and supportive environment are well-supported by the cloud providers, such as Amazon AWS ECS, Microsoft Azure Containers, Google GCP Kubernetes. Those technologies are essential for building microservices and great for automated testing, continuous integration and continuous integration. Doing data analysis and data science in a containerized environment has been documented as well [62]. The technologies can accelerate data and computational workflows in research and help to achieve generating finding results that are findable, accessible, interoperable and reusable (FAIR) [63], without introducing much complexity. Other notable benefits of adopting containerization in data science workflow include:

- Research software environments are easy to communicate and share [64];
- Reproducible results [65]; isolation of variables, such as OS environments, software versions;
- Packing original, intermediate or partially explored data as products and provide a unified way to access data;
- Portable and runs on a wider range of servers, including Infrastructure-as-a-Service (IaaS) and High-performance Computing (HPC) clusters;
- Better performance compared to virtual machines in terms of throughput, response time, and more;
- Can access hardware accelerators such as GPUs;

To summarize, some of the exciting streaming solutions were constructed with subsystems responsible for data ingesting, analyzing, persistent storage and archiving, and some visualization tools to monitor and visualize the results. When designing and implementing streaming processing systems, the same criteria for big data analytics platform evaluation should be considered, including availability, continuity, ease of use, scalability, ability to manipulate at different levels of granularity, privacy and security enablement, and quality assurance, user-friendly and transparent [35, 66]. Virtualization and containerization technologies, microservices- and components-oriented design patterns, cloud computing infrastructures can help achieve those goals. Streaming processing systems are usually built with machine learning and computational statistics methods to extract information and insights from the raw data. Bayesian methods such as PF and Particle MCMC are great candidates for inference with time-series data in the streaming processing systems.

2.2.4 Particle Markov Chain Monte Carlo Methods and Parallelization

Particle Markov Chain Monte Carlo [6] is a Bayesian method of a rising popularity that combines the power of PF and MCMC and aims to provide solutions for joint inference of state and parameter values in even high-dimension complex systems. This sampling-based estimation algorithm uses PF to simplify the likelihood calculation of the MCMC process, enabling it to explore high-dimensional parameter spaces efficiently with time-series data [67], making the MCMC algorithm general and universal. While of relatively recent origin, Particle MCMC has been applied across many areas, including bioinformatics, phylogenetic, hydrologic and environmental modeling and inference, robotics target tracking, and mobile wireless channel tracking [68, 69, 70, 71, 72]. One rising sphere of application of Particle MCMC lies in the estimation of states and parameter values of compartmental models in health, such as SIR transmission models. This was shown to work well with complex systems characterized as including many latent variables, and multi-dimensional observation data [7]. Practical Particle MCMC use requires addressing several challenges, including effective achievement of convergence, tuning parameters and hyperparameters, choice of priors, and challenges in the specification of models. However, a foremost pragmatic issue lies in managing the substantial associated computational burden. Fortunately, because of the modular way in which PF is combined with the MCMC sampling within Particle MCMC, many methods developed to speed up those methods individually can also be readily adapted to Particle MCMC. Past efforts have contributed many innovations to shorten the execution time for PF or MCMC separately or together; some have tried to parallelize such algorithms with multi-core machines, distributed clusters, GPU and FPGAs [73, 74, 75, 76] while others sought to simplify and modify the algorithm so that the computation can be reduced or become easy to parallelize while maintaining the accuracy of the algorithm [77, 78, 79]; selected works are compared in table 2.2.

Some works focus on efficiently running the applications on variations of MCMC, including:

- Population-based Markov Chain Monte Carlo, a combination of evolutionary algorithms (EA) with MCMC as one realization of population-based simulation inference methods [80], where many chains are explored at the same time to mitigate the problem where a single chain can converge poorly [81]. Because of the multi-chain nature of such modifications, it is well suited for GPU parallelization. In [76] GPU implementation of two advanced Monte Carlo methods were presented: Population-based MCMC and SMC. Up to 500-fold improvement was achieved by the GPU version of SMC while 35 for the population-based MCMC. Using custom precision can also bring different performance improvement on all kinds of computation resources while not affecting the sampling quality for certain variations of the population-based MCMC methods [77];

- A parallel implementation of Gibbs sampling for Particle MCMC was presented in [82] by introducing an auxiliary variable and takes advantages of multi-core machines while similar additions can also be used to utilize cluster resources [74];

- Mini batch algorithms when the volume of data is large [83]; in [78] a method was also discussed using subsampling to shorten the running time with large data set, by estimating the log-likelihood of each MCMC iteration using a subset of m observations which is far smaller than the total observation n and thus can speed up the overall algorithm.

Table 2.2: Monte Carlo Methods Accelerating Strategies

Name	Properties		Solution & Results	
Parallel Markov Chain Monte Carlo Simulation by Pre-Fetching [79]	Target Method	MCMC	Solution	Pre-fetching: Using one MCMC chain but instead of testing one step forward, spread out and move forward several steps so that each will use one thread/machine
	Parallel and Distributed	Yes		
	On Special Hardware	No	Result	Theoretical speed up log2(P) where P is the number of processors. Experiments shows speed-up decreases as number of processors increase.
Mini-batch Tempered MCMC [83]	Target Method	MCMC	Solution	Mini-batch Tempered: using a sampled subset of the data to estimated the log-likelihood to simplify the calculation of likelihood. Combined a novel modified parallel sampler to efficiently sample from complex posteriors.
	Parallel and Distributed	No		
	On Special Hardware	No	Result	The mini-batch tempered MCMC has been shown working efficiently despite the subsampling of data, and when combine with Equi-Energy sampler, the time complicity reduced to O(Nn) from $O(K^2Ns)$, where K is the number of parallel chains.

- Each Particle Filter step depends on the previous step, while each MCMC iteration also depends on the previous step. Much research also focuses on exploring the possibility of using multiple Markov Chains so that they can run on many machines; experiments show that for certain tasks, the statistical properties are close between the combination of several chains compared to one long MCMC chain [84];
- For the ordinary Particle MCMC algorithm, [75] presents an example about utilizing GPU acceleration. However, the propagation is only a simple equation that only takes minimal computation. Also, [73] shows us how we can define hardware to suit the computational demand for such methods.

Particle MCMC can be used together with System Dynamics models similar to the PF applications and has been shown to work well with complex systems, several latent variables, and multi-dimension observation data [7]. While doing System Dynamics modeling with Particle MCMC enables us to estimate the parameters reliably and automatically, compared to manual calibration, System Dynamics modeling itself demands tremendous amount of computation because of the integration of ordinary differential equations, especially when the integration step is small for accurate simulations.

The primary motivation of the work presented in this thesis is to accelerate the algorithm running speed of Particle MCMC with complex System Dynamics compartmental models built for public health and social well-being research. This paper sought to accelerate the use of Particle MCMC with complex SD models by migrating the algorithm to special hardware without changing the mathematical structure of the Particle MCMC algorithm. Without changing the Particle MCMC algorithm, we focus on adopting the algorithm to special hardware. We have selected CUDA GPU as the target device because of the performance potential, availability of the hardware, community support of the programming language and development kit and extensibility. GPGPU (General-purpose Graphic Processing Units) and CUDA (Compute Unified Device Architecture) have been used for accelerating machine learning and computational statistics methods and have been shown very effective. For example, deep learning has been benefiting from GPU during training, especially in distributed configurations, making it possible for fast learning with large training data as well as easier parameter tuning [85, 86].

Adopting an algorithm to GPU is complex. Although the GPU structure allows to spawn thousands of threads at the same time, the memory and bandwidth limits mean that the efficiency is challenging to achieve at the maximum when the individual tasks are two large, such as for reconstructing phylogenetic trees [68], where the tasks need to be further divided which resulting in even greater programming difficulties on GPUs. On the other hand, the processing power of GPUs is increasing quickly each generation. It would be valuable to examine the performance characteristics of particle MCMC on the current hardware to envision the future of parallelization of such algorithms.

Another line of research work surrounding MCMC is focusing on algorithms and testing methods to examine the sampler's performance in terms of state and parameter inference. One of the most common methods used is to verify the convergence of MCMC chains when multiple chains are running simultaneously. If they converge or mix well, it will indicate that the experiment has reached a credible conclusion, and thus no further execution will be needed. Specifically, Effective Sample Size (ESS) is one way to measure the performance of a single chain [87], while Gelman-Robin shrink factors test how multiple chains mixed to see if they all arrive at the same stationary distribution [88, 89]. If modification or improvement has been applied to the MCMC process, using those diagnostic methods can measure the effects of the changes [90, 91]. In another words, if we would like to examine the effect of changing the data input size on MCMC inference, we can measure the ESS and shrink factors when adding time points to a different MCMC run.

2.3 Algorithms Overview

2.3.1 Bayesian Inference

Two main interpretations of probabilities include Frequentist, where probability is defined as its relative frequency over unlimited trials, and Bayesian, where it is defined as belief or expectation of the event [92]. Under the Bayesian interpretation, to obtain the conditional possibility or expectation of an event, an initial or prior expectation will be updated as new data or evidence becomes available and finally transfer to a posterior probability which is the conditional belief given all the evidence. This provides us with the basic framework to perform Bayesian inference. In public and social data science, this is often the case where we have some historical time-series data, and we are interested in the parameters underneath that describe the system and the process.

When the system is simple such as those that parameters can be described using linear and Gaussian distributions, the update of prior distribution to posterior distribution is straightforward: we only need to multiply the probability density functions of prior and likelihood to get the probability density function of the posterior:

$$P(\Theta|data) \propto P(data|\Theta) \times P(\Theta) \tag{2.1}$$

More often, however, the system we try to model is very complex, and the posterior distribution is nonlinear and non-Gaussian. Thus a closed-form parametric expression does not exist for carrying out the posterior distribution calculation [93].

2.3.2 Recursive Bayesian Estimation

In the context of signal processing and filtering problems, state-space models are used to describe and analyze the probability dependency between the latent state variables and the observed measurement [94]. The system itself can be stochastic, which means it is subject to noise and is non-deterministic; the observing process is also stochastic and affected by measurement noise. To use recursive Bayesian estimation to infer the state x_t at time t given the observations $z_{1:t}$, $P(x_t|z_{1:t})$, in a state-space model with non-observable, or latent true state, two functions (models) of the underlying system will be assumed to be known. Assuming the process follows the Markov assumption, where the current state only depends on the previous state but not states before that; now with x_t as the hidden state at time t, and use z_t as the measurement we would get at time t, then the function g

$$x_t = g(x_{t-1}, \epsilon_t) \tag{2.2}$$

describes how the system would evolve from the previous state to a new state, with a certain noise ϵ_t; this is known as the transition model. The information describing the hidden states, also known as the measurement, can be calculated based on the states:

$$z_t = h(x_t, \delta_t) \tag{2.3}$$

However, as before, this function h also involves measurement noise δ_t. This is known as the observation model, or the sensor model as it is usually related to sensor reading. When both the functions are linear and the noises are independent and following Gaussian distribution then this becomes a linear-Gaussian model.

The recursive estimation refers to the formulation of calculating the posteriors recursively using the results from one step before. To calculate the posterior belief at time step t, we need to have the observation at time t, as well as the posterior of $t-1$ which can be calculated using posteriors of $t-2$ and so on. We could also think of using recursive Bayesian estimation to solve tracking problems as a method of induction. At each time step t, it will use the belief from the previous step, make a prediction using the transition model, and make this the prior; then gather the measurement of the current step, compute the likelihood of the measurement using the prior and make this the posterior distribution, which is the new and corrected belief of this step; this is now ready to use for the next step.

The algorithm can be summarized as follows [95, 4]:

- the prediction step: at time t, given from the previous step

$$P(x_{t-1}|z_{1:t-1}) \tag{2.4}$$

  need to compute

$$P(x_t|z_{1:t-1}) \tag{2.5}$$

  Using the law of total probability, we get this equal to

$$\int P(x_t, x_{t-1}|z_{1:t-1})dx_{t-1} \tag{2.6}$$

  Then, using conditional probability

$$\int P(x_t|x_{t-1}, z_{1:t-1})P(x_{t-1}|z_{1:t-1})dx_{t-1} \tag{2.7}$$

  Here, because our Markovian assumption, x_t should only depend on x_{t-1}

$$\int P(x_t|x_{t-1})P(x_{t-1}|z_{1:t-1})dx_{t-1} \tag{2.8}$$

  Where $P(x_t|x_{t-1})$ is the transition model mentioned above;

- the update, or correction step: now we have $P(x_t|z_{1:t-1})$, and the new measurement for this step, z_t, we need to compute the new belief $P(x_t|z_{1:t})$ by using the likelihood of the predicted state distribution given the new measurement:

$$P(x_t|z_{1:t}) \tag{2.9}$$

  using Bayes' rule, this equals to

$$\frac{P(z_t|x_t, z_{1:t-1})P(x_t|z_{1:t-1})}{P(z_t|z_{1:t-1})} \tag{2.10}$$

  because of the measurement model, we know that z_t solely depends on the current state

$$P(x_t|z_{1:t}) = \frac{P(z_t|x_t)P(x_t|z_{1:t-1})}{P(z_t|z_{1:t-1})} \propto P(z_t|x_t)P(x_t|z_{1:t-1}) \tag{2.11}$$

  that is, this new posterior is proportional to observation model and predicted estimation.

2.3.3 Monte Carlo Methods and Particle Filtering

Monte Carlo is a set of methods using random samples to approximate the underlying distribution. A simple example would be drawing random points from a plane to estimate the area under the curve. Implementations of Monte Carlo methods are used to estimate posterior distributions where direct calculations are tricky. The core idea is that by generating enough samples, one can approximate the posterior distribution; the challenging part is how to generate samples efficiently. Particle Filter and MCMC are methods to generate samples. Particle Filter is a non-iterative method, while MCMC is an iterative method that produces independent samples [14]. Particle Filtering is one efficient approach for doing recursive Bayesian estimation by using many particles to simulate the possible outcomes at each step. It is perfect for inference in a nonlinear state-space model with continuous time and state space.

Particle Filter follows the predict-update loop, but approximates the prior and posterior distributions using a set of particles. The density of the distributions will be represented by the the location (values) and the weight of the particles. This can be achieved as this follows the method of importance sampling; that is, we can approximate a distribution $p(x)$, by sampling n samples x^i from a **proposal** distribution, $q(x)$, then multiply by the (normalized) coefficients, or, importance weights distribution $w(x)$, if drawing from the proposal distribution is much easier than drawing from original distribution; the coefficients' importance weights distribution is defined as

$$w(x) = \frac{p(x)}{q(x)} \tag{2.12}$$

and then to approximate the original distribution

$$p(x) \approx \sum_{i=1}^{n} w(x^i)\delta_{x^i}x \tag{2.13}$$

where $\delta_{x^i}x$ is the Dirac delta mass at x^i. With this, instead of sampling from the true distribution $p(x^i_{0:k}|z_{1:k})$, we can simply from a proposal distribution $q(x^i_{0:k}|z_{1:k})$ and set the weights according to

$$w^i_t \propto \frac{p(x^i_{0:k}|z_{1:k})}{q(x^i_{0:k}|z_{1:k})} \tag{2.14}$$

Since:

$$\begin{aligned}
p(x^i_{0:k}|z_{0:k}) &= p(x^i_{0:k}|z_k, z_{0:k-1}) \\
&= \frac{p(x^i_{0:k}, z_k|z_{0:k-1})}{p(z_k|z_{0:k-1})} \\
&= \frac{p(z_k|x^i_{0:k}|z_{0:k-1})p(x^i_{0:k}|z_{1:k-1})}{p(z_k|z_{1:k-1})} \\
&= \frac{p(z_k|x^i_{0:k}|z_{0:k-1})p(x^i_k|x^i_{0:k-1}|z_{1:k-1})p(x^i_{0:k-1}|z_{1:k-1})}{p(z_k|z_{1:k-1})} \\
&= \frac{p(z_k|x^i_k)p(x^i_k|x^i_{k-1})p(x^i_{0:k-1}|z_{1:k-1})}{p(z_k|z_{1:k-1})} \\
&\propto p(z_k|x^i_k)p(x^i_k|x^i_{k-1})p(x^i_{0:k-1}|z_{1:k-1})
\end{aligned} \tag{2.15}$$

and:

$$q(x^i_{0:k}|z_{1:k}) = q(x^i_k|x^i_{0x:k-1}|z_{1:k})q(x^i_{0:k-1}|z_{1:k-1}) \tag{2.16}$$

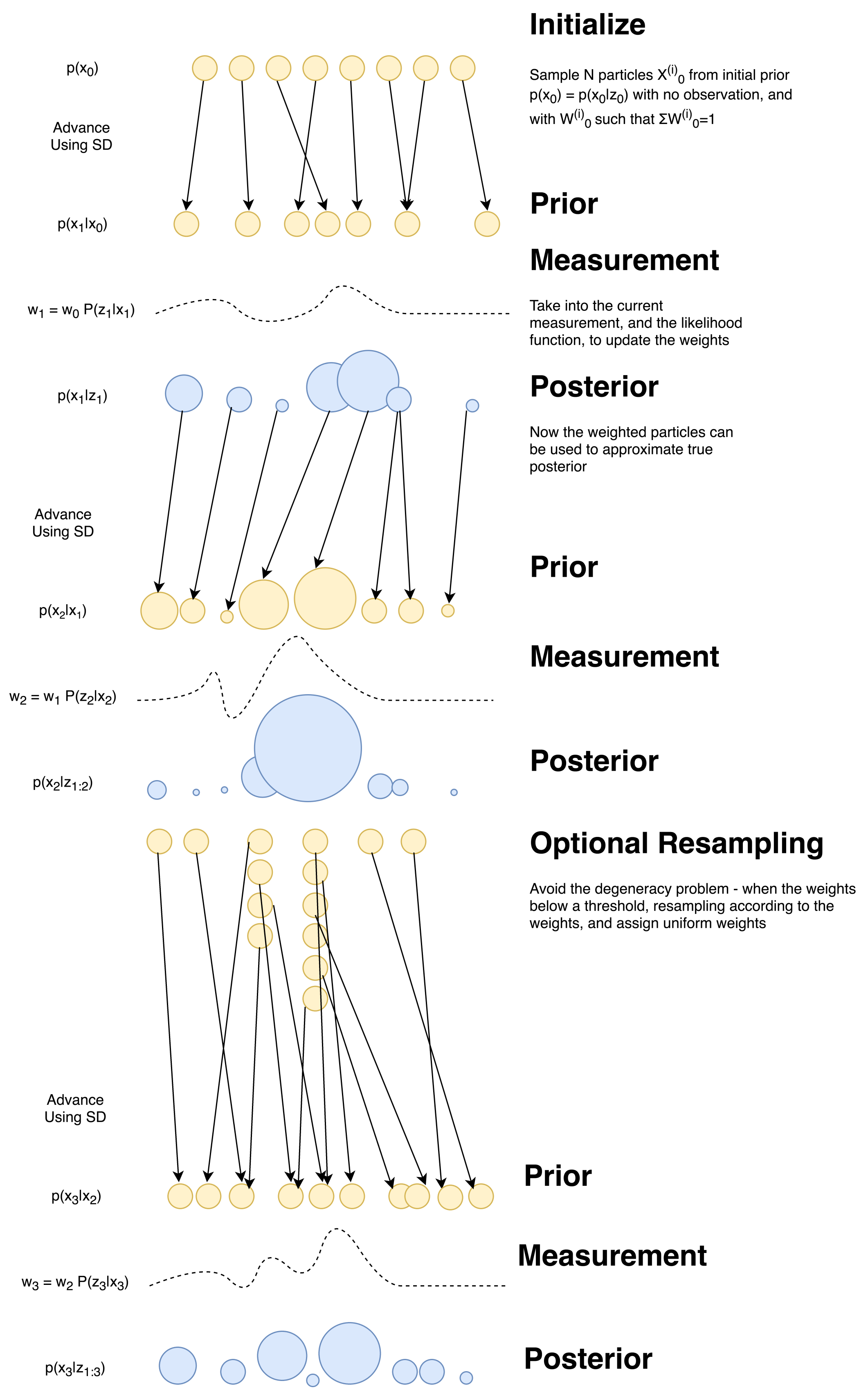

Initialize
$p(x_0)$
Sample N particles $X^{(i)}_0$ from initial prior $p(x_0) = p(x_0|z_0)$ with no observation, and with $W^{(i)}_0$ such that $\Sigma W^{(i)}_0=1$
Advance Using SD
Prior
$p(x_1|x_0)$
Measurement
$w_1 = w_0\, P(z_1|x_1)$
Take into the current measurement, and the likelihood function, to update the weights
Posterior
$p(x_1|z_1)$
Now the weighted particles can be used to approximate true posterior
Advance Using SD
Prior
$p(x_2|x_1)$
Measurement
$w_2 = w_1\, P(z_2|x_2)$
Posterior
$p(x_2|z_{1:2})$
Optional Resampling
Avoid the degeneracy problem - when the weights below a threshold, resampling according to the weights, and assign uniform weights
Advance Using SD
Prior
$p(x_3|x_2)$
Measurement
$w_3 = w_2\, P(z_3|x_3)$
Posterior
$p(x_3|z_{1:3})$

Figure 2.1: Particle Filter Algorithm Visualized

Thus:

$$
\begin{aligned}
w_t^i &\propto \frac{p(z_k|x_k^i)p(x_k^i|x_{k-1}^i)p(x_{0:k-1}^i|z_{1:k-1})}{q(x_k^i|x_{k-1}^i|z_k)q(x_{0:k-1}^i|z_{1:k-1})} \\
&= w_{t-1}^i \frac{p(z_k|x_k^i)p(x_k^i|x_{k-1}^i)}{q(x_k^i|x_{k-1}^i|z_k)}
\end{aligned}
\tag{2.17}
$$

This tells us that the new weights will be related to the previous weights, the likelihood function, the prior and the proposal distribution, which are all known at each step. Choosing the optimal proposal distribution is critical for the performance of PF; one common solution is to sample from the prior; that is, $q(x_k^i|x_{k-1}^i|z_k) = p(x_k^i|x_{k-1}^i)$ so that the weights update rule can be simplified to

$$
w_t^i \propto w_{t-1}^i \times p(z_k|x_k^i) \tag{2.18}
$$

Applying this with the predict-update loop, we get the basic structure of PF, which is in the form of Sequential Importance Sampling (SIS). The degeneracy problem of PF describing the situation where the majority of the weights are concentrated on a very small subset of particles as t increases and results in a poor approximation using the particles and the weights. The algorithm can be summarized as follows:

- Initialize a set of n instances of the model, $X_0^{(i)}$ using the assumed initial prior, $p(x_0)$, with equal weights $W_0^{(i)}$;
- Run the model for N_{train} iterations on the empirical data. That is, for $t \in [1, N_{train}]$, do:
  - Advance each particle using their own model and own parameters;
  - Then, taking into the current measurement (the current time step's value from the empirical dataset) and our selected observation model, to assign a new weight to this each particle using (2.18); the weights and particles can then be an approximation of $p(x_t|z_{1:t})$;
  - Sum up the individual weights and normalize them such that they add up to 1;
  - To avoid the degeneracy problem, we will also monitor the effective sample size:

    $$
    \hat{S}_{eff} = \frac{1}{\sum (w_t^i)^2} \tag{2.19}
    $$

    and if $\hat{S}_{eff} < S_{threshold}$, we will do resampling based on weights, but assign them new equal weights; resampling will be performed using systematic resampling algorithm so that the sampling time is $O(N)$.
- After the training process, will forecast N_{test} steps with the current particles to evaluate the model:
  - Let each particle advances step-by-step as before, but without the sampling and correction;
  - At each step, compare the overall prediction result using sampling with the empirical data points, and calculate the discrepancy using root-mean-square error (RMSE) method to evaluate the model.

This process is visualized in Figure 2.1.

2.3.4 Markov Chain Monte Carlo

Markov Chain Monte Carlo uses a Markov Chain to do random walks for Monte Carlo experiments. A Markov Chain is a random process where the sequence of the process is generated such that the future state will only depend on the current state but not influenced by the past states. They are many variations in terms of sampling methods. We will use the Metropolis-Hastings sampler as described in [96]. The key is that it is a sequential algorithm with each step depending on the previous because of the Metropolis-Hastings sampler. Given the current step as x and next step as x', a transition probability $P\left(x'|x_n\right)$ for the Markov process is needed, such that the final stationary distribution $\pi\left(x\right)$ can produce $\pi\left(x\right) = P\left(x\right)$. Thus, the transition probabilities we are looking for need to satisfy:

$$P\left(x'|x\right)P(x) = P\left(x|x'\right)P\left(x'\right) \tag{2.20}$$

Thus:

$$\frac{P\left(x'|x\right)}{P\left(x|x'\right)} = \frac{P\left(x'\right)}{P(x)} \tag{2.21}$$

To design the transition probabilities for the Markov process, we will divide it into two sub-steps. It will propose a x' given x, using the proposal distribution $g\left(x'|x\right)$; note that sometime this is denoted as $q\left(\right)$ instead of $g\left(\right)$. Then, this will be either accepted or rejected based on the acceptance ratio $A\left(x', x\right)$. Then:

$$P\left(x'|x\right) = g\left(x'|x\right)A\left(x', x\right) \tag{2.22}$$

$$A\left(x', x\right) = \frac{P\left(x'|x\right)}{g\left(x'|x\right)} \tag{2.23}$$

$$\frac{A\left(x', x\right)}{A\left(x, x'\right)} = \frac{\frac{P\left(x'|x\right)}{g\left(x'|x\right)}}{\frac{P\left(x|x'\right)}{g\left(x|x'\right)}} = \frac{P\left(x'|x\right)g\left(x|\,x'\right)}{P\left(x|\,x'\right)g\left(x'|x\right)} = \frac{P\left(x'\right)}{P(x)}\frac{g\left(x|x'\right)}{g\left(x'|x\right)} \tag{2.24}$$

The sampler accepts a sample using acceptance ratio $A\left(x_n, x_{n-1}\right)$, and it is defined as

$$A\left(x', x\right) = \min\left(1, \frac{P\left(x'\right)}{P(x)}\frac{g\left(x|x'\right)}{g\left(x'|x\right)}\right) \tag{2.25}$$

Thus, when $P\left(x'\right)g\left(x|x'\right) > P(x)g\left(x'|x\right)$, $A\left(x', x\right) = 1$ and $A\left(x, x'\right) = \frac{P(x)}{P(x')}\frac{g\left(x'|x\right)}{g\left(x|x'\right)}$, which have $\frac{A\left(x',x\right)}{A(x,x')} = \frac{P\left(x'\right)}{P(x)}\frac{g\left(x|x'\right)}{g\left(x'|x\right)}$, and vice verse. So if we follow this acceptance ratio to jump or stay, the final sample sets can be used to estimate the true posterior after many iterations. When the likelihood can be computed quickly, such as in [97], the MCMC algorithm can efficiently estimate parameters and the underlying states.

2.3.5 Particle Markov Chain Monte Carlo Methods

Particle MCMC uses PF to simplify the likelihood calculation of the MCMC process, enabling it to explore high-dimensional parameters space efficiently with time-series data [67], making the MCMC algorithm general and universal. Continuing from the previous section's MCMC illustration, in order to calculate:

$$\frac{p\left(x'\right)}{p(x)} \frac{g\left(x|x'\right)}{g\left(x'|x\right)} \tag{2.26}$$

when we want to sample the true state $x_{1:t}$ and the parameter vector θ given the observations $y_{1:t}$, namely from $p\left(\theta, x_{1:t}|y_{1:t}\right)$, one can replace the x from 2.26 with $\theta, x_{1:t}|y_{1:t}$:

$$\begin{aligned} &\frac{p\left(\theta', x'_{1:t}|y_{1:t}\right)}{p(\theta, x_{1:t}|y_{1:t})} \frac{q\{(\theta, x_{1:t}|y_{1:t}) \mid (\theta', x'_{1:t}|y_{1:t})\}}{q\{(\theta', x'_{1:t}|y_{1:t}) \mid (\theta, x_{1:t}|y_{1:t})\}} \\ &= \frac{p\left(\theta', x'_{1:t}|y_{1:t}\right)}{p(\theta, x_{1:t}|y_{1:t})} \frac{q\{(\theta, x_{1:t}) \mid (\theta', x'_{1:t})\}}{q\{(\theta', x'_{1:t}) \mid (\theta, x_{1:t})\}} \\ &= \frac{p\left(\theta'|y_{1:t}\right) p_{\theta'}\left(x'_{1:t}|y_{1:t}\right)}{p\left(\theta|y_{1:t}\right) p_{\theta}\left(x_{1:t}|y_{1:t}\right)} \frac{q\{(\theta, x_{1:t}) \mid (\theta', x'_{1:t})\}}{q\{(\theta', x'_{1:t}) \mid (\theta, x_{1:t})\}} \end{aligned} \tag{2.27}$$

Thus, as per [6], the proposal density for new candidates can be chosen as follows:

$$q\{(\theta', x'_{1:t}) \mid (\theta, x_{1:t})\} = q\left(\theta'|\theta\right) p_{\theta'}\left(x'_{1:t}|y_{1:t}\right) \tag{2.28}$$

Then, 2.27 becomes:

$$\begin{aligned} &\frac{p\left(\theta'|y_{1:t}\right) p_{\theta'}\left(x'_{1:t}|y_{1:t}\right)}{p\left(\theta|y_{1:t}\right) p_{\theta}\left(x_{1:t}|y_{1:t}\right)} \frac{q\left(\theta|\theta'\right) p_{\theta}\left(x_{1:t}|y_{1:t}\right)}{q\left(\theta'|\theta\right) p_{\theta'}\left(x'_{1:t}|y_{1:t}\right)} \\ &= \frac{p\left(\theta'|y_{1:t}\right)}{p\left(\theta|y_{1:t}\right)} \frac{q\left(\theta|\theta'\right)}{q\left(\theta'|\theta\right)} \end{aligned} \tag{2.29}$$

Using Bayes' law:

$$p\left(\theta|y_{1:t}\right) = \frac{p\left(y_{1:t}|\theta\right) p\left(\theta\right)}{p\left(y_{1:t}\right)} = \frac{p_{\theta}\left(y_{1:t}\right) p\left(\theta\right)}{p\left(y_{1:t}\right)} \tag{2.30}$$

And now the acceptance rate equation 2.29 becomes:

$$\frac{p'_{\theta}\left(y_{1:t}\right) p\left(\theta'\right)}{p_{\theta}\left(y_{1:t}\right) p\left(\theta\right)} \frac{q\left(\theta|\theta'\right)}{q\left(\theta'|\theta\right)} \tag{2.31}$$

Generally speaking, it is usually hard to compute $P_{\theta}(y_{1:t})$ in high dimensional systems. We use Particle Filtering to target the proposal distribution $p_{\theta'}\left(x'_{1:t}|y_{1:t}\right)$ as in 2.28, and sample one complete trajectory of a particle, $X_{1:t}^{*}$ as one approximate sample from $p_{\theta'}\left(x'_{1:t}|y_{1:t}\right)$, and then can get the marginal likelihood $P_{\theta}(y_{1:t})$ from [6]:

$$\begin{aligned}\hat{p}_{\theta}\left(y_{1:T}\right) &:= \hat{p}_{\theta}\left(y_{1}\right)\prod_{n=2}^{T}\hat{p}_{\theta}\left(y_{n}|y_{1:n-1}\right)\\ \hat{p}_{\theta}\left(y_{n}|y_{1:n-1}\right) &= \frac{1}{N}\sum_{k=1}^{N}w_{n}\left(X_{1:n}^{k}\right)\end{aligned} \tag{2.32}$$

In other words, we can get the products of averaged unnormalized weights at each time step using Particle Filtering to get the marginal likelihood and then use that to calculate the acceptance ratio. This allows us to combine compartmental simulations with MCMC since we already know that those models work well with the Particle Filter algorithm. The difference between PIMH (particle independent Metropolis–Hastings) and PMMH (particle marginal Metropolis–Hastings) is that the parameter θ is known in PIMH and unknown in PMMH; thus, it is the joint distribution that got sampled from in PMMH. The final form of the acceptance ratio with the estimation from Particle Filter is:

$$1\wedge\frac{\hat{p}_{\theta^{*}}\left(y_{1:T}\right)p\left(\theta^{*}\right)}{\hat{p}_{\theta(i-1)}\left(y_{1:T}\right)p\{\theta(i-1)\}}\frac{q\left\{\theta(i-1)|\theta^{*}\right\}}{q\left\{\theta^{*}|\theta(i-1)\right\}} \tag{2.33}$$

and it is shown in [6] that this estimation is converging to the true value in 2.31 as the number of particles $N\rightarrow\infty$. This process is visualized in Figure 2.2.

2.3.6 System Dynamics Models

System Dynamics modeling is a quantitative compartmental modeling method focusing on and a perspective shaped by feedbacks and accumulations within the system. In the health sciences, System Dynamics — and its cognate relatives also using compartmental modeling — is commonly used to simulate the evolution of the health status a population over a period of time. The state of the model at any one time is summarized by a set of stocks (compartments representing accumulations), each of which represents the population currently with a certain combination of characteristic, such as being in a specific health state. The value of each such stock (compartment) at that time counts the number of people within that state at that time. One example is the SEIR (Susceptible-Exposed-Infectious-Recovered) model where individual of a population can move between stocks representing the susceptible (S), exposed (E, representing latently infected infected), infectious (I) and recovered (R) populations, as shown in Figure 2.3. Stocks start with a specified initial value; thereafter, the change in the value of the stock is dictated by the in- and out-flows for that stock, with the rate of the change of the stock at any one time being given by the net inflow at that time. At any one time, the values of each stocks can be measured, and a vector of such values uniquely specifies the state of the system at that time.

Changes in the state of the system (here, the health status of the population) over time is characterized by flows representing the movement between stocks, from outside the model into a stock or from a stock

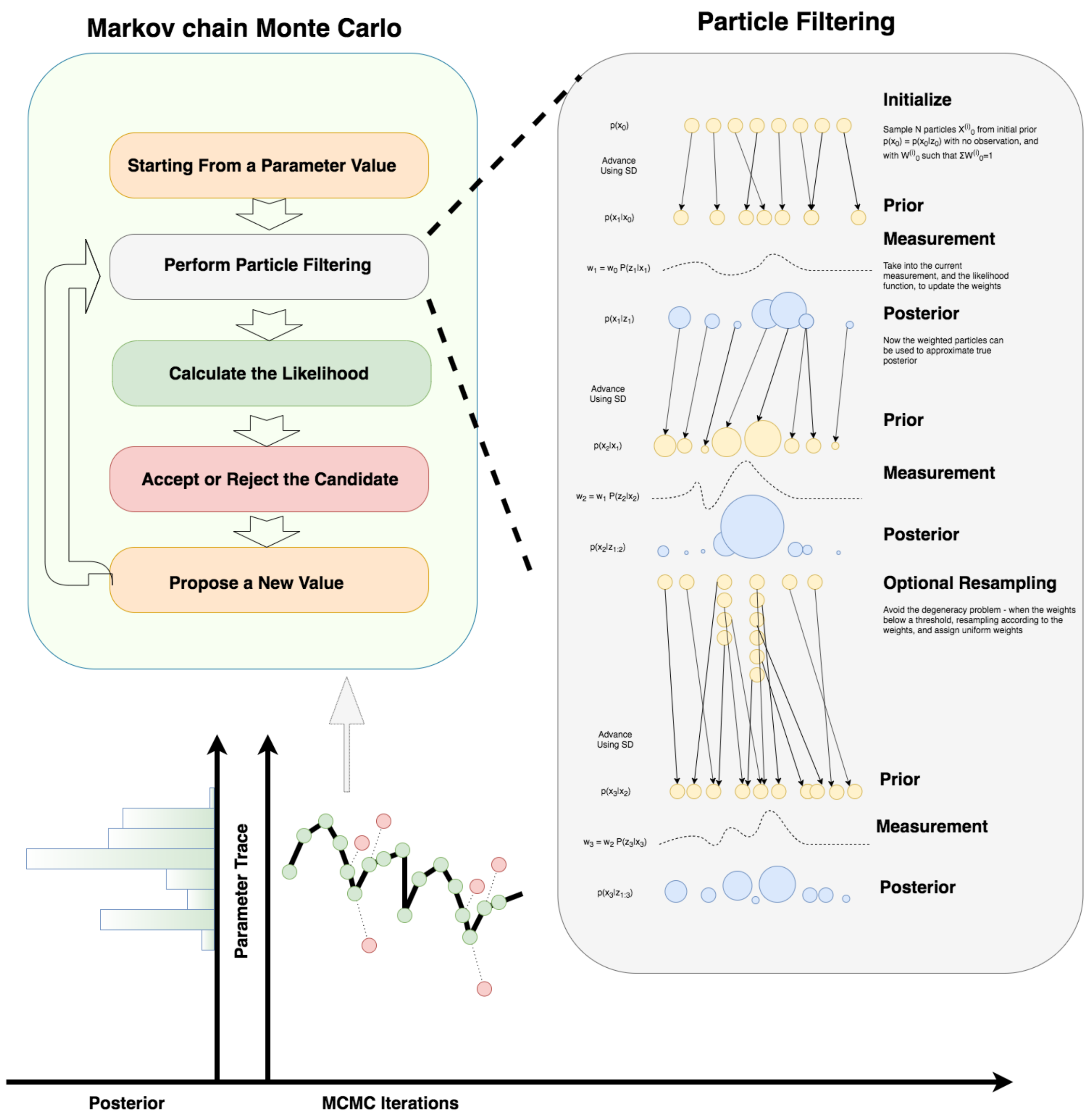

Markov chain Monte Carlo
Starting From a Parameter Value
Perform Particle Filtering
Calculate the Likelihood
Accept or Reject the Candidate
Propose a New Value
Particle Filtering
Initialize
$p(x_0)$
Sample N particles $X^{(i)}_0$ from initial prior $p(x_0) = p(x_0|z_0)$ with no observation, and with $W^{(i)}_0$ such that $\Sigma W^{(i)}_0=1$
Advance Using SD
$p(x_1|x_0)$
Prior
Measurement
$w_1 = w_0 P(z_1|x_1)$
Take into the current measurement, and the likelihood function, to update the weights
$p(x_1|z_1)$
Posterior
Now the weighted particles can be used to approximate true posterior
Advance Using SD
$p(x_2|x_1)$
Prior
Measurement
$w_2 = w_1 P(z_2|x_2)$
$p(x_2|z_{1:2})$
Posterior
Optional Resampling
Avoid the degeneracy problem - when the weights below a threshold, resampling according to the weights, and assign uniform weights
Advance Using SD
$p(x_3|x_2)$
Prior
Measurement
$w_3 = w_2 P(z_3|x_3)$
$p(x_3|z_{1:3})$
Posterior
Parameter Trace
Posterior
MCMC Iterations

Figure 2.2: Particle Markov Chain Monte Carlo Algorithm. Note that PF is providing the marginal likelihood to MCMC since the analytical one does not exist otherwise.

out of the model. Flows will be expressed with respect to some time unit, with that choice — for example, per day, per week, per year — depending on modeler convenience. Those models are nonlinear by nature. While system state governs the evolution of that state, parameters moderate the behaviour of such models. Independent parameters represent the exogenous values in the model, and generally represent assumptions imposed on the model, and are the opposite of the values generated by the model, which are categorized as endogenous quantities. It is worth pointing out that SEIR or the variations of such compartmental models obey the Markovian property: the next state of the system is solely depending on the current state and evolving continuously. Whether characterized with the feedback- and accumulation- specific perspectives and modeling philosophy of System Dynamics or more broadly employing compartmental structure, such models can usually[1] be summarized as a set of ordinary differential equations.

Equations describing the evolving of the stock values are shown below:

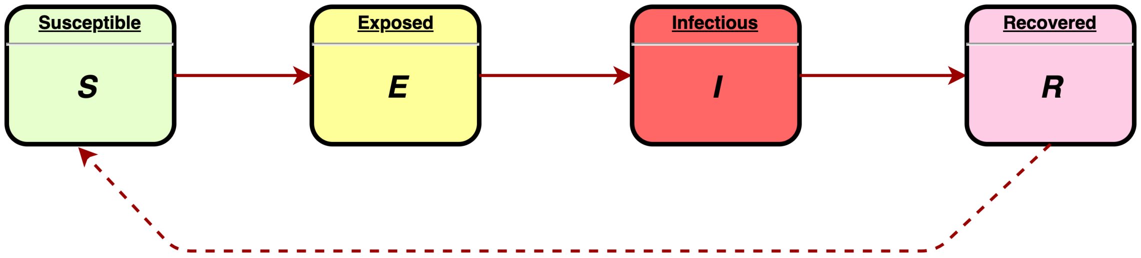


Figure 2.3: Stock-and-Flow Diagram of SEIR System Dynamics Model

$$\frac{dS}{dt} = \beta \frac{I}{N} S \tag{2.34}$$

$$\frac{dE}{dt} = \beta \frac{I}{N} S - \sigma E \tag{2.35}$$

$$\frac{dI}{dt} = \sigma E - \gamma I \tag{2.36}$$

$$\frac{dR}{dt} = \gamma I \tag{2.37}$$

Where $N = S + E + I + R$ is the total population and $\frac{dS}{dt} + \frac{dE}{dt} + \frac{dI}{dt} + \frac{dR}{dt} = 0$ because we assume a closed population in which death and birth lie outside the scope of the model. Parameters such β here represent the probability of infection transmission given exposure of a susceptible to an infective.

In this work we are interested in the compartmental model in epidemiology that is usually represented by System Dynamics systems. There are two kinds of predictions when using those models: projection and forecasting[98]. Projection is used to test assumptions and ask 'what-if' questions with the models,

[1]In some cases, aspects of model formulation — such as the presence of stochastics — may require alternative mathematical expression related to ordinary differential equations, such as stochastic differential equations.

while forecasting is specifically focusing on making predictions of what will happen under current real-work circumstances. System Dynamics models can accommodate both projections [99] and forecasting [100].

The stocks in those models are considered as the indications for providing understandings and making choices. In the context of infectious diseases and outbreak response, we would like to have highly grounded model based on empirical evidence quickly to make accurate projection and forecasting. We use PF or Particle MCMC methods to estimate parameters and sampling trajectories of states to overcome the shortcomings of traditional calibrations needed to ground those models; not only they can generate better and more accurate results but also have the ability to continuously perform this task of re-grounding the model automatically [1, 22].

We can view such systems as state-space models with hidden states. The system evolves forward as defined by the model. Each particle in PF or Particle MCMC has a full copy of the model state and evolves forward independently to the next observation time point. Stochastics are added to the model so that particles will diverge; the particles will also sample values from the distributions of exdogenous parameters. At the observation time point, empirical data can then be used to measure and evaluate the particles based on their states using the formulation of PF as described earlier.

2.4 GPU Programming and CUDA

2.4.1 Hardware Description

Graphic Processing Units (GPUs) are dedicated specialized computation devices responsible for efficiently handling tasks related to graphics and user interfaces in modern personal computer and mobile devices. The demanding for powerful GPUs raised with the growing market of the video gaming and high definition multimedia playback and streaming. Even though GPUs were not built for general-purpose computations initially, GPU characteristics make it perfect for those non-sequential high-throughput applications in servers. They can effectively accelerate computational-intensive algorithms such as machine learning and computational statistics, computer vision, image processing tasks and simulations. For example, Deep Learning has benefited from sizeable parallel hardware such as GPUs for fast learning, processing, interpreting and parameter tuning.

Compute Unified Device Architecture (CUDA) is a general-purpose heterogeneous programming language model for both graphic and non-graphic calculations on easily accessible NVIDIA GPUs.

There are many differences between CPU and GPU hardware that make GPU programming unique. Understanding the architectural difference between CPU and GPU is essential and can help us decompose the computation problem into parallel solutions and map it to the hardware to take advantage of the architectural design of those highly parallel devices. Programming with GPUs differs from multi-threaded CPU programming. The dramatically distinct core counts and features between the host CPU and GPU devices require a different programming model.

Many optimizations for host CPUs focus on latency when completing tasks, while GPUs are optimized for throughput, reflective of their purposeful design for rendering images and videos, where the computing time for individual pixels is crucial for finishing the full image containing thousands of pixels. GPUs have many simple processing cores that support single-instruction-multiple-data (SIMD) semantics. To accommodate I/O needs, GPUs are usually equipped with high-bandwidth, high-clock speed and large memory right on the chip to address the increasing demand for high graphic resolution and frame rates in recent years. GPU structure allows programmers to spawn thousands of threads simultaneously. The memory hierarchy and management are also distinct. Instead of the notion of stack, heap, and cache, GPUs have global, constant, shared memory, etc., each with distinct performance profiles; poor usage can severely impair performance. CUDA virtualizes the physical hardware, such that a thread is a virtualized scalar processor, with registers, program counter and state, and a block is a virtualized multiprocessor with shared memory.

Overall, modern CPUs are designed with multiple cores on a single chip or CPU die, with a large number of cache storage, usually divided into several levels. GPUs are designed differently, switching the focus from cache to core, even though the cores are simpler compared to CPU cores. There are distinct memory and cache spaces, but compared to CPU, per core memory is size-limited, but memory spaces are set aside for different purposes and designed based on access patterns. Using SIMT (single instruction, multiple threads) execution pattern, threads are executed in a warp that commonly includes 32 threads; the threads within one warp will share specific memory space for fast inter-thread communication.

Four core eight threads CPUs have dominated the middle-end to the high-end consumer market for the majority of last decades, while the current top consumer CPUs can have up to 64 cores. Current CUDA GPUs have at least 32 threads in a single threads warp as the minimal execution unit, and high-end consumer-grade GPUs such as GTX 2080 Ti would have 4352 CUDA cores.

SIMD (single-instruction-stream, multiple-data-stream) is a category of computer machine organizations [101] that focus on improving the effectiveness of computers by executing the same instruction on multiple data simultaneously. Array processors, pipelined processors and associative processors are all types of SIMD machines and comparing to the most conventional and simple SISD (single-instruction-stream, single-data-stream), there are several notable challenges, including handling the communication between processing elements (recently referred as processor cores, but not independent processors), degradation due to sequential code and branching, and more. The general format of those SIMD processors is that M data streams can be processed one at a time by M processing element, aiming to achieve $1/M$ bound of performance improvement over SISD. However, the actual improvement can be affected by many factors, such as branching degradation, and results in close $log_2 M$ relative performance.

SIMT (single instruction, multiple threads) execution model is introduced by NVIDIA with the G80 architecture GPUs that support the first version of CUDA [102]. The SIMT model is an enhanced version of SIMD, where multiple threads can be executed at the same time by multiple cores, possibly manipulating different data, which is close to how threads are handled on the CPU. Threads are launched in **warps**; for example, a single streaming processor (SM) can have 48 active warps, each with 32 threads. Each warp will have one instruction pointers that fetch the following instruction from the instruction cache to the warp queue waiting for the scheduling unit to pick up the instruction; it will not fetch the instruction 32 times for each thread. This reduces the overhead of instruction fetching and reduces the memory access, which brings latency reduction. Scaler cores in the SM are sharing controlling and other structures to save space for more cores. It is summarized by [76] that GPUs are heterogeneous computing devices that are cheap, easily accessible, easy to maintain, easy to code. They are dedicated local devices found in many workstations and laptops, with relatively low power consumption.

2.4.2 GPU Programming and CUDA

There are several ways we can develop programs for running on GPUs, and CUDA is the focus of our discussion here. CUDA was initially released in 2008 and become available with GPUs built with core G80, such as GeForce 8800. Newer generations of CUDA-enabled GPUs, added features and functionalities with new architectures and new CUDA versions. For example, the three-dimensional grid of thread blocks introduced in CUDA 2 with Fermi architecture and the Tensor Core API support from CUDA 9 when those first-generation deep learning specialty cores were added with Volta architecture. It is a general-purpose programming language that can program both graphic and non-graphic calculations on easily accessible NVIDIA GPUs. Other options, such as OpenCL, have lower popularity but are vendor-neutral and can be used with a broader range of hardware.

Programming with GPUs is different, even when comparing with multi-threading programming on CPUs. Most notably, the memory management and hierarchy are essentially different from CPUs. The locality is vital since moving data around is a very high-cost task. Instead of the notion of stack, heap, and cache, GPUs have global, constant, shared and other kinds of memory, each for different purposes. If incorrectly used, it can cause severe performance downgrades. CUDA virtualizes the physical hardware, such that a thread is a virtualized scalar processor, with registers, program counters and state, and a block is a virtualized multiprocessor with shared memory.

GPU acceleration gained its tremendous popularity over the past decade, especially with the common usage in artificial neural networks, deep learning and image processing. In [103] a cross-GPU neural network implementation was used because 1.2 million training examples are too big for a single GPU. But since multiple GPUs on a single machine can access each other's memory with minimal work by the host machine, dividing the neural network into two GPUs making training deep networks on large data set not only possible but also faster. The authors predicted that the performance of their neural networks would improve as GPUs become faster. Recently many deep learning frameworks led by TensorFlow can run on heterogeneous distributed systems with hundreds of nodes of CPU, GPU, and TPU (Tensor Processing Unit) devices [85, 86].

Execution Configuration

When launching a kernel function from the host to the device, the execution configuration will need to be specified in terms of grid dimensions and block dimensions.

```
    kernel_function<<<Grid_Dimension, Block_Dimension>>>(parameter1, parameter2,..);

```

Listing 2.1: Kernel Launch Configuration

Both grid dimension and block dimension are of type *dim*3, i.e. three-dimensional numbers. By dividing the threads into blocks and grids will help the developer map the logic to the hardware — for example, many rendering tasks are naturally three-dimensional, such as many images and video rendering tasks. General-purpose computation workloads usually need many blocks to scale on different GPUs. The configuration also has impact on the performance; different configuration will result in different execution flow and memory-caching and accessing patterns on the GPU.

Memory Model

The GPU programming model offers several types of memory for the developers to use while the CPU offers a flat memory structure where the hardware manages the actual memory mapping. CUDA GPUs have a large on-device Dynamic Random Access Memory (DRAM) normally in the size of several gigabytes. The majority of the DRAM is allocated as global memory accessible to threads in all blocks. Programs can use the global memory by allocate a continuous space using `cudaMalloc` and copy to and from host memory space. Individual thread have access to a per-thread local memory space on the device memory for certain variables that are not suitable for on-chip storage. Threads within one block will be executed together on a single SM processor and will be allocated a chunk of memory to provide high-speed read and write [76]; this shared memory is on-chip and thus offer the lowest latency, though it is usually much limited in space when comparing to on-device memory spaces. Constant memory is 64kb storage on-device memory but fully cached beside the chip, which means only one operation is needed to read from cached constant memory. Constant memory lives in the DRAM but is cached in several layers; once fetched, the read-only constant memory is stored very close to the processing units, and the cost of reading from constant memory is minimal, which is similar to read from per-block shared memory. Similar to global memory but different from shared memory, constant memory can be accessed by all threads in all blocks. Because it is defined globally, the content of constant memory exists while the application is live. It performs the best when all the threads try to read the same value at the adjacent locations. This kind of memory accessing pattern is called *coalesced* memory accessing which can reduce the cache missing and memory fetching and achieve the highest possible core utilization. There are tools shipped with CUDA toolkit to measure the core utilization to help developers understand the potential improvements available.

2.4.3 Programming Strategy and Best Practice

CUDA is a programming model that enables heterogeneous computing with NVIDIA GPU accelerators. CUDA C and C++ is the language extension that provides the interface to the CUDA computing devices. CUDA programming model is heterogeneous as the CUDA applications used both CPU and GPU to finish the task — the GPU is therefore called the co-processor to the central processing units. This model is similar to FPGA devices but different from programming and compiling targeting a specific platform, such as ARM devices [104, 105].

As summarized in [2], the essential strategy for efficiently developing CUDA applications is to adopt the Assess, Parallelize, Optimization and Deploy (APOD) design cycle as in Figure 2.4. This incremental developing cycle brings numbers of benefits compared to developing a fully optimized GPU solution:

- Using reference code and unit tests to verify the program's correctness during the migration to the GPU. Starting from a working and non-CUDA implementation will provide a set of correct/expected results for validating the CUDA code later;
- Because of the heterogeneous property, it is possible to optimize one part of the code and leave other unchanged; by optimizing the part that can make the most out of the highly parallel architecture of GPU, performance can be improved quickly and modularly;
- The cyclic workflow makes sure we assess the performance of the program at each step. The assessment will provide a deeper understanding of the program and algorithm, validate the improvement, and select the most valuable part to speed up next.

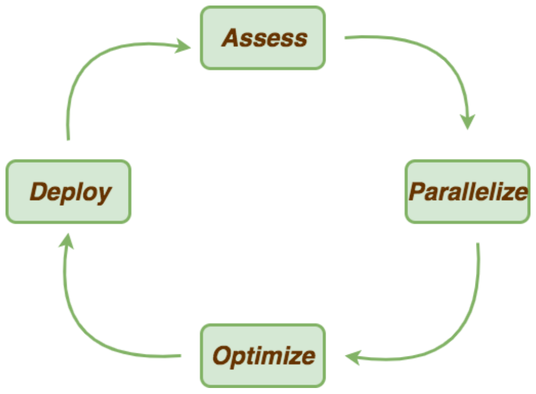


Figure 2.4: Assess, Parallelize, Optimization and Deploy (APOD) Cycle for CUDA Programming, Adapted from [2]

2.5 Summary

This chapter outlined the details of the algorithms and related research work. The streaming processing system to be described in the next chapter will be built on top of the existing solutions and tailored to fit our needs for simulations with PF and Particle MCMC. The Bayesian methods mentioned here can be combined with compartmental simulation models for accurate prediction and projection. In the context of public health and social well-being research, those can offer valuable input to aid decision-making proactively and responsively, even more so when enhanced by the streaming system. Accelerating them becomes very desirable because of the critical roles that those applications can play during critical times such as in pandemics. Graphic processing units discussed in this chapter have shown to be a strong candidate for such tasks. In the following chapters, our streaming architecture will be explained and applied to Particle MCMC and simulation models with the CUDA GPU-accelerated implementation.

3 Streaming Architecture For Simulation Models

3.1 Introduction

A data streaming system for simulation models enables the models to continuously consume additional information from the live data streams and progress forward without constant manual intervention. Such systems have become increasingly desirable in recent years. Analysis is required to be conducted in a fast, continuous and near real-time fashion to secure the potential value of the data. An efficient, effective and flexible software platform is needed to filter and stratify valuable information from the data that comes in with enormous volume and high velocity, while can simultaneously accommodate the variety and veracity of the data. One common scenario is that a unbounded data source is connected to a set of processing units consuming the data, such as those in online machine learning configurations. Response time is a critical factor and results can be obsolete quickly. Projections and forecasts are only valuable when they are produced before they are needed. The volume of available data will grow exponentially. A streaming system in which the data always flow is crucial for efficient large-scale data analytic, machine learning applications, and simulations. The ability to efficiently streaming the data is the critical; manually processing the data and then feeding them to processing units is obsolete. It is rare for standard simulation software to have streaming ability out-of-box but can be extended to support live data sources. AnyLogic is a popular simulation software for building high-performance simulation models and using Java as the programming language offers excellent flexibility and extensibility. Computational intensive algorithms such as Particle Filtering can be implemented within AnyLogic. The models built using AnyLogic are selected as potential targets when designing this streaming system and served as an example.

The primary considerations for designing such a system are listed in Figure 3.1. This chapter first outlines the core requirements of the streaming system using a simplified version of the software requirements specification (SRS) template adapted from IEEE Standard 830-1998 [1]. This systematic listing of the requirements is necessary; it acts as a standard way to communicate the system's functional behaviour and also emphasizes the crucial considerations for designs similar to this one [107]. While several systems and solutions can achieve subsets of the tasks and share some common characteristics, most of them lack a detailed description of the requirements or explanation of design choices. This can make extending or adopting those systems a challenging task - the layout of system requirements is aimed to simplify those tasks; also, this will help

[1]IEEE 830-1998: IEEE Recommended Practice for Software Requirements Specifications [106]

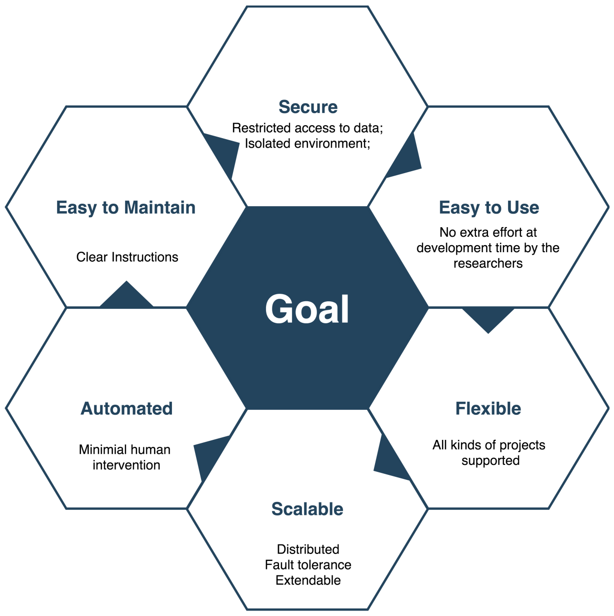

Secure
Restricted access to data;
Isolated environment;
Easy to Maintain
Clear Instructions
Easy to Use
No extra effort at development time by the researchers
Goal
Automated
Minimial human intervention
Flexible
All kinds of projects supported
Scalable
Distributed
Fault tolerance
Extendable

Figure 3.1: Streaming Architecture Requirements

precisely guide the development and, in the end, serve as the template for evaluation. A wide range of tools, frameworks and software are examined, and suitable ones are chosen with careful reasoning. The frameworks and tools will be discussed in each category, and the selected ones will be presented in detail to elaborate on why the choice can help build the final software solution. Followed those, the design and implementation are discussed in detail. We evaluate the system from different aspects, including ease-of-use, performance, scalability, availability and latency and the validation of requirements. We hope that those findings can serve as references for any future work related to similar applications.

3.2 Software Requirements Specification

Specification Introduction

Purpose This specification describes a software system that will work with multiple data sources in a streaming configuration and transports the data to simulation models to support online Bayesian statistic methods. Once setup, the software system should be deployed as a long-run service to continuously gather the data and run the models forward, while providing functionalities for monitoring, visualization and data storage.

Scope This specification is for the initial version of the system; the set of software being developed here is used as an illustration. With simple modifications, it can be used in the production environment. The software should focus on providing streaming functionalities to existing and future epidemiology and public health models. The data sources will regularly update to generate new data points, with a possible interval of milliseconds to days for each new data update.

Overall Description

Product Perspective This software product will need to contain three major components, a set of software that deployed as services interacting with each other, client libraries and plugins for languages and simulation software integration, and documentations and guidelines for maintaining the system and developing extensions to work with the system.

Product Features The system will have the following features:

- Concurrently supports multiple data sources; the data will flow through the system with minimal end-to-end delays;
- Concurrently supports multiple data consumers;
- Simulation models can join the system and start to consume the data at any time; options should be provided such that the models can start with existing data, whether that is custom data provided on the model side, or retrieving the data history from the server side if so desired;
- The overhead of integrating with the system and its components should be minimal; that is, for existing models, minimal changes should be made to work with the system, while for new models, no extra effort is needed in terms of model designing; the streaming data should be able to be consumed mostly the same as static data sources;
- The system can handle data sources in different formats, with different time units and time interval;
- The effort of operating the system is minimal; the support should cover most major operating systems and languages;
- The system provides monitoring and visualization functionalities to view the on-going experiments and perform analysis in a later time;

User Classes and Characteristics They are three different kinds of users:

- System Administrator: Will maintain and monitor the system; interact directly with the server and different system components; restore the system in the event of failure;
- Researchers: Primary users who will develop their own simulations models to work with the system, with the help of the guidelines and interfaces;
- Data Owner: Those collect data and want to distribute the data using a standard method.

Operating Environment The system will be able to deploy to standard servers, including public, private or hybrid clouds and developer machines. It mainly targets Docker container hosts, such as AWS ECS, Azure Container servers, or Google GCE Kubernetes hosts. Also, the system should work on developers' machines with correctly configured Docker hosts.

Design And Implementation Constraints The system needs to work with the simulation software AnyLogic. The connection to the model side should not take a vast amount of computational resources - rather, it should have minimal impact on the hosting machines running the simulation models.

Assumptions and Dependencies The system is designed targeting AnyLogic *8.5.0* and designed decisions are made based on the available features; later versions may include improvements and features for better integration with the system. The system and its components should make minimal assumption about the model structure.

System Features

Multiple Data Sources Concurrent data sources should be supported with modest server hardware and computational resources, without performance downgrade. The data source can be different in velocity, size and format, but should all be processed simultaneously and can be accessed and consumed by the clients using the exact mechanism. Familiar sources are those available on websites, social networks and data service APIs, relational databases, NoSQL such as Cassandra, Kafka, and other streaming and message queue systems. All data sources should support the options for live visualization and persistent storage.

Multiple Data Consumers The system should be able to distributed the collected data to more than one data consumers at a time. Multiple simulation models with Bayesian methods can subscribe and receive the streaming data simultaneously. Each model can join and leave independently with affect others.

Data Stream Handling The system should provide options for different usage of the data streams. In some cases, simulation models are built to start fresh for each deployment and will only need to receive new data points. In other cases, which is more often, the models will need to start from a particular date and time. For this, the system should have interfaces for the model to either supply the historical data when subscribing to the live data sources; or, the model can request and have access to the historical data. The system will need to provide the mechanism to save the intermediate data, in order to restore and repeat the data streams. Simulation models can join the system and start to consume the data at any time after the system has been deployed.

Integration To support streaming needs for simulation models with Bayesian statistics methods, it needs to integrate into existing applications of simulation models. Those models are built using AnyLogic or other platforms and programming languages. The integration interface needs to be straightforward to set up.

Data Heterogeneity The system should focus on time series data primarily, with support up to data streams with sub-second frequency. The time series data can have different time units and the speed can vary.

Deployment and Operation Taking advantage of modern lightweight containerization technologies to simply the deployment and operation process.

Security Security is also essential; firewalls and authentications need to be considered making sure: data uploaded are authentic; only authorized client models can consume the data; results are protected from unauthorized access.

Visualization A series of website user interfaces should be available to access and view the collected data, monitor the system and the models, and evaluate the experiment results. The GUI should display the real-time data collected but can also be review in a later time.

Other Nonfunctional Requirements

- Performance Requirements: End-to-end delay should be minimum, and the system should be scalable for many models and data sources;
- Easy to deploy and maintain: Each model needs to be able to subscribe to several live data sources, and each data source needs to be consumed by many clients simultaneously.
- Software Quality Attributes: thoroughly tested codebase;
- Fault tolerance and recovery: Capable of recovering from common network connectivity issues.
- The ability to monitor the system; easily view the logs and system usage and ensure performance.

3.3 Design and Implementation

3.3.1 System Architecture Overview

Application Programming Interface (API)-centric, components and microservices-oriented design can provide good flexibility and fulfill the requirements when building streaming systems for simulation models with Bayesian methods. Instead of building an application that works for one use case with one model and one data source, it is worth emphasizing the importance of building such a system in a modularized way. By building each component separately and defining standard APIs, the modules can communicate and work together to provide the service to support different models and use cases.

Figure 3.2 presents the core architecture of the system design. A single deployment is defined as a set of simulation models and their surrounding system components focusing on related topics and research questions. In most cases, those models will share one or more empirical data sources, or targeting similar issues, thus can be justified to deploy and operate together. Within one deployment, the system is divided into four sections:

- Data Sources: For each of the data sources of interest, an data collector will be set up to create the corresponding data flow. The data sources, as described earlier, can in one of the many formats with very different characteristics. We can see that a website with empirical data updated routinely is among the most popular method for publishing data related to public health events for best accessibility; other types of data sources would involve programmatically connect to databases, data warehouses, APIs or servers of data collection websites and mobile applications. All data sources that provide those raw data will be in the source set;

- Adapter Group: The software components that connect, collect and process the data from the data sources will be gathered in the adapter set. Because of the heterogeneous data sources, adapters are primarily distinct from one to another, though most of them can be constructed from templates. For example, for most social media network APIs without native streaming capacity, actively pulling processing pipelines are needed to get the data flow from the sources to the next steps. In this set, there will be secondary adapters, which will go through a standard set of operations, such as save to persistent storage, log and notify users, push to visualization sites.

- Central Hub: The hub is a central server that connects the data sources adapters and the models. It accepts the data from the adapters and broadcasts to the rest of the system; data source adapters can register and publish data series prepared to channels, and models can consume data by subscribing to the channels that are needed by the simulations. The hub also has built-in support for visualization of the data collected, data storage, and replay of pass data streams;

- Model group: Simulation models that consume the processed data will be grouped and placed in the model set. While the system's primary focus is to support AnyLogic-based Bayesian simulation models, other kinds of models can also be used.

- Analysis Group: The group of components and software provide the functionalities for analyzing and monitoring the running models. The experiments' output is saved, indexed and ready to be explored using web page-based visualization tools, in real-time or for later usage, offering the ability to view the results of single experiments and compare and visualize across different ones.

The connections and communications between the groups of components are defined as follows. For communications between data sources adapter set, depends on the type of the data sources, this connection will be HTTP-type of connections over public or private networks. Those between adapters, hub, models

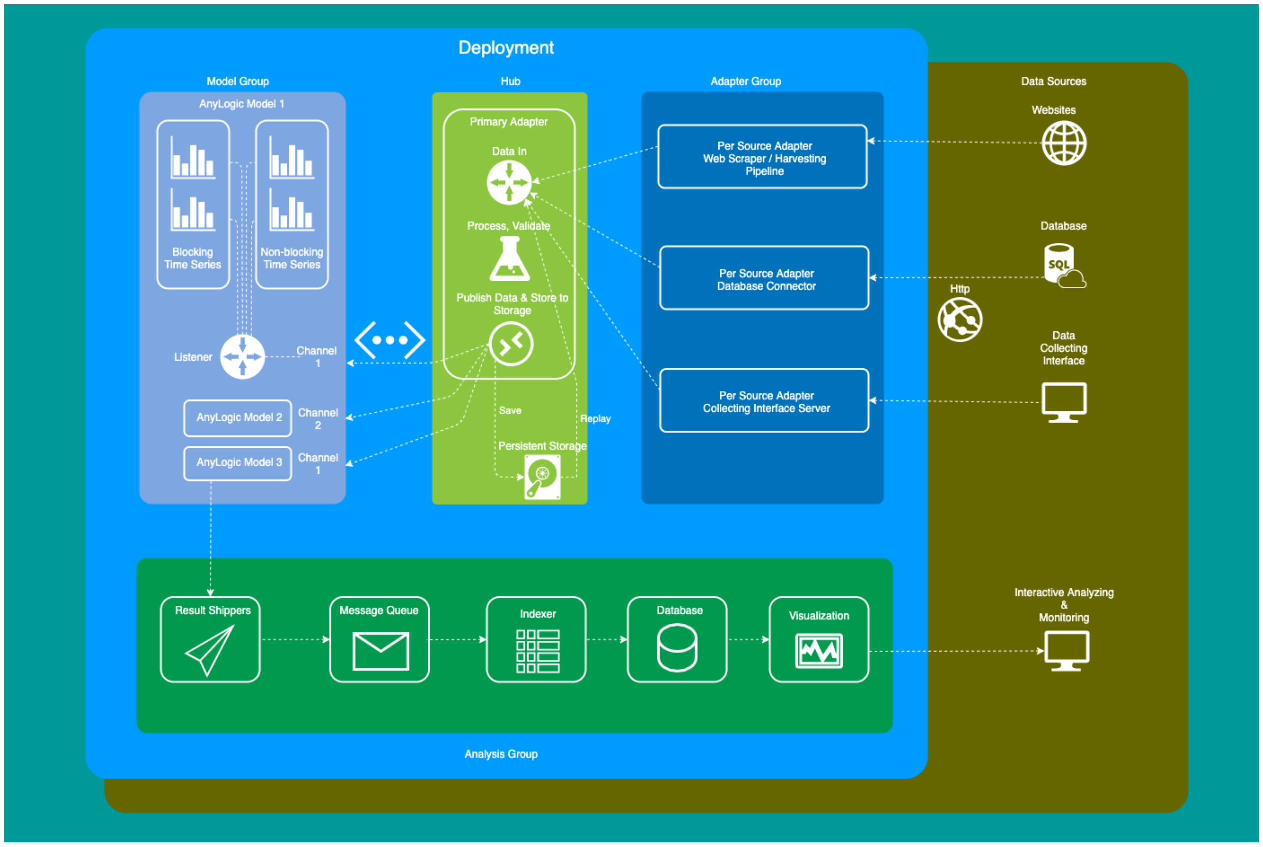


Figure 3.2: Overview of Streaming Architecture for Simulation Models

and analysis sections are controlled or authenticated internal API communication, Remote Procedure Call (RPC), or HTTP-type connections over the network.

As the demanding for streaming analytics increasing in recent years, many software solutions and frameworks emerged. It is essential to review mature products, software, tools, and frameworks related to streaming processing before making significant design decisions. The first step in drawing an improved system's design is to thoroughly review and understand the limitation and advantages of existing solutions. As listed in the background chapter of this thesis, good streaming solutions targeting public health, individual healthcare, simulation models and Bayesian inference possess common characteristics, including:

- Clear boundaries between ingesting, preprocessing, storage, modeling and visualization;
- Well-defined communication channels and protocols;
- Components for individual tasks, data sources, and models;

When selecting frameworks and tools for building the system, the inclusion criteria for the choice are:

- Scalable for large streaming tasks and needs to be able to deploy on distributed infrastructures;
- Failure tolerance and fast recovery with low operation and configuration overhead;
- Easy integration with other system components and support communications through HTTPS, preferable with RESTful APIs;
- Easily accessible, actively maintained and has active community support.

With the components-orientated design presented in Figure 3.2, we can accomplish the requirements while maintaining maximum flexibility. With those in mind, an implementation of the system is accomplished as follows: An AnyLogic plugin was built using Java language with Maven managed packages to connect to models built in this software, offering the interface for the model builder to integrate the streaming system. The plugin uses another custom-built Java package, and it handles the communication between data collecting adapters and models. A live time series visualization website is built using Python and *Plotly* library. The hub is implemented with Redis, using the PubSub and Stream data type. Docker and Docker Swarm are used for deploying and operating. Container monitoring and system monitoring is accomplished with ElasticSearch software stacks, namely ElasticSearch, LogStash, and Kibana; a ZooKeeper instance is at the core to ensure availability across the system for components built with Apache Storm and Kafka. We use ElasticSearch for experiment results indexing. The experiments visualization and monitoring are achieved with the Kibana web visualization interface. LogStash, Kafka, FileBeat are used for experiment results and metrics collecting, transforming and shipping. Several sample applications have been implemented and tested, which will be discussed in detail in the later section. On the data source adapter side, several real-world examples will be presented; Apache Storm was chosen as the main framework for most adopters for harvesting and processing the data. One AnyLogic model is used to demonstrate the integration with the models.

3.3.2 Data Collecting Adapters with Distributed Streaming Platforms

Data sources are rarely standardized working with public health and healthcare data [35]. When developing the system, it is required to consider the heterogeneity in terms of speed, size, format and priority of the data sources. Adapters are deployed as individual services and need to be robust and can recover from failures and exceptions, such as those caused by unstable internet connections. The streaming system proposed here has relaxed requirement over the architecture and implementation details of the data collecting adapters; the requirements are focused on the communication interfaces. After the collectors have the time series data ready, data is shipped to the system's central hub. Each adapter will request to create its own named messaging channel in the hub first, where the channel name donates the name of the data source to be recognized by the downstream models. The data type can be any but are usually floating-point numbers and integers, with an optional time step index as part of each message.

High-performance distributed streaming process frameworks and platforms emerged over the past few years share the common characteristics of distributed in nature, simple abstractions for quick design and deployment, built-in fault-tolerant and more. Examples include Apache Kafka, Apache Spark, Apache Storm, Apache Flink, Apache Samza, and Cloud providers options such as Microsoft Azure Streaming Analytics, Amazon AWS Kinesis and Google Cloud Platform Dataflow. Those can be divided into two categories based on how messages and data are processed: Those that process them by mini-batch, such as Spark Streaming, and those that handle messages one by one natively, such as Apache Storm. All of those frameworks can be part of the system described in this chapter for the data collecting tasks. The choice of a framework depends on many factors. One selected framework is described here in-depth, Apache Storm, to show the core features and focuses when choosing. While there are many well-designed generic or customized streaming solutions available on the market, Apache Storm is one of the most widely used streaming frameworks. It stands out because of low latency, high performance and distributed nature, readily accessible APIs and simple programming abstractions.

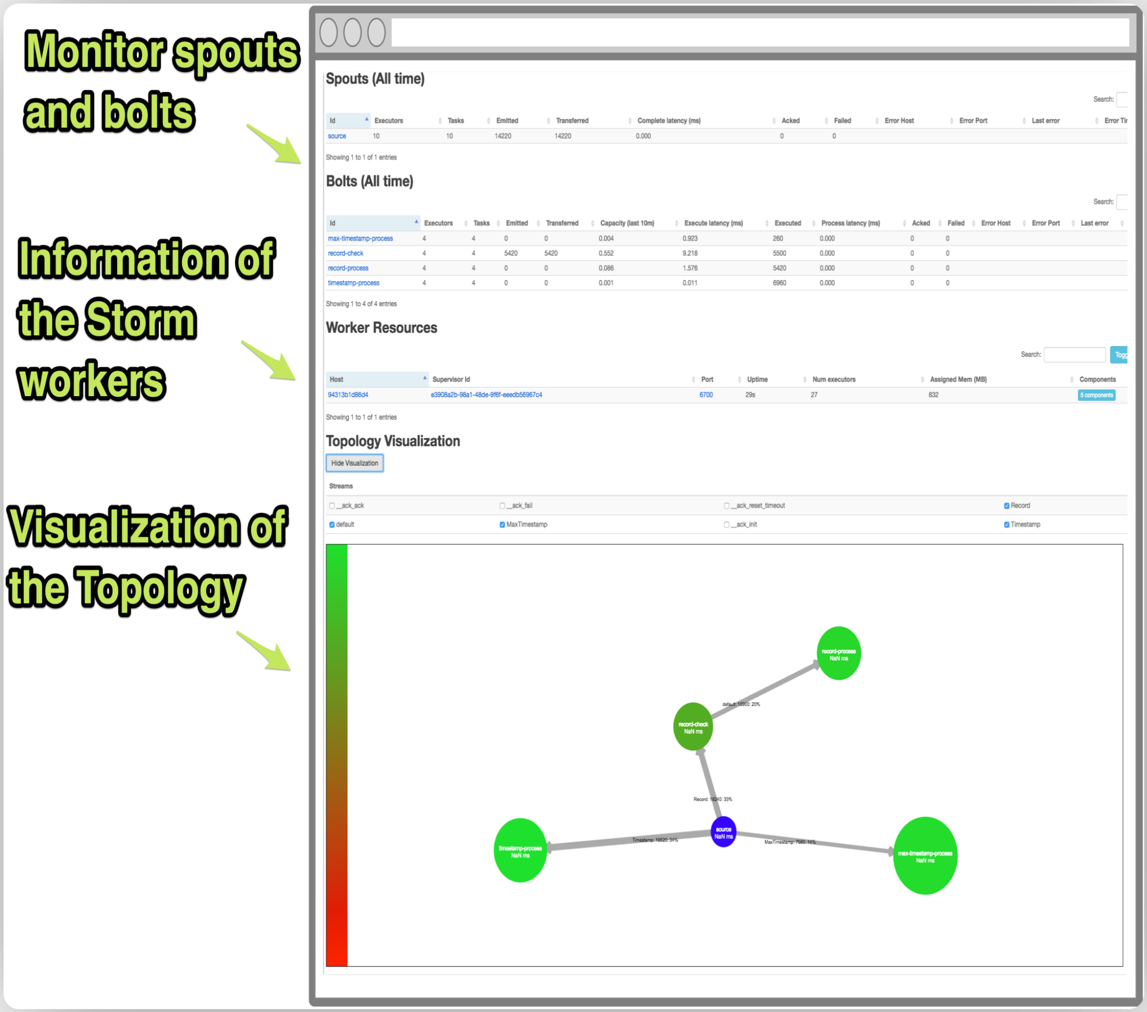
Cluster Structure Many big data tasks require computation power that usually cannot be satisfied by a single machine and a cluster of machines is required instead. Similar to Apache Spark, Apache Storm is designed to be deployed on configured clusters. Three kinds of nodes, or services, are needed for a Apache Storm cluster: Zookeeper nodes, Nimbus nodes and Supervisor nodes. Apache Zookeeper is a commonly used distributed application centralized coordinator, providing functionality such as maintaining stateful configurations, naming and distributed synchronization. It can be set up on an Apache Spark cluster for availability consideration – in the cases where the Spark master node fails, Zookeeper can elect a new master node from the standby master nodes to avoid the single point of master node failure. Apache Storm uses the Zookeeper node to achieve the same availability requirement. More than that, Storm preserves most of its state in Zookeeper, so that the Nimbus nodes and Supervisor nodes can be stateless. When catastrophic failure happens on any stateless node, it can be quickly restored without restarting any submitted job. This is critical for Apache Storm because most jobs run endlessly once submitted. Nimbus nodes serve as the master node in a Storm cluster; new Storm jobs need to be initialized and submitted on a Nimbus node. A Nimbus node is responsible for distributing the tasks and data to the worker processes in supervisor nodes and monitoring the progress. When a worker process or a supervisor node down, the Nimbus node can reassign the task to another node without affecting the overall job progress. All other nodes can be Supervisor nodes; on each such machine, multiple worker processes can run tasks from different jobs. Apache Storm is also YARN compatible, which means it can run side-by-side with other frameworks such as Hadoop or Spark on a YARN-managed cluster.

Abstractions One of the key features of Apache Storm is that it provides straightforward data processing abstractions. A *spout* is an abstraction for a data source unit, where a *bolt* is an abstraction for a processing unit; data flow between those in the form of a tuple. The spout interface has a *NextTuple()* method, which is invoked repeatedly to emit tuples. Inside this method, one can hold a subscription to an external data source such as a SQL database or a message queue such as Apache Kafka, watch for changes of files, or simply use a streaming source such as Twitter4J. The *topology* of a Storm application is a Directed Acyclic Graph (DAG) built using Spouts and Bolts and connected by different grouping strategies. One Storm application can have one or more typologies working simultaneously to express the business logic. When building the topology, replica count can be specified for spouts or bolts, depending on the workload. The grouping strategies will handle the distribution of tuples between replicas; for example, a field grouping will send all tuples with the same value in a specified field to the same instance of a Bolt.

Guaranteeing Message Processing Apache Storm also has a built-in mechanism to guarantee messages or tuples are processed in a certain way. With the help of basic **Ack()** and **Fail()** methods, Storm can be tuned to have each message processed at least once through the topology. This is useful in cases such as sensor data, where we want to include all possible time points or duty cycles. Built on top of the Storm, Trident is a high-level abstraction that provides additional functionalities, which includes an exactly-once guarantee needed for tasks involving transactions.

UI Apache Storm is shipped with a browser-based monitoring UI and includes many optional third-party monitoring interfaces. This native web UI provides an easy way to monitor the rate, latency and count of each processing unit, display a visualization of the topology, and give navigation through job logs, as shown in Figure 3.3.

Summary The mentioned properties of the Apache Storm make it a perfect choice for implementing data collecting adapters while achieving the maximum flexibility, outstanding performance, and improved usability. In this chapter's applications section, examples have been presented with different data sources and are built with various kinds of streaming frameworks. In one example we build a streaming pipeline using Apache



Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Ti
source	10	10	14220	14220	0.000	0	0				

Showing 1 to 1 of 1 entries

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error
max-timestamp-process	4	4	0	0	0.004	0.923	260	0.000	0	0			
record-check	4	4	5420	5420	0.552	9.218	5500	0.000	0	0			
record-process	4	4	0	0	0.086	1.576	5420	0.000	0	0			
timestamp-process	4	4	0	0	0.001	0.011	6960	0.000	0	0			

Showing 1 to 4 of 4 entries

Worker Resources

Host	Supervisor Id	Port	Uptime	Num executors	Assigned Mem (MB)	Components
94313b1d86d4	e3908a2b-98a1-48de-9f6f-eeedb56967c4	6700	29s	27	832	5 components

Showing 1 to 1 of 1 entries

Topology Visualization

Streams

☐ __ack_ack	☐ __ack_fail	☐ __ack_reset_timeout	☑ Record
☑ default	☑ MaxTimestamp	☐ __ack_init	☑ Timestamp

Figure 3.3: Monitoring the Job Using Storm Native Web UI: Rate, Latency and Process Count on Spouts and Bolts, and Visualization of the Topology

Stream to retrieve and collect publicly posted social media contents using the API. The collected data is then cleaned and transformed. The Storm application can then group data by time window and extract time series, while persistent the intermediate data to database. With Apache Storm connecting the data source and acting as the upstream of the streaming system, data would flow efficiently and effectively in the system and help the system achieve its goal in delivering a low-latency easy-to-use streaming system for simulation models.

3.3.3 Central Hub Using In-memory Message Broker

Similar to streaming processing frameworks, message brokers are ready-to-use, high-performance software and services facilitating functionalities for efficient communicating and message passing of other system components. Examples are Apache Kafka, ActiveMQ, RabbitMQ, Redis, and solutions from cloud service provider such as Amazon AWS Simple Queue Service. Similar to the previous section, we will discuss the features and characters of Redis in detail and describe why it would be an excellent choice for the system. Redis is a fast and reliable in-memory high-performance data structure storage system. It can store common data structures using key-value schema and provides APIs for accessing them from common programming languages, which makes it a perfect choice to use as a speed-critical database, cache service, or message broker. While Apache Storm makes it easy for processing units such as bolts and spouts to replicate, but since all the spouts replicas run the same code, coordination is needed and it is essential to be able to distribute the tasks among them in a bag-of-tasks fashion, and a Redis instance would fit perfectly for this job. It is an excellent solution for inter-process communication since it is resident all in memory and can provide the best performance. Because of the in-memory nature of Redis, it is ready to handle storage and query requests much faster compared to traditional data stores. Among all the other data types, Redis supports lists and sets. Lists can be pushed and popped from either side to make them work as a queue or stack, while the insert (**SADD()**) and membership check (**SISMEMBER()**) on a set are all **O(1)**. In the later sections, we will show how we use those structures efficiently to coordinate between Apache Storm spouts. Redis also supports setting a Time-to-Live (TTL) on any data object, and offers threads-safe atomic operations such as increase-by-one (**INCR()**) and get-set (**GETSET()**). Those functions can be useful for Storm applications, such as those undertaking aggregate counting of certain events or keywords in input data streams. PubSub is one of the data operations supported by Redis that uses channels to distribute messages. Each channel can have as many subscribers as possible, and the delay is minimal. It is "Fire and Forget" schema: The message will be removed from the storage once delivered to all the subscribers. The newly released Redis version 5 includes a new data type called Streams[2]. The Streams data type is built with the primary use case as processing system log type of messages. Comparing to PubSub, the message will be saved so that new consumers can access the messages from the beginning or query the message from a specific time point, similar to use *tail* utility to read log messages from the beginning or starting from a particular line. Beyond

[2]Redis Streams: `https://redis.io/topics/streams-intro`

that, it also supports subscription groups to consume and process messages in one shared channel collectively. Redis Streams and Pub/Sub are also different in how to consume the messages. The streams API is similar to the pulling strategy where the client can check actively for the new message whenever the processing of the current messages is finished. The Pub/Sub would push the message to the client, and the client should be designed to process the message quickly, use separate threads to process or have a receiving queue for buffering. The hub in our system takes advantage of Redis and implements the downstream in two kinds of interface, backed by Redis's PubSub and Streams primitives. Each time series will publish to their own named channel. One channel can be subscribed to by multiple clients. Upon arriving at a new data point to the channel, all clients will receive the update and move the model one step forward. The system will provide two modes: High-velocity and persistent. The high-velocity mode will best for the case with an extremely high frequency of new data, on the scale of minutes to milliseconds, and where the data expired fast and historical data is not needed. The persistent mode will best for time series that has a unit of day, week or month. The entire time series will be available upon subscribing. The high-velocity model is backed by Redis Pub/Sub and the persistent model is implemented using Streams data type. Persistent storage is implemented as a combination of log files and log collectors. The data are stored using Docker volumes to restore to working orders swiftly in the case of Redis down.

3.3.4 Database as Streaming Data Source

Database is another common streaming data source. Several approaches are available to use a database as a data source for the streaming system depending on factors such as server and firewall setup, data type, volume and velocity. One flexible and straightforward method would be installing a scheduled taks to query data in batch periodically and check for updates; this approach requires zero configurations on the source database and should work with different database systems. The disadvantage of this raises from the choice of the checking interval. Suppose the interval is set to a smaller value, such as seconds: in this case, the querying and checking will not be the most efficient when data updates are rare and can overflow the database server when updates are frequent and possibly slow down the server to process regular query and insertion requests. If the interval is set to a large value or dynamic adjusted, it will introduce delays in processing time. Essentially all the new records will be delayed by the time of the interval. Compared to this pulling strategy, a pushing approach involves more integration on the source database side and requires the database server's support but can offer much better performance and lower latency. Taking Cassandra database as an example, because of how data updates were performed, changes are written to the log file before actually committing to the database file; this implementation of Cassandra could offer many benefits such as guarantee the consistency of the database and improved disaster recovery. Such logs can be monitored, and the changes can be pushed to external services for tracking data changes without massive performance impact on the database server.

To show this, we develop a sample data collector built on top of this Change Data Capture (CDC) feature of Cassandra. The sample solution building with Java will watch the log files and push changes to a Kafka message queue instance, where a separate data source adapter will fetch the messages from Kafka to process, calculate, aggregate and output the time series. While the Java file watcher is running on the database server inside a single Cassandra instance, the Kafka instance and data processing adapter runs outside of the Cassandra servers. Thus, those that require storage and computation resource do not compete resource with the Cassandra database. This can be extremely useful if the database collects sensor data, wearable data, or other log-type appending only data, where updates and deletion are rare and not required to capture.

3.3.5 Persistent Storage and Replay

Distributed databases, especially NoSQL solutions, are essential components in building robust and scalable streaming data collecting systems. They are designed to handle loose-schema or schema-less data, including logs or sensor data, targeting large dataset with high throughput. Beside serving as the data sources that store the raw data for the streaming process as we just mentioned, distributed databases can also be deployed downstream of the message queue to persistent data for future tasks. They can store data collected by Apache Storm and other data collectors. The data can be easily queried, searched, or visualized in a web server, as shown in Figure 3.4. Distributed NoSQL databases can scale over a cluster and save data as multiple replications with built-in mechanism for non-blocking reading and writing for high concurrent IO and loose coupling. Examples include **HBase** that build on top of Apache Hadoop and its HDFS to take advantage of the Hadoop distributed file system implementation. There are also solutions designed to store system metrics, logs, and all kinds of time series data, such as InfluxDB and ElasticSearch. They have standard features such as built-in visualization, the ability to deploy on-premise or use as SaaS, plugin-based metric collections. Such solutions are perfect for the streaming result's persistent storage and later query and visualize the result quickly.

The persistent storage of the streaming data passing through the central hub serves another vital purpose. If the simulation models and experiments need to pass data or when the services experience an unexpected shutdown, the data storage can replay the data stream and be consumed again by the simulation models. Data will be logged to files by a Redis client to illustrate this concept with the sample setup; to replay the time series, a Docker container built with Elastic FileBeat will read the logs and emit the message back to the central server. Simulation models can request the replay, and the container will spawn from the server.

3.3.6 Integration with Simulation Models and Software

There are generally two different configurations on the software side to switch an application or algorithm to work with static observation data and streaming live data points. The first one is where the applications run to finish processing the current ones and wait for new observation points to become available. For iterative methods where more accurate results can be obtained if we increase the processing iterations, this requires the data points arriving at a constant speed, and then one can estimate the count of iterations, or the task

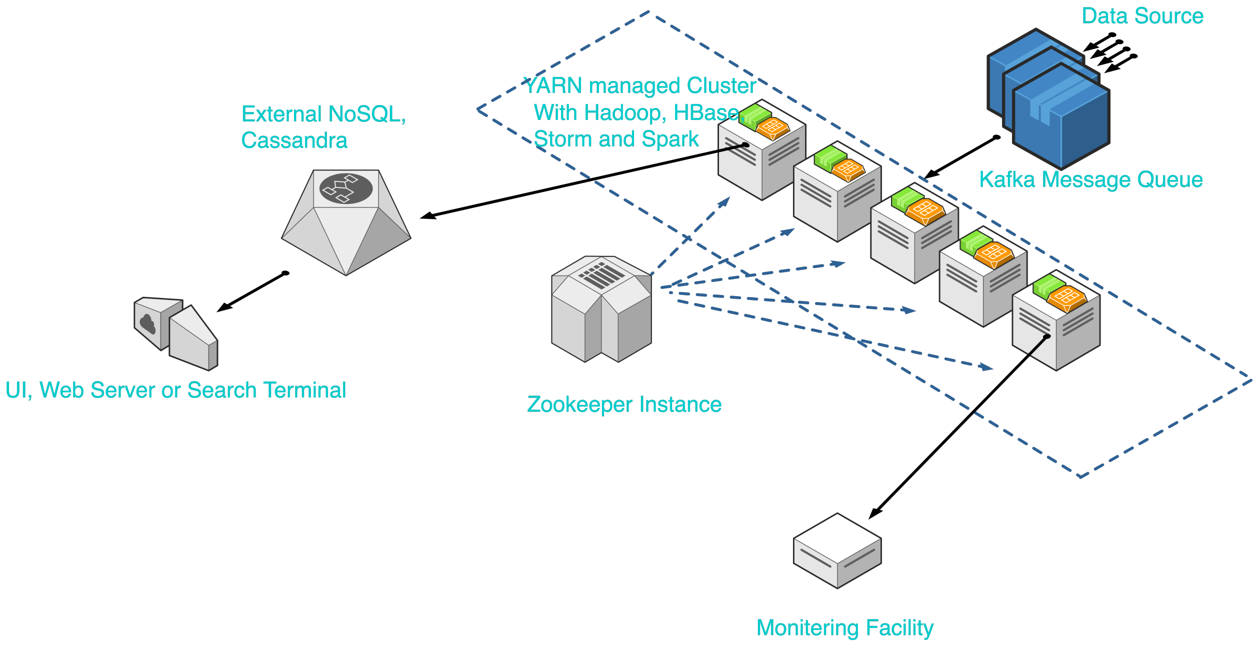


Figure 3.4: Infrastructure of a Robust and Scalable Streaming Data Collecting System with Apache Storm

size, based on the time window between time points. The second one is running until interrupted by new observation. While this requires an early stop mechanism, this method can guarantee the algorithm can achieve as many iterations as possible, and could be especially valuable when the data points arriving at a different speed. The non-iterative methods, such as PF, usually fall into the first category by choosing the amount of work suitable to the interval that observation arrives. For PF with even length intervals, that means choosing a fixed number of particles to finish before the next point becomes available to avoid piled up of unprocessed tasks. For such applications, only one thread is needed, and this thread is blocked when calling the synchronized functions to get the next observation point. On the other hand, for iterative algorithms such as MCMC and Particle MCMC, though we can also choose a fixed task size or the count of iteration, it is more convenient and efficient to interrupt the unbounded length task upon the arrival of a new observation. To accomplish this, it usually requires a separate thread to monitor the data source. The main thread is running the algorithm and only proceeding to the next task when the other thread informs the arrival of new observation over inter-process communication.

It is crucial to decide when to block or not block the execution of the streaming processing and block at others. For example, as a data upstream, the preprocessing side needs to select a non-block interface. But when connecting to the model, for example, in AnyLogic, the blocking I/O can be chosen to make sure that the model only moves forward when new data is available. Though it depends on the client-side, a non-blocking I/O can also be provided. One example of synchronous and asynchronous interfaces is Java's IO and NIO libraries, and blocking and non-blocking data types implementations such as for a queue. Using those along with the language support of multi-threading, we can implement thread-safe and high-performance functions to process streaming data.

We design the integration with the simulation software AnyLogic as follows: On the model group client side, there will be three componenets. Subscriber library, which communicates directly with the publisher, is implemented in Java and packed and distributed using the Maven repository and named **simulation_streaming_client**. The second part is the AnyLogic plugin **StreamingCollection** that can be used in the AnyLogic software interactively will use that library to provide the streaming interfaces to the model builders. It use the **simulation_streaming_client** library while providing a direct interface to other AnyLogic simulation models. This design provides multiple benefits: First, the subscriber library can handle all the other external dependencies easier, with the Maven building tool's help. If we implement both in AnyLogic, all the libraries need to be downloaded individually and independently linked to AnyLogic. This can be tedious when new versions of external libraries come available. Second, the parts that might need to modify because of the change of system layout, deployment environment, or simply security and reliability patches are all encapsulated in this subscriber library. Since the interface between the subscriber library and the AnyLogic Streaming Collection library is fixed, as long as we keep the interface identical, all those updates can independently apply to the subscriber library. This decouples the models with the streaming system, such as streaming components that can be updated without changing the AnyLogic Streaming Collection library or the already modified or deployed models. Third, by having that AnyLogic **StreamingCollection** library implemented in AnyLogic, this streaming feature is accessible through the AnyLogic palette UI and used in a way that is already familiar to the users, where interactions such as drag-and-drop and customize through the properties window are all supported. The last of the three is the simulation model itself, which uses the AnyLogic plugin when the model needs to access the live dataset.

Simulation engines such as the one in AnyLogic have their own scheduler to dedicated to invoke model events. More specifically, the engine will trigger the event based on their designated model time; when multiple events happen simultaneously, the engine will arrange and invoke the events based on FIFO (First-in-first-out), LIFO (Last-in-first-out) or random orders. User interactions through the user interface are treated similarly as events; the engine will process both the model events and user interaction events in order. The execution of one of those events is uninterruptible. This behaviour of the engine scheduler has those implications:

- Because the events are uninterruptible, they are **blocking** by default. If any event has blocking function calls in it, the engine will block until the event is finished;
- Since the engine also handles the user interactions, if the engine is busy executing a long or blocking event, the user interactions can become not responsive until the current event is finished;
- Controlling actions on the engine itself, such as engine pause, is thus can only happen in between events; if the pause function is called in one of those events, then the engine will only go to *paused* status after the current event is finished;
- When the simulation model is in the *paused* status, no model events will be executed, but the scheduler can still handle the user interactions;
- New threads created within the events will run independently of the engine; blocking child threads will not affect the engine or user interactions.

The design choices are then made based on those properties, as follows. The plugin designed for AnyLogic supports two modes: Synchronized and asynchronous modes, for blocking and non-blocking access to the next available data point in the data streams. The synchronized blocking access is straightforward in design: When calling *getNext()* on a time series using the plugin, this function will block the current thread until the next data point is available. The function can be called within the AnyLogic model, such as events or functions invoked by events. The advantage of this mode is that it provides easy access to the underline *live* data using the same interface as the *hard-coded* data, which minimized the changes needed to adopt the streaming plugin into the existing models. The disadvantages are also clear: Because the function is blocking, the engine will be blocked while waiting for the new data point, and the UI will be non-responsive. This becomes trivial when the data points arrive at a very high speed. The waiting is instantly finished and allows the engine to handle user interactions. This is also suitable for models executing in a production environment where no exploratory interactions are needed. On the other hand, The second mode provides the function to avoid blocking the engine execution and making sure the AnyLogic UI is responsive. By scheduling an event right before accessing the next data point to check if data point is available, the *isAvailable()* function of the plugin will issue a *pause* command on the engine if waiting is needed, and create a separate thread to call *resume* on the engine upon the data is available. The engine will thus go into *paused* status right after this checking event; the engine can process all user interactions and automatically continue the execution when the watching thread calls the *resume* command. This logic is encapsulated in the plugin and only expose the checking function *isAvailable()* to the modeler, and the rest of the data accessing logic are kept the same as the first mode.

The following steps are needed to integrate the plugin into an existing AnyLogic model:

- Gather the dependency library file *simulation-streaming-client.jar* and AnyLogic module *Streaming-Collection* library and add those to the model dependencies section;

- A new palette item called Streaming Collection library will be available in the AnyLogic UI. Drag and create a new instance of Streaming Collection from it and replace how the original model gets data points, whether it is an array or a table function. The parameters needed for the streaming collection are channel name and time unit, in the case where the time unit of data differs from the model time unit.

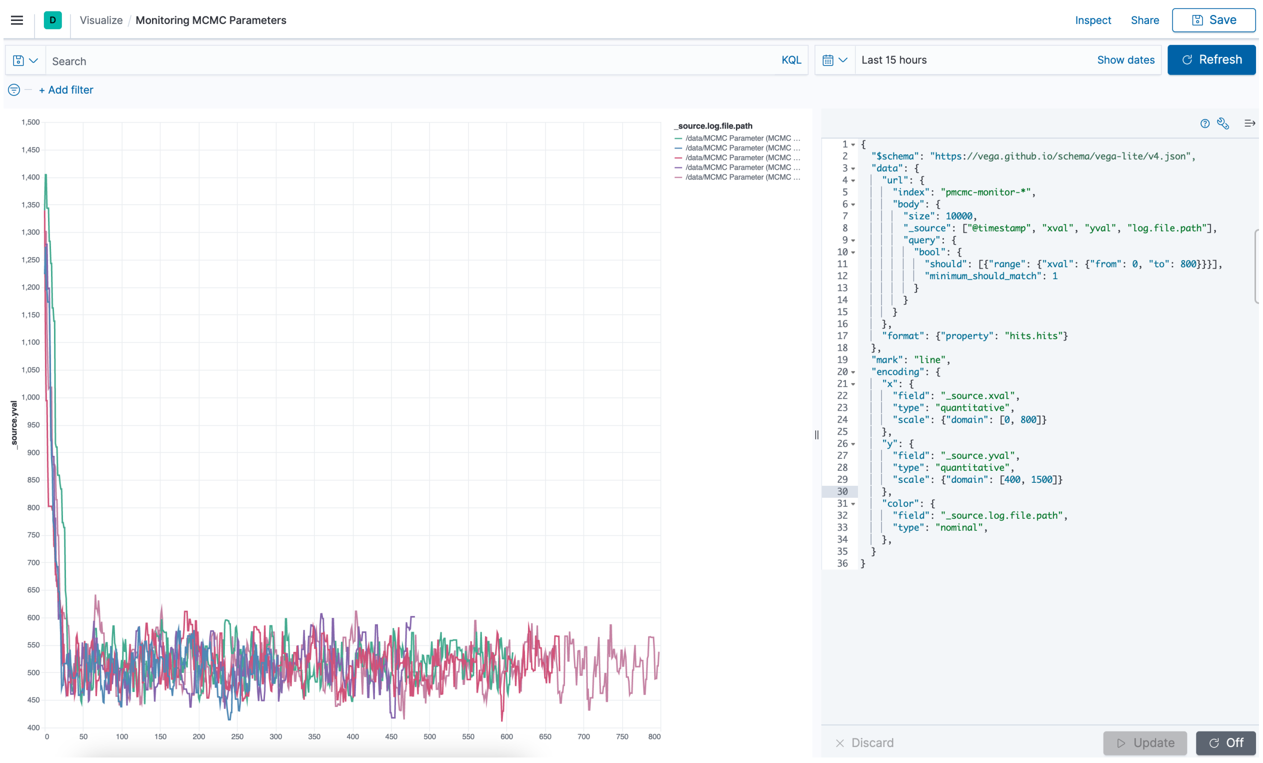
- Add as many of the collection as needed for each live data sources; create AnyLogic events and call *isAvailable()* on the collection object if needed.

Then the model is ready to be deployed with the rest of the system and proceed forward to make prediction and projection with the streaming data sources.

3.3.7 Streaming of Experiment Results & Real-time Visualization

The system design can be completed with a centralized reporting system for the streaming system to achieve the fully automated data digesting for real-time decision-making. The model presentation within the AnyLogic interface should not be the end of the pipeline. Monitoring the running models one-by-one is infeasible especially in a production environment. For a complete solution, we can build the system that collects and ships model output to a centralized remote server. AnyLogic simulations can produce a large output volume, especially during calibration, parameter variation, and running ensembles of realizations. The output can be processed and filtered while the simulation is still running. As one of the system's most immediate extensions, mirroring the streaming-in system to build the streaming-out architecture is desired. One implementation is accomplished using Elastic ELK (ElasticSearch, LogStash, Kibana) software stack. ELK is designed for the enterprise to build easy-to-use search over logs and other data. This makes it a perfect fit for our simulation requirements, whether in the development cycles or the deployment phases. By logging all outputs, even partial or unstructured, ELK would still let us look over by time, versions, runs and many other matrices in real-time. The models' output will be sent to ElasticSearch using LogStash and indexed within ElasticSearch, where Kibana provides query functionality for live interaction with the model output, and can build real-time visualizations and dashboards to provide information for decision-making. With the ability for dashboard-style on-demand querying and visualization for all the experiment executions. It can compare different values and results over different model versions or different realizations of the same version. With the help of visualization and interactive query tools, we will be able to have a glance at the results way before the simulation finishes; this can be a great time saver if we can early terminate useless long-run experiments. We believe such applications can dramatically increase the time efficiency of developing and calibrating AnyLogic models. Because the indexing of data results is happening simultaneously while the experiments are running, Kibana could provide visualizations without waiting for the experiments to finish. The efficient indexing setup on the ElasticSearch can make sure that we can observe and monitor all the running models' results simultaneously and build a visualization that provides insights far beyond a single experiment. For data that not being shipped to Elastic, the visualization will be built using Python Plotly Library. Comparing to other visualization solutions such as ggplot and matplotlib, Plotly can build interactive plots and optimized to be hosted and viewed in browsers. The real-time properties can also help tune parameters and settings and stop unwanted experiments earlier to save computation resources. As shown in Figure 3.5, with the Kibana web view, using simple visualization language Vegas we can query the indexed efficiently from ElasticSearch and generate plots with a defined time range, and the ability to refresh the plots quickly as new results available; in this figure, five instances of a Particle MCMC model are executed simultaneously, and the sampled MCMC values are sent and indexed in real-time as the models progressing forward. This visualization provides us in real-time the ability to monitor the experiment instance, check progress, and make early-stops to free computational resources when desired; dashboard-style visualizations can be set up quickly for different deployments of the model. Additionally, since the data points are stored in ElasticSearch and indexed by key experiment metadata, one can analysis and summarised across all past experiments based on filters; for example, we can check the key metrics such as acceptance rates of the Particle MCMC algorithm with respect to iteration settings for all past experiments using Vega visualization language in Kibana.

Similar to the data ingress in simulation models and simulation software, the egress of simulation results is also simplified by libraries and plugins provided. Data shippers are developed for different kinds of simulation models. For AnyLogic, we have a similar plugin-style shipper developed to ship structured and unstructured model output to ElasticSearch. A Maven-managed Java library **simulation_output_shipper** has been built which is used in our **ResultShipper** AnyLogic plugin. The plugin provides classes and methods to directly send data points to Kafka, with the need to configure Kafka properties and connections. The interface is simple and straightforward and mimicking visualization constructing process in AnyLogic. When the modelers want to add data visualizations such as line plots or histograms, the steps include: defining a dataset to hold



```
{
  "$schema": "https://vega.github.io/schema/vega-lite/v4.json",
  "data": {
    "url": {
      "index": "pmcmc-monitor-*",
      "body": {
        "size": 10000,
        "_source": ["@timestamp", "xval", "yval", "log.file.path"],
        "query": {
          "bool": {
            "should": [{"range": {"xval": {"from": 0, "to": 800}}}],
            "minimum_should_match": 1
          }
        }
      }
    },
    "format": {"property": "hits.hits"}
  },
  "mark": "line",
  "encoding": {
    "x": {
      "field": "_source.xval",
      "type": "quantitative",
      "scale": {"domain": [0, 800]}
    },
    "y": {
      "field": "_source.yval",
      "type": "quantitative",
      "scale": {"domain": [400, 1500]}
    },
    "color": {
      "field": "_source.log.file.path",
      "type": "nominal",
    },
  }
}
```

Figure 3.5: Real-time Visualization of Experiment Result with Kibana and ElasticSearch

the data points, and inserting the data points during the experiment. To use the data shipper plugin, after importing the plugin, an instance of the data shipper can be created by drag-and-drop. Properties such as experiment metadata can be added in the software's graphic interface. Whenever new data points need to be inserted into a visualization dataset, the sending function can be called with the x and y values, along with the dataset name. The plugin uses the **simulation_output_shipper** library to communicate with the rest of the streaming system. The message will first arrive to one of the Kafka instance, and then picked up and formatted by Logstash instances connected to the Kafka. The result is then sent to the central ElasticSearch. We use Avro serialization toolset and Confluent Schema Registry to facility the serialization and de-serialization of the Kafka messages.

3.3.8 System Availability, Exception Handling and Security

One important characteristic of many real-time continues systems is the ability to recover from failures and disasters. For a long-run process, it is imperative to handle errors and exceptions. The system here is designed for such events as well. The separation between data collecting and model make sure that errors and exceptions are contained. The Docker and virtual machine-based infrastructure provides flexibility on monitoring and replicating. Those individual components also have built-in such functionalities or can be obtained by pairing with solutions such as Apache ZooKeeper. Streaming systems as long-run services need to have minimal downtime and interruption to achieve desired availability. Apache ZooKeeper is one of the services for accomplishing the availability requirement across distributed services, such as Hadoop, Kafka and Storm, as in 3.4. It is a coordination service for distributed services, provides functions such as key/value storage, configuration storage, locks, synchronization and more. At the core, it is abstracted as a simple file system that supports common operations in a structure called **znode**. The master-slaves setup can be commonly found in distributed systems to manage the tasks; that is, one master node will be responsible for communicating and coordinating many slave nodes. A failed slave node can be spawned up and replaced quickly, but if the master node is down, the system will fail as a whole. Thus it is essential to have some master-selection system. ZooKeeper's synchronized locks will help the election of master nodes in cases where there are backup standby master nodes. Other services have built-in functionalities to achieve the same goal. For Redis, high availability is achieved by Redis sentinel. Sentinel and Redis Cluster are two replication solutions that come with Redis, but the former focuses on HA, while the latter uses partitions to increase the writing performance. On the running environment layer, Docker Swarm Clusters and Kubernetes Clusters provide a good interface for achieving high availability. They will consistently issue health checking commands to individual service, ensure the expected number of replications is observed, and use a similar multi-master structure to guarantee the clusters themselves are always available. Adopting Docker and Kubernetes enables us to quickly restart the process and application. Kubernetes will make sure there are desired amount of instances running at any time in a production environment.

By introducing persistent storage, history records can be safely stored. On the AnyLogic simulation model side, an interface has been provided. When starting a new run, a time series can be constructed as *existing data points* plus *future data points*. In the case of a failure, previous collected, computed, and stored data points can be used in that fashion to restart a new instance of the simulation model.

The security of the system is one of the most important aspects as well. The rich information in the multi-dimensional data collected can give us useful insights. However, it is always a prominent concern that the data mining on those data will cause personal data leakage. This is a great threat to personal privacy and can often be exploited by cybercriminals and cause even greater loss, whether financial, timely or emotionally. Thus it is deemed one of the most important issues for all parties to preserve and protect sensitive personal in formations while doing data mining and research. A clearly defined security policy is the foundation that other security elements such as audit and access control can be built on [108]. After identifying which level we need to secure the system, and then we can restrict the system using layers of security policy:

- Auditing and Logging: Logging of important events and errors is a common requirement of a system, and achieving it efficiently for a large system such as this one can be hard sometimes. The system is taking advantage of the Docker logging system, where the Docker daemon will save all the container output to files. Since each such container is always running one process, Docker essentially saves the logs for each system process, whether it is a data source adapter, the hub or other.

- User Access Control: Redis DB or the hub support user authentication. The Visualization web pages also need to be guarded by authentication, which is provided with ELK. This can make sure only approved models can consume the data.

- Networking and Firewall: TLS certifications are applied to the hosts to secure the data transfer. Separate instances of the hub can be deployed and each will be accessed only by authorized IP from the designated port controlled by container networking.

3.3.9 Containerization and Docker

It has become essential to design the tools that can take advantage of the distributed servers or cloud infrastructures because of the nature of public health analysis [35]. The current system can be easily deployed to cloud services, such as AWS, Azure, Google Compute Engine, and on-the-premise self-hosting; the system following the best practice by dividing the system into functional microservices and using container to manage and deploy. Containerization with Docker is an indispensable aspect of the system for both the development and deployment phases. It improves usability and stability and also simplifies the testing, extending and maintaining of the system.

Docker containers are incorporated into the development process for several reasons. First, there are ready-to-use Docker images for the components such as Apache Storm, Zookeeper, Cassandra, and Redis, allowing the quick setup of cluster infrastructure on a single developer machine. The environment can start and stop in minutes while guaranteeing reproducibility. The development process can save checkpoint - if new services or network has been added to the system, the setup is always described by the standardized Docker syntax, which also makes version controlling of the architectural setup possible. There are also containerized monitoring and visualization tools available – for example, by running the *redis-browser*[3] image side by side in one Docker compose environment, the values in the Redis server can be seen in real-time inside a browser. Second, it brings the benefit of easily sharing and communicating the design of the system. Docker Compose files are codes that specify all aspects of service deployment and serve as good documentation to communicate the full services stack. It will describe the images, applications, ports, data and volumes, network structures in one file. Docker Compose can describe the entire system in a single declarative configuration file and deploy with one command [109]. Also, Docker can help manage complex systems and avoid common compatibility and versioning issues. The image building process of Docker allows us to specify the versions of the software, libraries and operating systems base image to ensure verified working combinations. Docker Compose will deploy the system using specific versions of containers to guarantee compatibility is preserved. Third and the most important, the setup is portable - once developed with Docker, the resulting system can be deployed to large-scale production environments with minimal changes, such as to a Docker Swarm or Kubernetes cluster, or even HPC (High-Performance Computing) clusters that are not configured for YARN or Storm, with the help of containerization solutions such as Singularity. This leads to the advantages of using Docker containers to run the system in the production environment - deploying the system with Docker containers

[3]Redis Browser: https://github.com/humante/redis-browser

lets us enjoy features provided such as enabling automation, high availability, load balancing, restarting and recovering, easy updating with minimal downtime, comprehensive logging and monitoring and much more.

Introducing Docker brings an easy development process and can also guarantee to work on different infrastructures. It is lightweight and performs well, and it will shorten the developing process to get models with streaming capability up and running faster. Running many such containers only requests a small set of hosts with minimal configuration. The resilient nature eases us from designing complicated logic to handle exceptions such as power interruption or unexpected reboot, as we can quickly restore the server to the previous state.

3.4 Applications

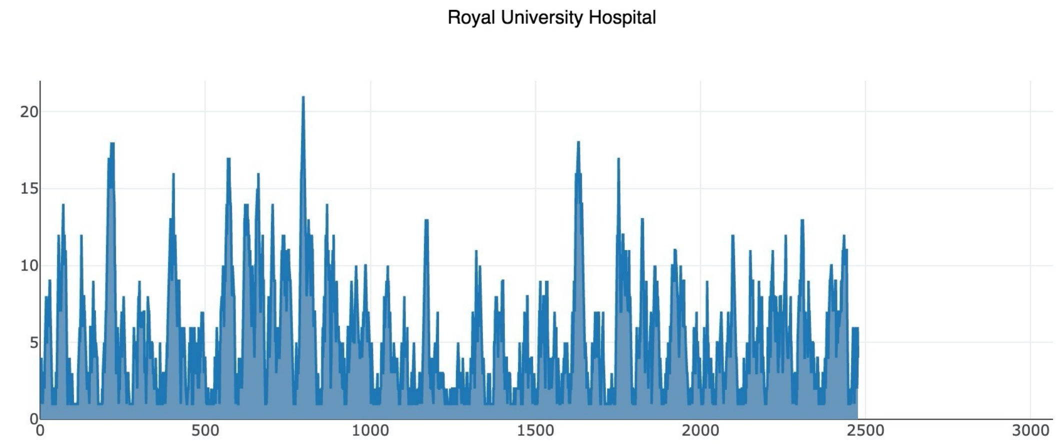
3.4.1 Saskatoon Emergency Department Waiting Time Model

As explored in [41], prediction on the availability of medical resources, such as emergency medical services, will help an individual patient make an informed decision quickly and benefit health care delivery system as a whole. The Saskatoon Emergency Department's Waiting Time model, built by colleagues in the CEPHIL laboratory, is a hybrid simulation model aiming to explore the efficiency of the local emergency health care resources, and can generate short-term predictions about ED resource usage. We built a data collecting adapter that can collect the waiting time data from the Saskatchewan Health Region (SHR) website. The model is then modified to connect to the live data stream and and can provide prediction in real-time.

Data Description The waiting time information is available on the SHR website[4] includes patients waiting, average wait and longest waiting time for ED locations across the city of Saskatoon, in the province Saskatchewan of Canada: Royal University Hospital Adult ED and St Paul's Hospital. The data will be updated every 15 minutes, and only the most recent data is shown on the website. One month of waiting patient counts was collected using the system I built and plotted in 3.6. For all locations, it is clear that there is a daily cycle pattern. To collect the data, we build a collecting adapter using Python with BeautifulSoup library. As the website is updated with fixed interval, the adapter is designed to check and download the HTML of the website every 5 minutes and only sends a new data point to the hub if the website is updated. The BeautifulSoup library is a powerful tool for parsing HTML and DOM manipulation and can locate the waiting time information from the downloaded webpage.

Model Description It is a compartmental model, with discrete-event components, to model how patients flow through the emergency department. The stocks, as shown in 3.7, are used to simulate each critical step for patients in the emergency room. The simulation model is built using AnyLogic. It has been previously

[4]Saskatoon Health Region, Emergency Wait Times, Saskatchewan Health Authority: `https://www.saskatoonhealthregion.ca/waittimes`



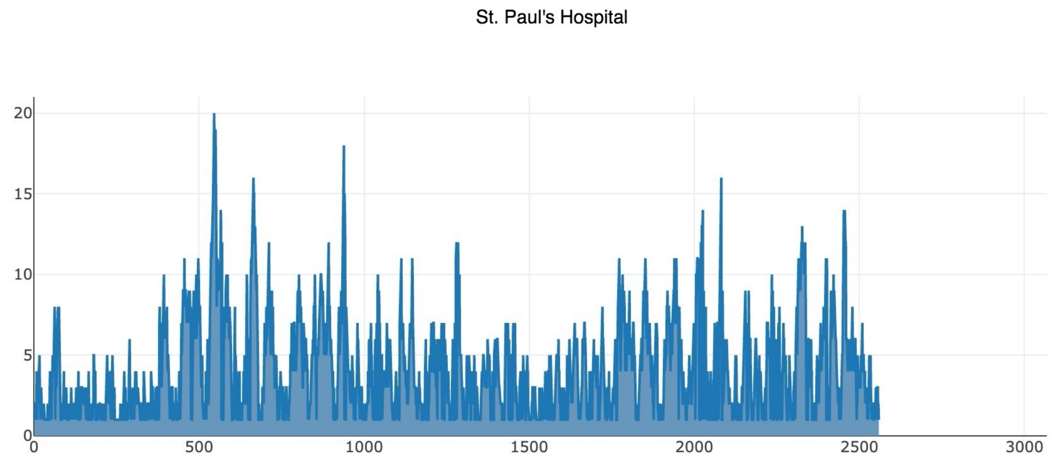


Figure 3.6: Saskatoon Emergency Department's Waiting Patient Count. Displaying results harvested from November to December for one month. Both open 24 hours per day.

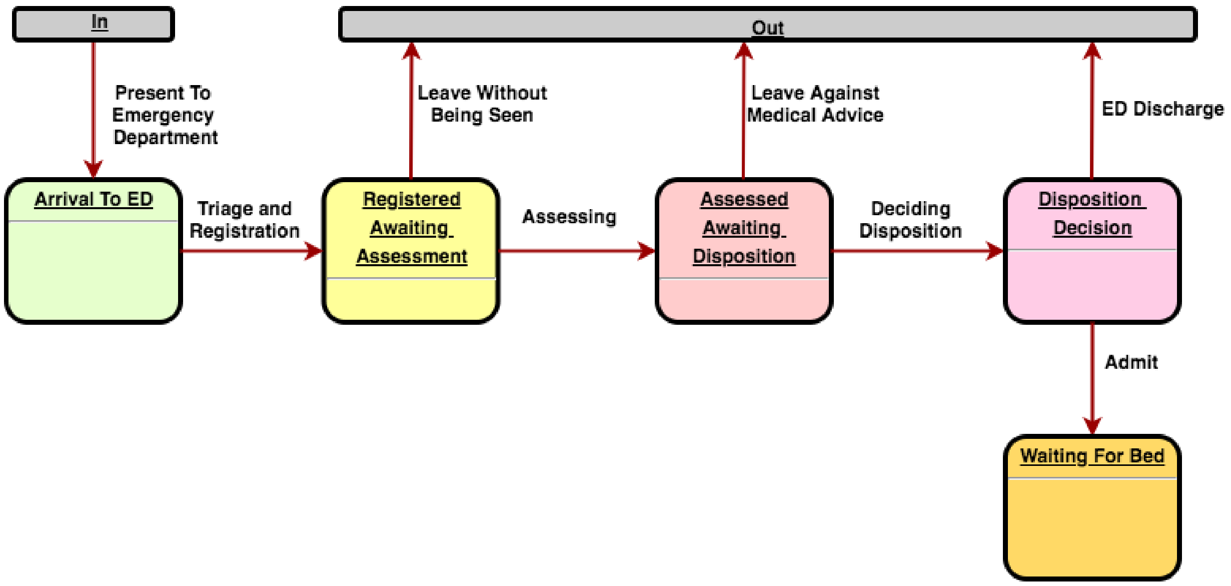


Figure 3.7: Stock-and-Flow Diagram for the ED Waiting Model

converted to use the empirical waiting time to compare the waiting time generated from the model with Particle Filter to ground the model. The waiting counts were harvested and saved to CSV files and read into the AnyLogic model previously. To use the streaming system with AnyLogic integration, we can change the data access section of the model to use the streaming plugin developed; specifically, this model we demonstrated here involves one line of code changed for accessing the live data, shown in 3.1.

```
// Original: Accessing the previously harvested data point from an AnyLogic table function
    for current time step
int iEmpiricalPatientsWaiting = roundToInt(tfPatientsWaitingCount(time()));

// New: Accessing using time index from a streaming  collection called
    livePatientsWaitingCount
int iEmpiricalPatientsWaiting = roundToInt(livePatientsWaitingCount.getByTime(time()));
```

Listing 3.1: Changes Needed for the AnyLogic ED Waiting Model When Connecting Streaming Services

Running Result The system was tested while running stable for 20 days continuously, with 10000 particles for the underline Particle Filter model. The testing system is Ubuntu 16.04 with Intel i7-4790k CPU, 32GB of RAM and SSD storage with AnyLogic 8. If plot the 2D histogram, for the entire time, the posterior, prior and empirical are in 3.8. If we zoom in, we can see clearly how the perception and prediction got updated every 15 minutes with a day. We can zoom in again to look at two steps to see the influence of an updated data point visually. Such updates in prediction about what is the waiting counts can be for each ED department can become precious for patient decision making and agencies monitoring and responding.

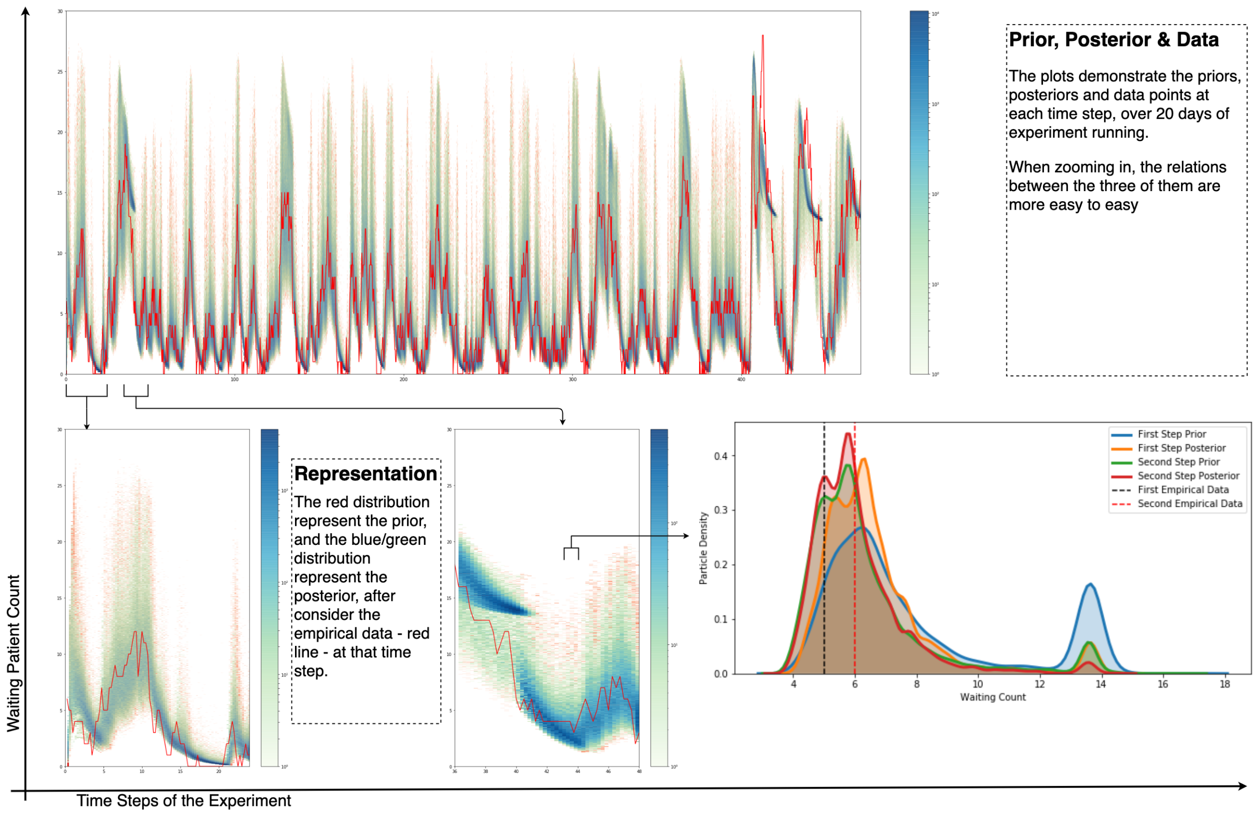


Figure 3.8: ED Waiting Model Running for 20 Days, Showing Prior, Posterior, and Empirical at each step.

3.4.2 Tumblr Harvesting using Apache Storm and Redis

This section presents a high-volume cluster-orientated streaming process application that harvests user posts and blogs on the social network Tumblr. The API only has functions for specific search endpoints. Apache Storm was chosen as the streaming framework because of the performance and flexibility. Tumblr is a social network website in the form of publicly available multimedia micro-blogs. Unlike Twitter, one user can create multiple blog spaces at a time for use for different purposes. Each of those blog spaces will have a unique name and a unique URL in the format of `{blog-name}.tumblr.com`. Tumblr has more than 500 million blogs, with 11 million blog posts per day[5]. Even though it primarily targets content creators and marketing-related content, the giant user base still makes Tumblr an excellent data source for public information-related research. We seek here to develop an efficient, effective and ethical method of harvesting the Tumblr posts continuously in real-time.

Tumblr API Following best practices in sharing user data through APIs, the collecting system should not request per-user permission to access user-to-user activities. For example, a blog's followers can only be accessed by a third-party application after the user grants OAuth permission to that application. Other information, such as blog posts, blog names and post timestamps intended to be publicly visible, can be retrieved through the API using API keys that do not require per-user permission. Tumblr restricts API access to 1000 requests per hour or 5000 requests per day for each registered application, a ceiling that can be waived upon request with the justification of the application's purpose. Videos, audios, and photos with captions, plain text, question/answer or quotes are allowed formats for Tumblr posts; there is no location information on a post unless the user has specified it in the text. Similar to Twitter, users can add multiple hashtags on a post, which can be used to group and classify. Unlike Twitter, there are no official endpoints for retrieving using queries and filters, nor third-party libraries close to Twitter4J that provide ready-to-use functions for streaming. We need to write our own customized harvesting/streaming platforms using Apache Storm because of all the above limitations.

Harvesting System We implement the real-time harvesting system as shown in Figure 3.9. At the start of the Storm system, it will write preset seed hashtags into Redis under a *tag-task* key. For example, if we are interested in the location-specified posts, we can use seeds such as *#Canada*. After initialization, the source bolts will take one of those seed hashtags and use it to build the API request to get a list of posts tagged with that hashtag and their blog names. Once the source spout receives those, it will parse and emit three different streams: it will parse and get all the texts of a post, if they exist, and emit those to post bolts; it will get all the hashtags from those posts and emit them to tags bolt – most posts will have more than one hashtag; it will also emit the blog names of those posts to the blog bolts. The blog bolts and tag bolts will test, using the Redis in-memory set, whether this is a new blog name or tag; if yes, it will push the blog

[5]Tumblr About: `https://www.tumblr.com/about`

name or the tag along with a timestamp to the Redis queue named *blog-task* or *tag-task* for source bolts to grab. Those task queues will proliferate so that the source bolts can make requests to Tumblr continuously. After the post bolts, functions can be added depends on the requirement. The post bolt can optionally test whether a post is a new one since we might get the identical posts through different tags if we want to process each post exactly once. Alternatively, we can do the count of keywords in those posts using counting bolts; or the post bolt can connect to a NoSql database such as HBase or Cassandra and store the harvested data for future analysis. The harvesting system's output is sent to the primary adapter for subscribed models to use in the simulations.

3.5 Evaluation and Discussion

We evaluate the system design from the following criteria: Scalability and performance, usability, maintainability and extensibility.

The system design refers to the Cloud Native best practice. The system can deploy quickly on all different machines. All the frameworks, software, and languages chosen are vendor-neutral and can run on private or public cloud infrastructures. Using containerization technologies and building the system with microservices are the most important ones of all the characteristics of Cloud-native application [110] and those are well adopted by this project. The system can further take advantage of service fabrics and orchestrating systems such as Docker Swarm or Kubernetes[56, 59]. The component-orientated design of the system has several exceptional advantages; first, it makes sure that the errors and exceptions are isolated and contained; second, we can replace, upgrade or modify individual component quickly; third, separation can eliminated redundant code, logic and instance to manage; rather, one data source is collected and processed once and can be shared by all the models.

The usability of the system is reflected by the following: When integrating new models to the system, the number of lines of code that need to be changed is minimal, and the user can choose to develop and test the model first before integrating the streaming system. The model needs to know only the 'channel' of the subscribed time series. Similar advantage goes to shipping experiment results to the central repository built with ElasticSearch. The AnyLogic plugins designed simply the integration steps by providing familiar interfaces to the model builders and developers. For processing data sources, new adapters are easy to develop following the interface proposed and the presented samples. With Docker it is easy to test and add the newly created data collecting adapters to the system and also taking advantages all the benefits offered by the rest of the system. The system can be fitted to many different use cases. It can manage the congestion of the streams. When the model is slower than data, the model side's queue will queue all the incoming data points. It offers different modes when integrating with models to facility different requirements, whether blocking or non-blocking.

On the deployment and operation side, because of the complete Docker-based configuration, we are able

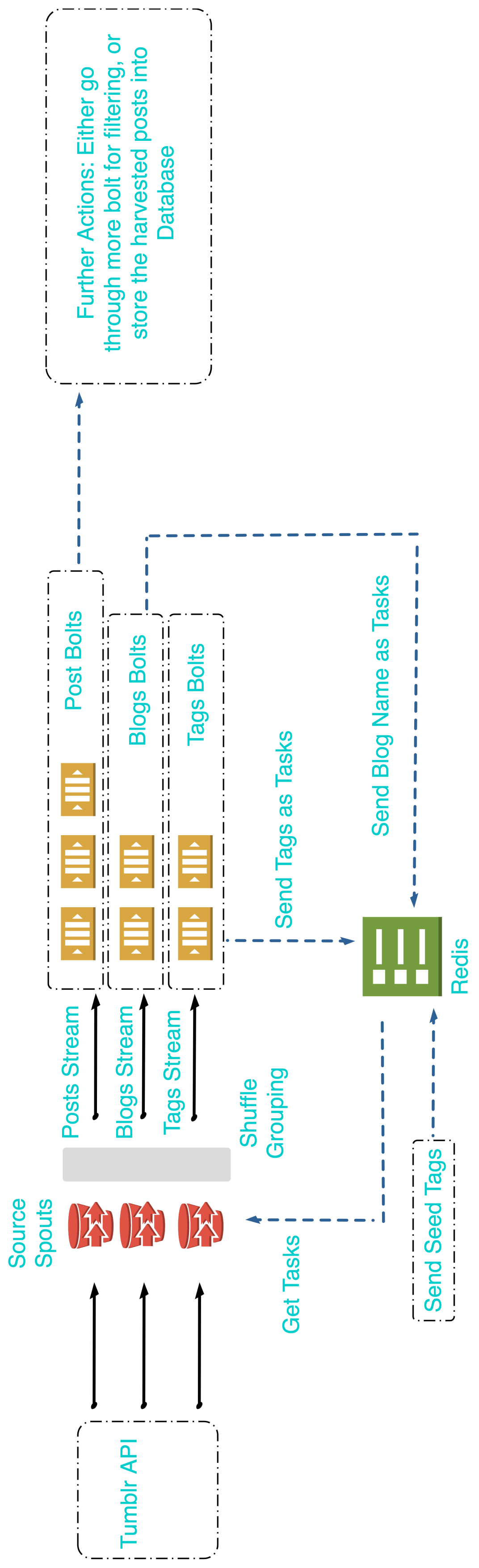


Figure 3.9: Tumblr Posts Harvesting Using Apache Storm

to go from implementation to production much faster. For testing, one docker file can spawn up the entire system. Also can deploy to the Kubernetes cluster for production. Mounted drives ensure disaster recovery. Monitoring uses ELK instance setup with another set of Docker containers. Since the front end is runner under Docker, it can enjoy all the high-availability features provided by Docker. Fault tolerance and disaster recoveries system such that most projects and data containers are stateless means that we need to only backup the storage. For long-run service, the mounted volume will be replicated and reused. Because of the Redis setup and accessible APIs for the adapters, the adapters can be reused later by another system easily without much effort. The loosely-coupled components enables easy customization, and individual components can be modified and updated based on different requirements.

The system can be extended from several directions in the future. First, even though the streaming system can handle data sources with changing time intervals between data points, the model and the algorithms usually take a constant amount of time to process each data point. A strategy is needed to detect if the model is behind the data and will not catch up unless the model reduces the time needed processing each data point or aggregates and skips specific data points. Second, while the system is designed for common failures and can recover quickly, there are cases where data points are not available from the source. Currently, this could prevent the models from progressing forward. To accommodate this situation, both the streaming system and the models need to incorporate logic to handle missing data points; for the model with Particle Filtering algorithm, the underlying model needs to proceed forward for that time step and skip the correction using data. Next, as seen during the COVID-19 pandemic, the officially reported case numbers and other statistics are subject to corrections; for example, case numbers can be adjusted to reflect missing or double-counted incidents days after initial release. Again, to properly handle this and make sure the models and system can progress forward while taking the corrected time series data, both the models and systems needed to be designed to roll back and start from a specific date or time again. All those are crucial for building the automated continuous reporting and modeling system.

In summary, the system designed fulfills all the requirements listed in section 3.2 and future extensions are needed for a more robust and flexible system.

3.6 Summary

This chapter provides a detailed description of the streaming system built for epidemiology simulation models with Bayesian statistic methods. The system fulfills the requirements listed at the beginning of the chapter and leaves room for various future extensions. The major contributions of this work include the follows:

- We review and summarize the current solutions on the topic of streaming processing in the field of public health and behavioural big data research, and give a detailed software requirement for streaming processing system for simulation models with Bayesian methods based on those;
- A broad set of software, tools and technologies are reviewed and compared, and we identify the key

features required for design such a streaming system;

- An implementation of the system is given, with two representative example use cases to demonstrate the system;
- We also provide the evaluation of the system from several aspects that are important to such design;
- Practically, we demonstrate how to use available tools to build high quality streaming solutions. The system takes advantage of Redis, Storm and other tools to create a streaming platform with minimal delays between when data become available and when the model receive those; the API-centred communication and Docker-based deployment make monitoring, scaling, and operating models much easier and highly customizable.
- The system enables the streaming simulations and experiments discussed in the later chapters of this thesis.

In the following chapters, we will discuss Particle MCMC with simulation models, and how the GPU-accelerated algorithm and the streaming system presented in this chapter will work together to provide a powerful inference toolset for public health and behavioural researchers.

4 Parallelizing Particle Markov Chain Monte Carlo-Enabled Dynamic Models Using GPU

4.1 Overview

Particle Markov Chain Monte Carlo (PMCMC) is a set of powerful contemporary Bayesian inference frameworks available to many inference problems. Compared to other related methods – such as Sequential Monte Carlo method of Particle Filtering (PF) – it requires a longer running time because of the iterative nature and computationally intensive PF process that forms one of its components. Moreover, in contrast to Particle Filtering, it is not as intuitive as to how to adapt PMCMC to work with streaming analytics with incrementally observed values. While there are many successful applications of the PMCMC method, those two are among the top practical challenges when using the method. In this section, I present an implementation of the method employing a System Dynamics state-space model that utilizes high-performance graphic processing units and a strategy of deploying PMCMC by reusing the previous run's samples in the subsequent PMCMC run with updated data points. Because of the parallelizable nature of the Particle Filtering mechanisms within PMCMC, the GPU implementation is anticipated to significantly decrease the execution time for simulations, and also enable the exploration on this strategy of incrementally reusing samples; the hope is that this PMCMC variation might decrease burn-in periods and speed the statistical convergence that is a central need for the MCMC component of PMCMC. We sought to combine such GPU-based computation and incremental sample use to shorter the simulation time even further and make inference results available sooner and incrementally.

There are several existing implementations of the Particle MCMC method; unfortunately, they all suffer from several common disadvantages. These include implementations in R designed to support easy interaction but not focusing on speed, and only usable for smaller models and simple inference tasks. Others, such as those implemented on GPUs and FPGAs for speed, are difficult to integrate with compartmental simulation models that are of central importance in health and community well-being applications of PMCMC. Prior to this thesis, the CEPHIL lab had designed and built a customized PMCMC library in the languages C and R that was specifically designed to work with compartmental System Dynamics models. Despite the performance gain from a highly efficient C-based implementation, running times measured in days are still frequently required for a reasonable number of particles and MCMC iterations. This component of the thesis sought to improve that customized codebase. While accelerating the code using GPUs, we will also need to retain the ability to efficiently work with System Dynamics models and use the resulting codebase in the streaming context to examine further how the streaming configuration can benefit PMCMC with compartmental models.

This chapter will introduce the problem and then present the implementation details of the solution. The exposition then continues to discuss the experiment and testing machine setup, followed by an application of the code on a System Dynamics model with performance and experiment results. In the next chapter, a more comprehensive examination of the implementation is presented, employing a more complicated simulation model combined with many experiments.

4.1.1 Previous Work

The work presented in this section is a continuation of the work of the CEPHIL lab in applying Particle MCMC algorithm on System Dynamics models. There were several foci of previous such work. First, Dr. Nathaniel Osgood designed and implemented an optimized C codebase serving as the engine to run PMCMC with System Dynamics/Ordinary Differential Equation [ODE] models. Reflecting a modular design philosophy, a binary interface was defined to allow the engine to be linked to user-provided modules implementing particular application-specific System Dynamics models and likelihood functions. While the computation is carried out in C code, the initial system further implemented a template in the R language providing the mechanisms for parsing inputs, parameterizing such models, collect results and perform basic analysis with plots and visualizations, and storing model output. Second, that codebase included a sample System Dynamics stochastic SEIR models, a variant of that defined in on [20], anddescribed in detail in the following sections. The integration between C and R is accomplished by the *.C()* interface, which can invoke functions in C code that is compiled using R's *SHLIB* command to shared objects and DLL[1] files for dynamic loading and loaded into R. This combination of R and C is aligned to most MCMC and statistics packages, where R provides an excellent ability for visualization and analytics, while C help ensure suitable performance and a streamlined memory footprint. The sample SEIR model demonstrated how to adapt models from equations. Several other smaller application-specific models were subsequently added to this codebase. The third component is a System Dynamics model developed collaboratively within CEPHIL and focusing on opioid prevention [7]. This is an age-grouped model, which means stocks are stratified to represent sub-populations based on the history of chronic pain and opioid use disorder, and opioid tolerance, resulting in 144 stocks in total. The model was developed and tested in AnyLogic as a Particle Filter model, and then translated to run within the PMCMC codebase as mentioned above. Experiments with that model demonstrated several critical insights about applying System Dynamics models with the PMCMC method. First, the C and R codebase is noticeably faster than AnyLogic's Particle Filtering even when running the PMCMC method sequentially. Second, the PMCMC algorithm is capable of supporting models of such complexity. Thirdly, it has been discovered that specific input parameters are crucial to the experiment's

[1]DLL: Dynamic-link library, shared library format used by Microsoft Windows.

model performance. Discovering the correct combination of input parameters means the model can offer much greater value when performing state inference and parameter estimation. Finally, the number of particles and MCMC iterations required each quickly exceed thousands and usually the more the better. The number of particles required for adequate sampling rises with the complexity of the model because more particles are needed to sufficiently cover the expanded model state space. For such a complex model, this combination usually results in days or weeks of execution time. To secure confident conclusions from such experiments, realizations employing the experiments' parameter settings often need to be run several times, with the results being summarized across multiple MCMC chains. The waiting times can be unbearable if any adjustment is required. Such observations highlight the critical importance of speeding up the codebase. This chapter's work starts from those solid previous contributions from CEPHIL, aiming to develop solutions to the problem so that additional future applications of similar simulation experiments can benefit from the accelerated PMCMC implementation.

4.1.2 Problem Analysis

Before describing the implementation of the CUDA accelerated PMCMC algorithm, it is crucial to assess the problem to understand the computation distribution in the algorithm and to analyze the potential for speedup. The algorithm's running time can be divided into two categories: The component that can be parallelized and the part that must be carried out sequentially. For the Particle MCMC program with compartmental System Dynamics simulation models, calculations are carried out by integrating ordinary differential equations (ODE). The basic structure of numerical inegration of ODEs – including the absence of branching, the presence of a simple global loop over integration steps, simple arithmetic operations – make it a strong candidate to run on each of the CUDA cores. We can choose to map particles to processors such that the numeric integration of one particle throughout the time period between observations carried out on a processor core, and no communication is required. We know that for a model with s stocks, n observation points, $\frac{1}{x}$ integration step, running with p particles for m MCMC iterations then the running time is $O(s \times n \times x \times p \times m)$; If we could parallelize over particles, then ideally we only need $O(s \times n \times x \times \max(\frac{p}{processors}) \times m, 1)$, decreasing the running time by the number of cores on the device, up to $\frac{TotalSequentialTime}{MaxTimeofanyParticle}$, since barriers are needed to synchronize at each steps of the computations of individual particle. The parts of the program that will not be parallelized in this work include computation steps for setup of chains, ancestry matrix construction, generation of a new candidate sample, and the acceptance rate calculation. These steps require less time compared to conducting the integration steps over all the particles, particularly when using very small integration timesteps. If we estimate that non-parallel code requires 2% of the overall computation time on the sequential version of the code, then by Amdahl's law [111], we have:

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{4.1}$$

with 1000 GPU processor cores, this will result in a 46.7 fold speedup under ideal conditions; doubling the count of processor cores to 2000 would increase the speedup to 48.8. Were the percentage of code requiring sequential handling responsible for just 1% of the computation time, the parallelization would result in 91.0 fold speedup.

The key difficulties of implementing such parallelization include the following:

- The probabilistic nature of the algorithm makes the verification of correctness of the migration hard, as the results will change for each execution. Such validation is also complicated by the fact that keeping random seeds fixed for reproducibility is not viable when random numbers are generated in parallel;

- It is essential to preserve the flexibility of the code in allowing the implementation to work with different simulation models. The facade pattern employed in the current CPU code makes the integration of new compartmental models easy and straightforward. The core part of the code to be moved to the device varies from model to model and needs to be kept straightforward. Moreover, the system secures much benefit through an R harness used to drive the system, and it is important that the R and C integration be supported with the introduction of CUDA.

- Specific to the programming stack, because the code needs to be able to be called from R, there is a need to compile the code using the CUDA compiler *NVCC*, with the results linked so as to generate the shared library files to include in R. Also, the device code will only be able to call the device code, and not generic functions in the R library. This means that all of the math libraries, including those from R's C headers, will need to be removed and replaced with a device version of the same functions.

- For the standpoint of the transparency, usability and evolution of the codebase, the structure of the code needs to be maintained. Specifically, each function's interface needs to be preserved.

4.2 Implementation

The development process has undertaken the following critical steps:

- Building baseline: Assess the current implementation and design test harness and function-level unit tests to help verify the code once starting the migration;

- Preparing the project for CUDA code: Investigate how to compile C/C++ and CUDA code to shared objects used by R, and also support to migrate individual functions;

- Implementing missing sampling and statistic functions on GPU, such as *rnorm*, and *dnbnorm*;

- Moving functions: Shifting functions to CUDA one at a time to measure the running time and to tune individual functions;

- Applying different kinds of strategies, such as exploring the use of Constant Memory and high-level CUDA libraries such as Thrust *reduce* to compute a sum quickly, and then *transform* to normalize the weights;
- Perform comprehensive experiments to evaluate the final results.

Computational Pattern Assessment

Application profilers can sample and analyze applications' execution time, which can help understand computational performance patterns. We selected the Linux *perf* utility to conduct profiling on the application before and after the migration from sequential to parallel implementation. For the PMCMC program, the primary bottleneck is that involving the computational operations, which means that the most crucial part of understanding is the CPU usage of the program, whereas I/O and memory usage are lesser concerns when assessing the computational pattern.

The flame graph [112] is one kind of visualization of system utilization sorted by function calls. This is different from other kinds of visualization. The X-axis depicts different functions, ordered alphabetically so that the same stack samples can be merged for improved visualization, and grouping the sampled stack traces, with repeated sampling of the same stacked trace combined to visualize the proportional time taken (denoted via colour). The Y-axis represents the stacks' depth, ordered from call graph ancestors at the bottom to descendants at the top. The purpose of visualization is to depict the frequency of each function in the stack trace samples to understand how the running time is distributed between the different components of the application. The assessment can point us to the functions in the codes that serve as the foremost bottlenecks and help check and assess any change made to the code by visually evaluating if the change offers serious prospects of materially reducing the running time of certain functions in the application.

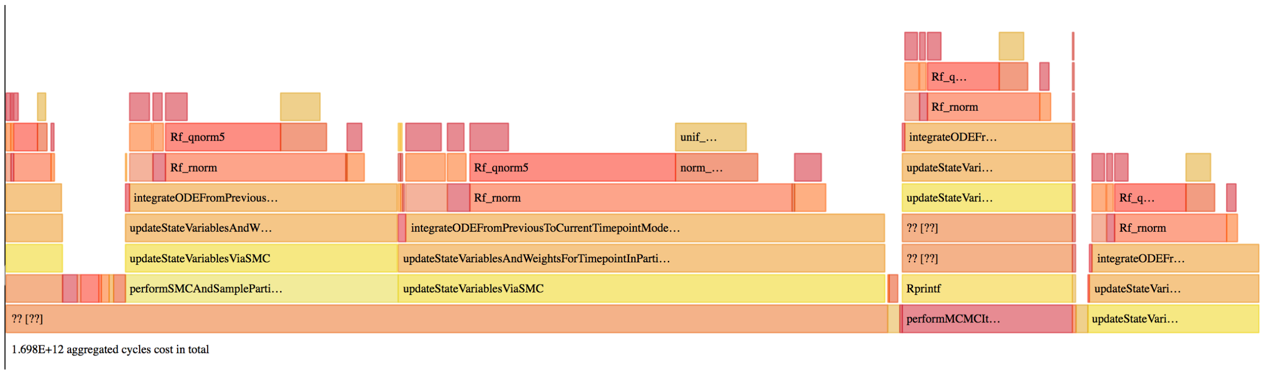


Figure 4.1: Flame Graph of *perf* Profiling Results for CPU Runs

In Figure 4.1, we can see that one of the most time-consuming functions is the one that integrates the ODE from the previous observation time point of the empirical time series to the next observation time point. This is the function that calculates the stock and flow values over time, as governed by the equations of the ODE. When calculating the current value of the model states, besides all the stocks and flow values, the model also needs to vary specific dynamic parameters undergoing random walks characterized by normally distributed perturbations; those are achieved by the calls to Rf_rnorm, which uses the Rf_qnorm5 routine to generate samples following a normal distribution.

Prepare for Migration to GPU

A set of utility functions was developed to systematically invoke part of the code acting as function level tests to achieve better control, correctness checking, and measurement. These include:

- A serialization utility that can save and load experiment output for reproducible simulations. This serialization utility act as one of the main methods to verify the migration's correctness;

- A set of testing harnesses. For R, we use the library *testthat*. R is also used to call the C function *miniunit* to run C-based unit tests. An important element of the solution chosen here is a conditional compilation with a different version of the source code, so as to produce different compiled codebases for testing and production. For the testing version of the codebase, mock versions of functions such as those random number generators are provided in a separate file. A simple way to support such mocking is to let the random number generator always generate the same number. Though this can work for some functions, it can not test other functions, such as those that involve distributions and sampling from a set of particles. To be able to simulate the random number generator while maintaining the same result as on a CPU, a possible solution would be first using a regular number generator to generate a table and then set up the CPU to read this table sequentially. To set up a similar lookup mechanism on GPU can be challenging; since GPU will call all the random number generators in parallel; offering such functionality would require additional mechanisms, such as by assigning lookup keys to the table and making sure different threads pick up the correct values.

Heterogeneous Compilation and Execution

NVIDIA's CUDA compiler, NVCC, is a C/C++ compatible compiler based on LLVM that enables compilation of device code and host code. Similar to the C/C++ compiler, it also supports external library files, such as the R library needed in this work for R integration. Previously, the *R SHLIB* command can handle the compilation and linking of C/C++ code and generate shared libraries to use with the *.C()* interface. The CUDA code will need to be compiled to the shared library first. The R SHLIB command will then be used to link the shared library needed to run the CUDA version of the algorithm from R. Retaining the R binary interface in this way also makes debugging with CUDA-GDB possible by initializing R's debugging session and with CUDA-GDB serving to attach to different CUDA threads within the GDB session.

Reflecting the fact that the above process uses two sets of code targeting different kind of processors, this work put several components in place to simplify the integration of new models and reduce the chance of errors:

- For the functions and files that required modification, the same structure and layout are retained. For the part of the code that is same across both kinds of processors, we separate such code into source files and shared between the two codebases.

- To simplify the process of sharing variables between R and C/C++, we develop a source code processing utility to generate R scripts based on specific C/C++ header files using pattern matching. This setup can help minimize duplicate code and reduce human errors.

- We further extend this to support loading different experiment scenarios using header files. The resulting program can compile different header files and generate the model with different experiment parameters. This mimics the functionalities commonly found in other simulation software that separate the definition of a model from that of experiments, such as AnyLogic.

Thread Synchronization

Starting from the central computational task of the algorithm, the first step of the GPU implementation involved moving the differential equation calculation of the PF process to the GPU. Using the mocked data setup, it is possible to test this one function's GPU and CPU versions side-by-side. While results presented later in this thesis will examine performance under different models and in the context of the broader system, the anecdotal results from one experiment were encouraging: In the context of a medium-sized model's ODE, when integrating over one time step with 16384 particles, the CPU takes 18.6 seconds. By contrast, only 0.194 seconds are required for GPU with 1920 CUDA cores – a 95 fold improvement for this core function on our test machine. The next step consisted of moving the likelihood calculation, which is executed for each particle at each observation point. This task is quite straightforward; because particles do not interact during numerical integration, no inter-thread communication or synchronization is required. A key step requiring cross-thread synchronization within PF (and, by extension, PMCMC) concerns the additional components of the observation step, during which weights must be normalized, and resampling performed.

One common challenge of multi-thread parallel computing lies in handling inter-thread communication when data exchange is needed between individual tasks. In CUDA GPU programming, the communication can be achieved using shared memory spaces and thread synchronization as an alternative to direct message passing. Each thread can modify values on the same shared memory; by calling $\_\_syncthreads()$ in the device kernel function, threads of the current function will block here until all the threads reach this same position. Then, it is safe to access the modified values from the memory to accomplish the inter-thread communication.

In implementing PMCMC with System Dynamics models, we identify one place where such communication and thread synchronization is required. When migrating the resampling step in the original (CPU) code, we need first to perform a draw from the multinomial distribution based on normalized weights to get the counts in each bin. Having performed the random draw, we remember the sampled particle index for each particle and apply a random permutation; the random permutation implementation is based on uniform distribution and quick sort. To accomplish this step efficiently on a GPU, we need to do a parallel multinomial draw for sampling; after each thread on the GPU conducts a draw of a single sample from the distribution – essentially, picking an index of the source particle (ancestor) that will occupy this particle position going forward. Having identified this particle, this thread can simply copy the ancestor's state variables. Also, since this is for PMCMC – for which accumulation of an ancestry matrix is essential to reconstruct trajectories – we need a separate device memory to store the ancestor's id along the way. Since we will perform a separate individual multinomial draw on each CUDA thread, we will get particles in a permuted order by default, an explicit permutation step is not required, and each particle need only record the ancestor's id. CUDA offers only essential random number and statistics utilities, and we need to implement the others by ourselves. For instance, we can obtain uniformly distributed random draw within an arbitrary range specified by an ordered pair of double precision numbers by straightforward scaling of the result of the CUDA function *norm* that generates numbers between 0 and 1. Functions such as the calculation of negative binomial distribution used by the likelihood calculation were created using the implementation in the R project as reference. The multinomial sampling function needed by the resampling step of the MCMC process was modified based on a previous GPU implementation of the same function[2]; while the original implementation returns n samples at a time, here it is modified to generate a single sample per invoking. After weights have been normalized and indices of the particles to carry over to the next time step have been decided following resampling, a swap of particles will copy all the state variables of the parent particle to the new child particle locations. This process involves copying the parent state variables to a temporary location and then modifying the array of state variables. A synchronization call is needed to make sure all *READ*s have been finished before any *WRITE*s to the array. Since the swap step is carried out in parallel on the GPU, it is still much more efficient than non-parallel modifications of the data array iteratively on the CPU.

Memory Management

Memory management is crucial for GPU programming, and there are distinctive features of the GPU programming context when utilizing the device memory. There are several levels of memories available on GPU compared to the CPU memory hierarchy. There are global, shared, constant, texture and local registers. An excess amount of local registers required by a kernel function will reduce the kernel execution's parallelization. The heterogeneous programming model means that the memory management operations can be expensive. Such operations include allocating memory, copying variables between the device and the host, and reading by GPU chips from GPU memory or cache. Using the correct memory at the right time would reduce the memory overhead and could dramatically decrease the overall execution time.

Whenever possible, we should reuse memory allocation and minimize memory operations; we investigated the trade-offs between memory allocations and memory copying when achieving both simultaneously is impossible. We only allocate the memory on the GPU device at the start of each MCMC iteration and copy

[2]A GPU-based multinomial resampling function is provided by former member of CEPHIL Koushik Pal.

Table 4.1: Assessing Improvement from Combining Memory Allocation and Copy

Combine Memory Allocation Copy Assessment	Influenza Model, 512 particles, 50 PMCMC iterations, 77 observations, 0.01 integration time steps		
Machine	D2	D1	Kepler
Allocate and Copy Only the Needed Observations at Each PF Step	5.72	8.22	13.92
Allocate and Copy the Entire Observation Matrix Ahead of Time	5.59	6.33	13.26
Improvement	2.3%	5.3%	4.7%

it back to the host CPU at the end of that MCMC iteration. Even though that means we need to carefully handle the device memory pointer while maintaining the modularity of functions, other System Dynamics models can easily migrate to the GPU codebase.

Since each PF step would only read the empirical values observed at the current time step, naturally, we would seek to reuse the same allocated memory across all PF steps and then copy the values to the device. Alternatively, we can allocate the entire observation matrix and copy the data over at the start. We would then trade performance with space, as combining memory copy actions can better utilize the host-device communication bandwidth. In Table 4.1, we assess the benefit of allocating all memory for the observation matrix, compared copying to smaller allocations at every Particle Filter step, using testing machines from Table 4.3. We observed 2.3% to 4.7% performance improvement by allocating all memory for the observation matrix up-front compared to repeated copying from the reused allocated memory.

Using Constant Memory

In the context of high-performance computing, locality of accesses to memory are of vital importance, since moving data is a very high-cost task. The CPU programming model offers a flat memory structure where the hardware manages the actual memory mapping. By contrast, as noted above, GPU program model does not employ a flat memory structure. Instead, it offers several types of memory for explicit use by developers. Constant memory lives in the device DRAM but is cached in several layers. In other words, once fetched, the read-only constant memory is stored very close to the processing units, and the cost of reading from constant memory is minimal, similar to reading from per-block shared memory. The empirical observation

Table 4.2: Assessing Improvement when Using Constant Memory for Observation Array

Constant Memory Assessment Setup	Influenza Model, 512 particles, 50 PMCMC iterations, 77 observations, 0.01 integration time steps		
Machine	D2	D1	Kepler
Without Constant Memory, Running Time in Seconds	5.53	6.21	13.21
Observations on Constant Memory, Running Time in Seconds	5.52	6.20	12.77
Improvement	0.18%	0.16%	3%

array would be a perfect candidate to move to the constant memory: Firstly, no modifications are required for the observations; secondly, all particles are trying to access the same observation at the same time point of the entire time series; finally, because all the particles need to access the observations, it is better to use constant memory than shared memory or texture memory. Thus it is better to use constant memory than shared memory or texture memory. The downside of the constant memory is that it can only store 64KB for the entire device. But for many applications of PMCMC, such a block of memory will offer sufficient space to store observations. In table 4.2, we listed the assessment results for moving the observation array to device constant memory from global memory. As a result of this shift, we observed a 0.16% to 3% performance improvement. While this is relatively small because of the smaller length of the observation array used, it bears emphasis that the observation is a relatively small array. The allocation and initialization only happen once for the entire PMCMC chain. The per-MCMC-iteration overhead will also diminish with a longer chain.

GPU Accelerated Library

While CUDA provides many essential functions for common tasks, many libraries are built on top of CUDA and provide implementations of additional functions that can be used together with CUDA. These include ArrayFire, which provides data structures and functions for array and matrix manipulations, and Thrust from the CUDA toolkit. Thrust is designed as a C++ template library and functions are implemented on standard containers of ADTs (Abstract Data Types); it offers operations such as sort, scan, and transforms on containers. Those can simplify the implementation process and support good performance and easily maintainable code. This approach encapsulates implementation details, removes the need to tune the kernel functions and launch configuration. This allows us to calculate without the need to design CUDA kernel functions explicitly for common tasks, such as sum, reduce, or map functions to members. For data intensive tasks, Thrust has been show to perform close or surpass manually implemented and tuned low level CUDA implementations, especially on newer CUDA devices [113]. In this work, we choose Thrust to calculate the sum and then normalize all particles' weights to take advantages of the existing implementation and the familiar C++ STL interface. This uses Thrust to calculate the sum of all the particles weights, instead of implementing our own pair-wise reduce, such as in [75]. A Thrust device vector can be created from a CUDA device memory pointer directly; once we have calculated and received the unnormalized weights in parallel using kernel functions, we create the Thrust vector of the unnormalized weights. Then we can switch to using Thrust functions directly, without the need to copy the memory between the device and the host. To obtain the normalized weights, we first use Thrust reduce with *plus* operation to get the sum of the unnormalized vector, then fill another equal length vector with the (identical) sum value and apply Thrust transform with *divides* operations. When done, we convert the Thrust vector back to CUDA memory pointers using Thrust *copy*.

Instead of fully relying on higher level libraries, we write CUDA kernel functions for other steps. This gives us better control of the implementation and helps to achieve the best possible performance. Writing kernels functions also retains the familiar interface designed to enhance transparency of the system design in terms of its computational statistics underpinnings, and code modularity and testability.

4.3 Experiments and Testing Environments

4.3.1 Overview of Experiments

The primary goals of experiments in this section is to evaluate the improvements in performance secured with the GPU compared to the CPU implementation, and to experiment with and test a novel adaptation of PMCMC for streaming. Responsive to these goals, experiments were be carried out in two main directions.

Firstly, one set of experiments sought to compare the performance between the accelerated GPU code and the original CPU code. The comparisons covered various devices and different ODE simulation models varying in terms of model structure and complexity. Such experiments assessed the performance impact and aided understanding of patterns of dependence of performance on resources and how the code utilizes the available hardware resource. Such assessments included running the same setting with GPUs resources in different contexts: With low to middle-grade workstation GPUs, with GPUs in servers including cloud providers, GPUs in high performance computing environments, and with different operating systems, and with other hardware such as RAM, disk, and CPU. By showing the running time with the original CPU version of the algorithm and the new GPU implementation on each device, we sought to see which variables exerted the most significant effects and if bottlenecks could be avoided. We will then run different settings on the same machine by varying parameters such as the count of MCMC iterations, particle count, and others that can influence the algorithm's execution speed. By conducting such experiments, we sought to examine the scalability of the implementation, and to try to find the pattern and make a prediction about the potential improvements of GPU version over CPU version were more powerful GPUs to become available.

The second set of experiments instead sought to first perform inference on synthetic data, and then on real-world observation data. Two models were examined in this area, and were used to explore the effects of using online streaming Particle MCMC with varied time-series length. By creating a simple model with synthetic data as input, we have an opportunity to test the performance of the PMCMC algorithm with System Dynamics models knowing the true values of the MCMC parameters. If we supply the true value of the MCMC parameters as the initial point of the MCMC chain, it would essentially examine the ideal situation where very rapid convergence is achieved for the chain. These experiments sought to use this observation to can compare the algorithm's performance with and without streaming under such ideal conditions. Beyond such "synthetic ground truth" experiments, other experiments sought to examine the effects of streaming PMCMC in real-world applications by examining the behaviour of complicated models in the context of empirical data.

4.3.2 Testing Environments

This work conducted experiments to assess performance improvements, better understand the computational patterns, and investigate hardware utilization. By comparing the running time between the original serial version of the algorithm and the new GPU-accelerated parallel implementation in different testing environments, we sought to identify variables impacting the running time most significantly and locate bottlenecks. We also sought to illuminate the scalability of the implementation by varying parameters such as the counts of MCMC iterations and particles, identifying the pattern and anticipating the potential improvements of GPU versions over CPU versions if more powerful GPUs were to become available. The hardware specifications are important factors moderating performance improvements and need to be considered when seeking to understand the results. As shown in Table 4.3, test computers and environments were selected to represent a wide range of machines, including CPU-, memory-, or GPU- bounded single computers, but also high-performance computing environments and remote servers. A MacOS-based computer was also included to demonstrate the cross-platform property of the CUDA codebase with R and C. A cloud machine was further set up on Microsoft Azure. It is an NC6 Promo [3] instance, which is an entry-level cloud GPU node provided on Azure, with an estimated monthly rate of CAD\$380 if runs 7×24. We also listed the key hardware specifications for all the GPUs used on those machines in Table 4.4. The GT and GTX series of the NVIDIA cards are built for gaming and everyday tasks, while the Tesla series are for workstations, servers and clusters, specifically. While 4 to 8 GB of onboard memory is usually enough for gaming, Tesla cards would generally have more, sometimes doubling the onboard memory comparing to the GTX cards of the same generation. Tesla cards are also equipped with a more advanced memory type, such as HBM2 (High Bandwidth Memory) offering much higher bandwidth. Combining the larger space with higher bandwidth, those GPUs can achieve faster host-device communications – such as is required when copying objects – and also hold more objects at once.

[3]https://azure.microsoft.com/en-ca/pricing/details/virtual-machines/linux/

The experiments associated with such diverse hardware were designed to support a solid understanding of the accelerated algorithm's performance pattern by covering a varied set of hardware resources.

It has been common for computation-intensive research to move to remote servers such as supercomputer clusters and cloud providers. Microsoft Azure has virtualized GPUs with PCIe (peripheral component interconnect express) pass through, allowing virtual boxes created on Azure to directly access shared GPUs. Note that even though High Performance Computing (HPC) clusters such as Compute Canada are also shared machines, they use Slurm [4] managed direct access to GPUs. Slurm can assign individual GPU to the task so that the task can use the entire GPU without the need to worry about resource competition, or the noisy neighbor effect. Experiments performed in this work with Slurm-managed Compute Canada nodes are carried out by requesting one GPU per task. We are interested in virtualization and its effect on performance because it brings many benefits for applications similar to those explored here. [5] According to Microsoft Azure documentation, the node type we chose is optimized for compute-intensive and network-intensive applications and algorithms.

When performing testing, the environment were such that no other application uses a notable amount of GPU processing cycles or memory other than the necessary UI rendering of the desktop environment. Code was compiled with CUDA 10.1, R 3.6.0, using *clang++* on macOS and *g++* on other systems according to the C++11 standard. Testing results are gathered and plotted using Jupyter Notebook with R and Python.

4.4 Accelerating a Streaming SEIR PMCMC Model with CUDA

We present here a simple SEIR model to illustrate the combination of System Dynamics modeling with the PMCMC algorithm; the model is simple, involves minimally complex stochastics and can be used to help ensure that the implementation is correct. It also serves the purpose of setting up the baseline for benchmarking and performance testing on a variety of hardware or as a point of comparison with other implementations and libraries. As mentioned in earlier sections in this chapter, this simple model has already been used in the implementation process to measure and assess tradeoffs between different possible individual implementation decisions, such as with respect to the type of memory to use. The above sections have discussed how the Particle MCMC method can be used with compartmental models and how to speed up such experiments using the general-purpose GPU and CUDA programming paradigm. In this section, a simple SEIR System Dynamics model will be used to demonstrate how to work the PMCMC CUDA code, measure and characterize the performance improvements, and illustrate how the speedup enables the streaming inference. The final component of this section examines the impacts of using streaming online learning with PMCMC. The model discussed here is an extension of that examined in [20]. In that paper, an agent-based model (ABM) was set up to generate synthetic ground truth data based on different network connection types and chosen

[4]Slurm Workload Manager, https://slurm.schedmd.com/overview.html

[5]Microsoft Azure Documentations, https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-gpu

Table 4.3: Testing Machines

ID	Name	OS	CPU Type	System Memory	Disk Type
1	D1	Ubuntu 18.04	Intel i7-4770k @ 3.9GHz	32GB	SSD
2	D2	Ubuntu 18.04	Intel i3-6100 @ 3.7GHz	32GB	SSD
3	D3	Ubuntu 18.04	AMD A6-5400K @ 3.6GHz	16GB	SSD
4	iMac	macOS 10.13	Intel i7-3700S @ 3.9GHz	16GB	SSD
5	Cedar	CentOS 7	Intel Xeon E5-2650 v4 @ 2.2GHz	Vary	SSD
6	Graham	CentOS 7	Intel Xeon E5-2683 v4 @ 2.1GHz	Vary	SSD
7	Beluga	CentOS 7	Intel Gold 6148 @ 2.4GHz	Vary	NVMe SSD
8	Kepler (K1, K2)	Red Hat 4.8.5-39	Intel Xeon E5-2620 v2 @ 2.1GHz	24GB	HDD
9	Microsoft Azure NC6 Instance	Ubuntu 18.04	Intel Xeon E5-2690 v3 @ 2.6GHz	56GB	SSD

parameter values for the contact rate per week. This ABM was used to periodically produce the ground truth count of individuals who had transitioned from the latent state to the infectious state since the last such report. Based on such data, random noise reflecting false positives were added to the count, thus acting as the measurement noise around ground truth and constituting the empirical observations. Then a Particle Filter SEIR model was built to compare with traditional calibrated model in terms of prediction discrepancy. While it was mentioned in the previous sections, it bears noting that the C version of this model that was used with PMCMC to estimate the contact rate was built previously by the original author of the C PMCMC codebase as an accompanying sample and test model for that codebase. The work here moved the same demonstration model from the C CPU version to the GPU version, modified the agent-based model to generate a long time series, and used it to test the GPU SEIR model in the streaming context.

This section is organized as follows: The SEIR model will be characterized, including the synthetic data generating model. This section next illustrates the changes undertaken to adapt this model to use of GPUs. That discussion is followed by discussion of performance measurement in terms of execution speed. Next, the section continues on to show the work conducted to enable the PMCMC code to work with the streaming platform introduced in the earlier chapter, and presents how the PMCMC performs in a streaming capacity with the SEIR model acting to operate on new data points becoming available.

Table 4.4: Testing Machines GPU Specifications

ID	Name	GPU Type	GPU Memory Size	GPU Memory Type	GPU Memory Bandwidth
1	D1	Nvidia GTX 1050 Ti @ 768 cores	4GB	GDDR5	112GB/s
2	D2	Nvidia GTX 1070 @ 1920 cores	8GB	GDDR5	256GB/s
3	D3	Nvidia GTX 1050 Ti @ 768 cores	4GB	GDDR5	112GB/s
4	iMac	Nvidia GT 650M @ 384 cores	0.5GB	GDDR5	80GB/s
5	Cedar	Nvidia Tesla P100 Pascal @ 3584 cores	12GB - 16GB	HBM2	549GB/s - 732GB/s
6	Graham	Nvidia Tesla P100 Pascal @ 3584 cores	12GB	HBM2	549GB/s
7	Beluga	Nvidia Tesla V100 @ 5120 cores	16GB	HBM2	900GB/s
8	Kepler (K1, K2)	Nvidia Tesla K20c @ 2496 cores	5GB	GDDR5	208 GB/s
9	Microsoft Azure NC6 Instance	Nvidia Tesla K80 @ 4992 cores	24 GB	GDDR5	480 GB/s

4.4.1 Model description

A simple SEIR model, such as that shown in 4.2, is a type of infectious disease modeling that can be used to characterize and simulate disease epidemiology within a population, by subdividing the population into susceptible, exposed, infectious and recovered stocks (compartments). A healthy individual will start in the susceptible state and later get infected to the disease with a certain probability after contacting with an infectious individual, thereby moving to the "Exposed" compartment, representing a latent state in which a person is infected but not yet infectious. People in the exposed and infectious states will move to the next state after a certain mean length of time, based on the natural history of the disease The SEIR model is a quantitative model and the dynamics of the system can be described using the following state equations:

$$\begin{aligned}
Susceptible :&\frac{dS}{dt} = -(\beta \frac{I}{N} c)S \\
Exposed :&\frac{dE}{dt} = \beta \frac{I}{N} cS - \sigma E \\
Infected :&\frac{dI}{dt} = \sigma E - \gamma I \\
Recovered :&\frac{dR}{dt} = \gamma I
\end{aligned}$$

If we further consider an SEIRS model, where immunity of the disease will disappear after a certain mean period, then the state equations for the susceptible and recovered stocks will be modified as follows:

$$\begin{aligned}
Susceptible :&\frac{dS}{dt} = -(\beta \frac{I}{N} c)S + \mu R \\
Recovered :&\frac{dR}{dt} = \gamma I - \mu R
\end{aligned}$$

System evolution in such models is deterministic. Introducing stochastics, the infection process can be modelled according to a Poisson process; under that case, the state equations for the susceptible and exposed stocks are given as follows:

$$\begin{aligned}
Susceptible :&\frac{dS}{dt} = -\frac{Poisson((\beta \frac{I}{N} c)S\Delta t)}{\Delta t} + \mu R \\
Exposed :&\frac{dE}{dt} = \frac{Poisson((\beta \frac{I}{N} c)S\Delta t)}{\Delta t} - \sigma E
\end{aligned}$$

While the contact rate is one of the most critical parameters in this category of models, it can change from time to time during a pandemic, on account of the impact of interventions such as public health orders, risk perception-driven changes in mixing, and other factors. It is sometimes required to use varying contact rates with respect to time and other external factors to form a time-dependent dynamic models, such as in the method presented by B. Tang et al. [114] for simulating and predicting in the initial stage of COVID-19 in characterizing the increasingly restrictive interventions and public health responses, such as those resulting

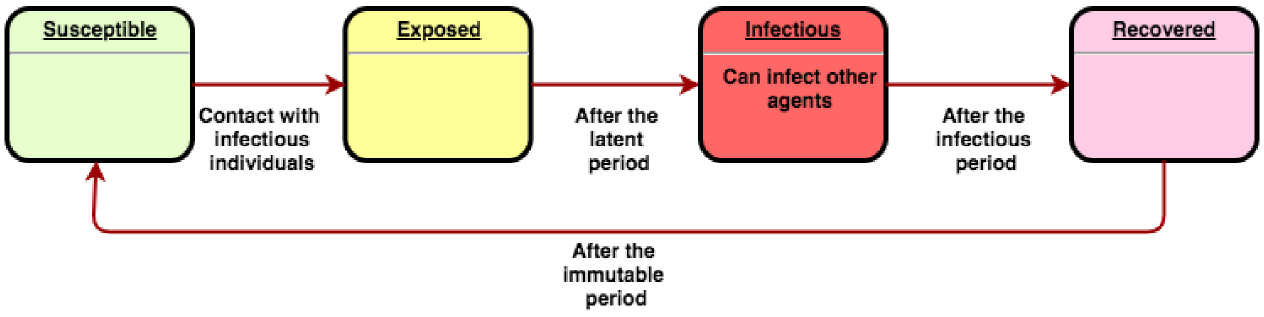

Susceptible
Exposed
Infectious
Can infect other agents
Recovered
Contact with infectious individuals
After the latent period
After the infectious period
After the immutable period

Figure 4.2: Susceptible-exposed-infectious-recovered (SEIR) model

in the lockdown of cities and forced quarantine of communities. Be able to use an appropriate contact rate for the model is of great importance for understanding the disease dynamics and supporting good insights and predictions from the model. Using empirical statistics and the model can also help us make inferences with respect to that important parameter and understand the effectiveness of interventions and other response measures.

Under similar assumptions about the epidemiology of that infectious disease, an agent-based model can be built to mimic the same system that is depicted in the SEIR model. Such an agent-based model characterized the evolution of each member of the community with individual variations being captured via model statistics. The agent-based model will utilize a state chart such as that depicted in Figure 4.3 to simulate contact between individuals. Exposure and transmission of pathogen in the model is normally achieved by inter-agent messaging, whereby individuals in the infectious state will periodically send the 'virus' message to others in their networks. Using the same parameters as the compartmental SEIR model, the agent-based model will result in somewhat similar output, but in a way that exhibits stochastic variability.

Instead of using the original SEIR model included in the PMCMC codebase, we modified both the compartment model and the ABM to produce corresponding *SEIRS* models, such that the simulated infections will not go extinct. This way, a long time series can be generated to test the model, mimicking the streaming input in the context of other endemic communicable diseases, such as seasonal influenza. The parameters used to generate ground truth data in the SEIRS ABM model are listed in Table 4.5. Similar to the process described in [20], measurement noise is added as follows: We assume that with a certain likelihood, some incidents case will be reported while others are missed, and there are certain falsely reported cases (false positives). The likelihood of individuals other than cases being reported per week and the likelihood of incident infecting cases being reported per week are used to capture such measurement noise. The experiment was carried out as follows: First, synthetic data was generated by the SEIRS ABM model and analyzed. Second, we set up the PMCMC code base with the SEIRS model to run with those synthetic data. Third, the inference results were evaluated with the parameters used to generate the data, while also explore the performance pattern of the GPU accelerated code base and the effectiveness of streaming PMCMC configuration.

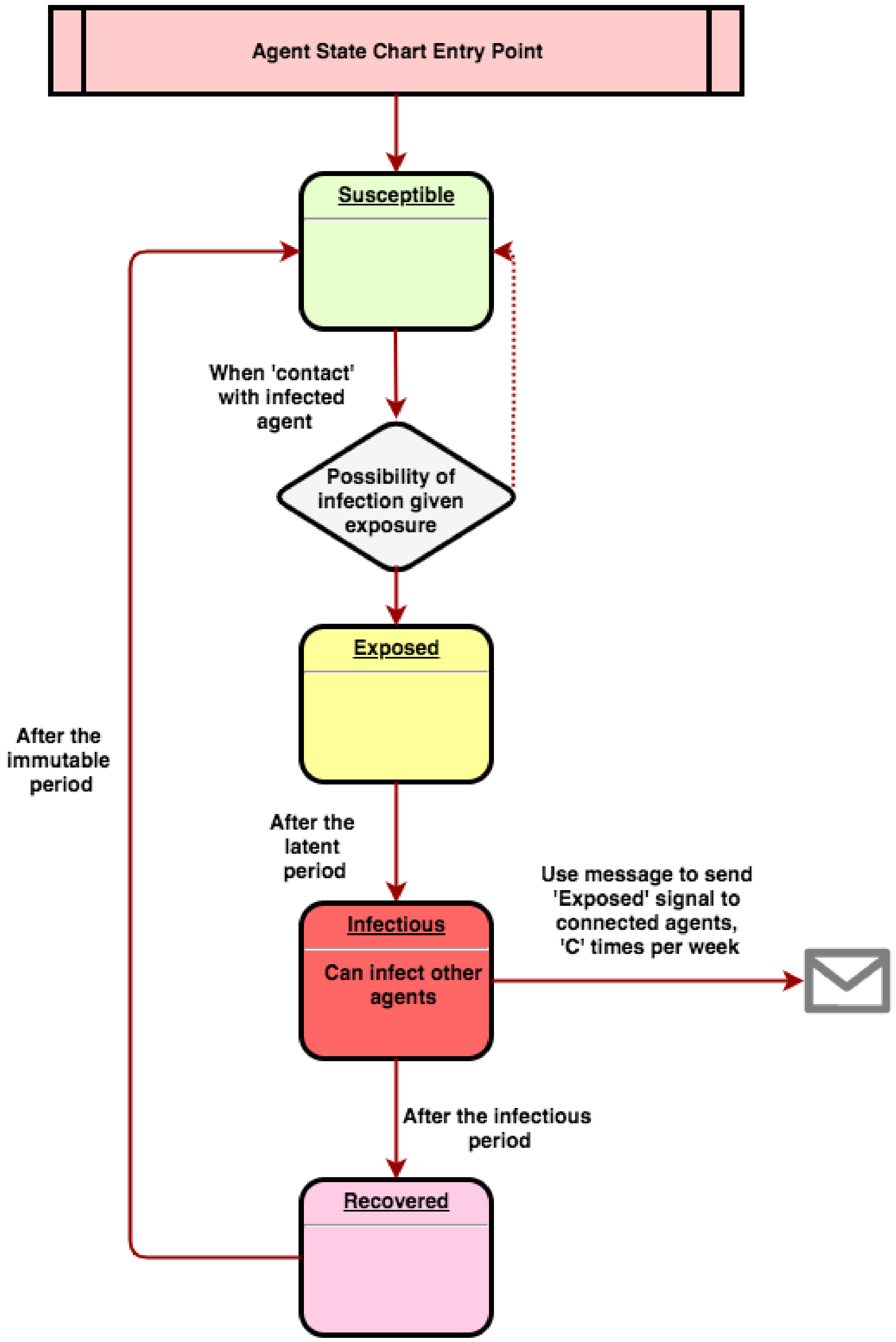

Agent State Chart Entry Point
Susceptible
When 'contact' with infected agent
Possibility of infection given exposure
Exposed
After the latent period
Infectious
Can infect other agents
Use message to send 'Exposed' signal to connected agents, 'C' times per week
After the infectious period
Recovered
After the immutable period

Figure 4.3: Agent State Chart in an Agent-based Simulation

Table 4.5: Parameters and Values of the SEIR model

Parameter	Value	Unit
Probability of Infections given exposure	0.005	
Latent Period	0.42857	Weeks
Infectious Period	1	Weeks
Have immunity	10	Weeks
Contact Per Unit Time	Changing Over Experiments	Times/Weeks
Total Population	300	Count
Likelihood per week of non-infective cases reporting	0.001	
Likelihood per week of incident infective cases reporting	0.85	
Network Type	Random	
Connections per agent	300	

Figure 4.4 plots three lines to display the difference between the count of the newly infected individual from the System Dynamics model and a single realization of the Agent-based model. The red line shows the actual counts. The green line adds reporting errors as previously mentioned and thus represents one possible realization of synthetic data that can be used for the PMCMC algorithm. The integration of the System Dynamics model depicted here is simply the output of the deterministic ordinary differential equations, and does not reflect the incorporation of the stochastics infection flow. As a result, the result from the System Dynamics model remains invariant across multiple realizations. By contrast, the agent-based ground truth is generated within the ABM, and the randomness from both infections and that reflected in measurement error causes the result to be different each time. While the contact rate c can be inferred easily using the System Dynamics result, the synthetic *empirical* data contains both measurement noise and system noise, reflecting the randomness in the probabilistic behaviour of each agent. Such factors make it much harder to infer the correct rate. Different contact rates can change the dynamics of the system and produce different results. In 4.5 we can see that when changing contact rate c, the synthetic data shows similar shapes, which makes inferring the contact rate a challenging task in this situation.

4.4.2 Using PMCMC To Estimate Parameters and States

A key difference between the goals of the Particle Filter algorithm and those of Particle Markov Chain Monte Carlo methods is that, while the former samples just from the latent state of the associated state equation model, the latter can jointly sample from both the latent state and parameter space. We can thus use PMCMC to estimate via sampling one or more parameters of interest about the system we are modeling. In the most simple aforementioned SEIR model, the model can be applied to different diseases by use of the corresponding latent period, infectious period, transmission probability and population characteristics and

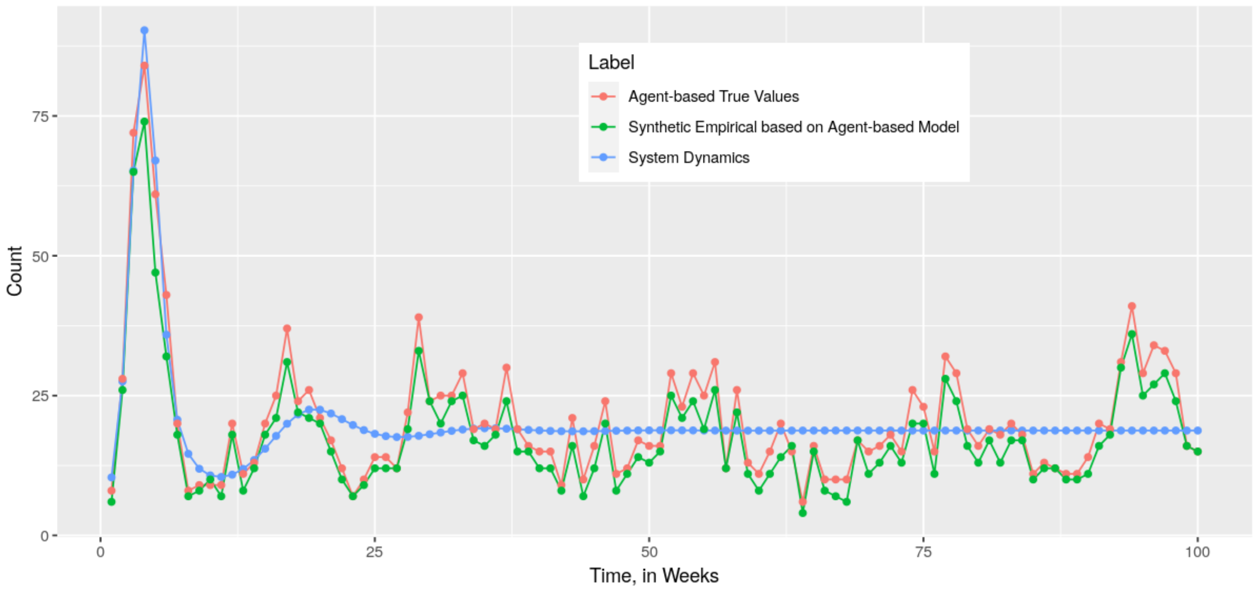


Figure 4.4: Synthetic Data from the System Dynamics and Agent-based Models

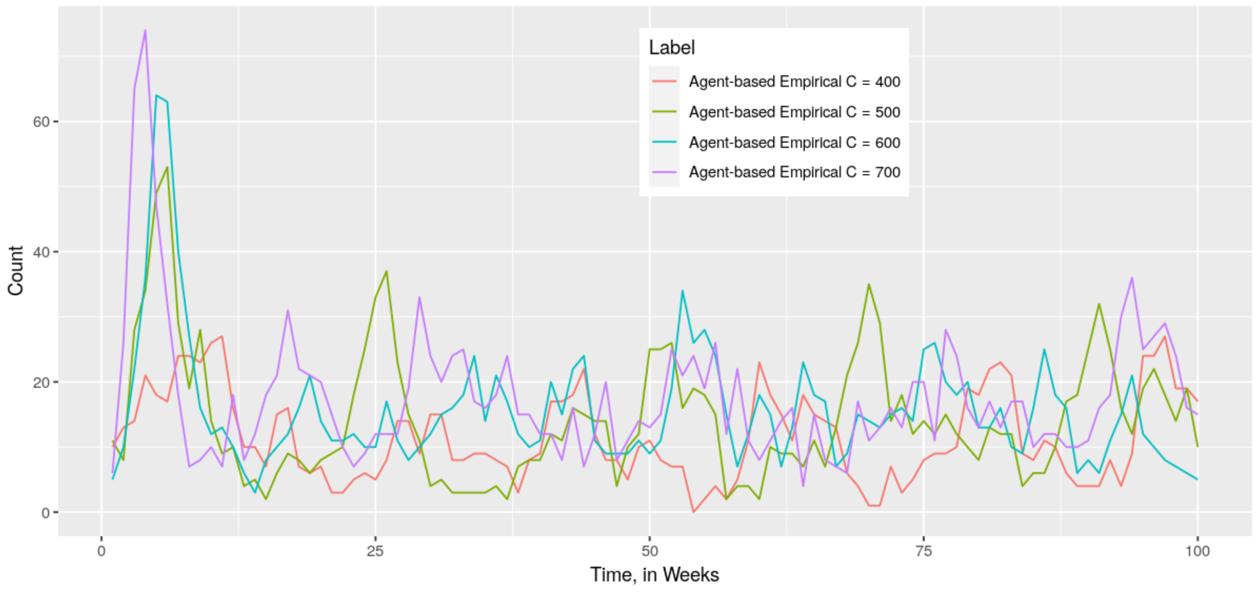


Figure 4.5: ABM Synthetic Data With Different Contact Rates

contact rate. Specifically, for a disease spreading among a particular population or community, the contact rate is one key parameter that can reveal the pattern and offer the ability to predict into the future. Thus, when applying PMCMC for the SEIR model, the contact rate is chosen to be the only PMCMC parameter considered. The random walk involved in selecting candidate samples within the PMCMC algorithm will lead it to explore a wide range of possible values for the parameter to be estimated. It is most natural to implement such random walks in an unbounded space. For the parameter, this implies a probability that the random walk pursued with respect to that parameter can become negative. For parameters such as contact rate, it is usually the case that negative value needs to be avoided since this will cause a negative flow value from susceptible to exposed stock. As is common within statistics in general and other PF models [22, 115] in particular, there is a transformation of the original parameter value performed to render it into an unbounded domain. In this case, the nature logarithm of contact rate is applied; this leads to a domain range of $[-\infty, +\infty]$ being mapped to $[0, +\infty]$.

Regarding migrating the model from the CPU implementation to the GPU implementation, the main changes lie in the functions' interfaces. Those functions are GPU kernels; they need to adapt the SIMD paradigm so that the accessing and assigning of arrays' values need to be changed accordingly.

Another detail extends from the fact that these are now GPU kernel functions: As a result, the function calls within those functions can only be device functions. There are several categories of functions. The first category includes those related to logging and debugging. For example, a kernel version of $assert()$ function will be needed. The next category includes random number generators for different distributions. An example is $knorm$ from $R.h$. The final category consists of functions associated with other mathematical and statistical methods, such as $logit()$ and negative binomial density functions, which are used to calculate likelihoods. Other kinds of functions will be similar to those. The next major step is to verify the correctness with unit tests and integration tests by controlling the input, focusing on one step of the algorithm at a time, saving the intermediate data using serialization, and performing comparisons and validation that the statistical behaviour of the system has been preserved in the conversion from CPU to GPU.

4.4.3 Performance of the GPU Accelerated SEIR Model

The final version of the SEIRS model used for performance measurement has four stocks, 32 synthetic data points, a single parameter sampled within MCMC to estimate the contact rate, using the generated synthetic data with a known contact rate. Unless mentioned otherwise, each experiment's combination is repeated 20 times in each testing environment to mitigate the influence of confounding factors, such as competition for resources of other applications on the same host or the speed reduction due to the hardware's temperature. For all experiments, the integration time step for the ODE is set to 0.01, such that 100 integration steps are needed to approximate the change in the System Dynamics model between observation points.

Hardware Variations

The experiment in this section will focus on the PMCMC code's performance with the SEIR model, as judged by running time on different kinds of machines and platforms. To start, two different sets of experiments are shown here in Table 4.6. The first set runs with 4096 particles and 800 iterations. This combination is representative for several reasons. First, choosing several thousand particles is usually considered a good starting point and acceptable for smaller models. It is the common choice before the GPU acceleration; such a particle count is also of the sample magnitude of the number of cores on the tested GPU devices. Specifically, the testing machines used in this work have 700 to 5000 cores.

The second set of experiments runs with 32768 particles and 100 iterations. The particle count will put a considerable amount of pressure on available cores on the device and memory capacity and memory bandwidth. With 8 times as many of particles and $\frac{1}{8}$ iterations, the total computational workload to be performed is kept invariant across those two settings.

Table 4.6 shows that with 4096 particles and 800 iterations, the CUDA accelerated code is faster than the original CPU version of the code in most cases. It is fastest on HPC cluster *Beluga*, which is equipped with the high-end NVIDIA Tesla V100 that has abundant cores and faster, larger device memory. It is slowest on *iMac* across all the GPUs. However, all GPU runs are faster than the non-parallel CPU version, especially considering that our testing machines span from dated personal workstations to powerful up-to-date HPC.

As shown in Table 4.6, with eight times the particle count, it is evident that the fastest GPU runs are still on *Beluga*, and slowest on *iMac*. What changed is that the slowest GPU run is much slower than all the CPU-based runs. This is expected, as even though this 8-year-old entry-level mobile GPU platform can support CUDA acceleration and execute the GPU PMCMC implementation, the 384 CUDA codes and 512MB on device memory far from satisfy the resource needed for the tasks. This shortcoming mostly extends from the GPU on-device memory constraint; the on-device memory becomes the bottleneck for large particle counts. The on-device memory is full and needs to be cleaned and loaded again every time the computation for the current batch is finished. Given this, atop the overhead of moving data to and from the device, running on GPUs with such specifications will not benefit the model execution. By contrast, will take longer than simply finishing the CPU tasks. All other modern GPUs are faster than CPUs in our tests.

Patterns are evident when looking at execution times reflecting how CPU and GPU architectures handle many particles, as seen in Figure 4.6. On CPUs, the running time across the two architectures is almost the same, which makes sense for sequential execution of two jobs of the same size when memory is sufficient. A minor difference can come from the random number generation calculation associated with acceptance criteria, which the first set needs to execute 800 times while the second one only needs to perform it 100 times. When the GPU and host CPU are powerful, and memory is faster and sufficient, the difference is slight on the GPU. On personal workstations, such as on *D2*, *D3*, *iMac*, the differences are much larger. The GPUs on such hosts are consumer-grade, and the host CPU and bus lack the requisite capacity to transfer 32768 particles efficiently. Sometimes the on-device memory will not be able to hold or cache efficiently to

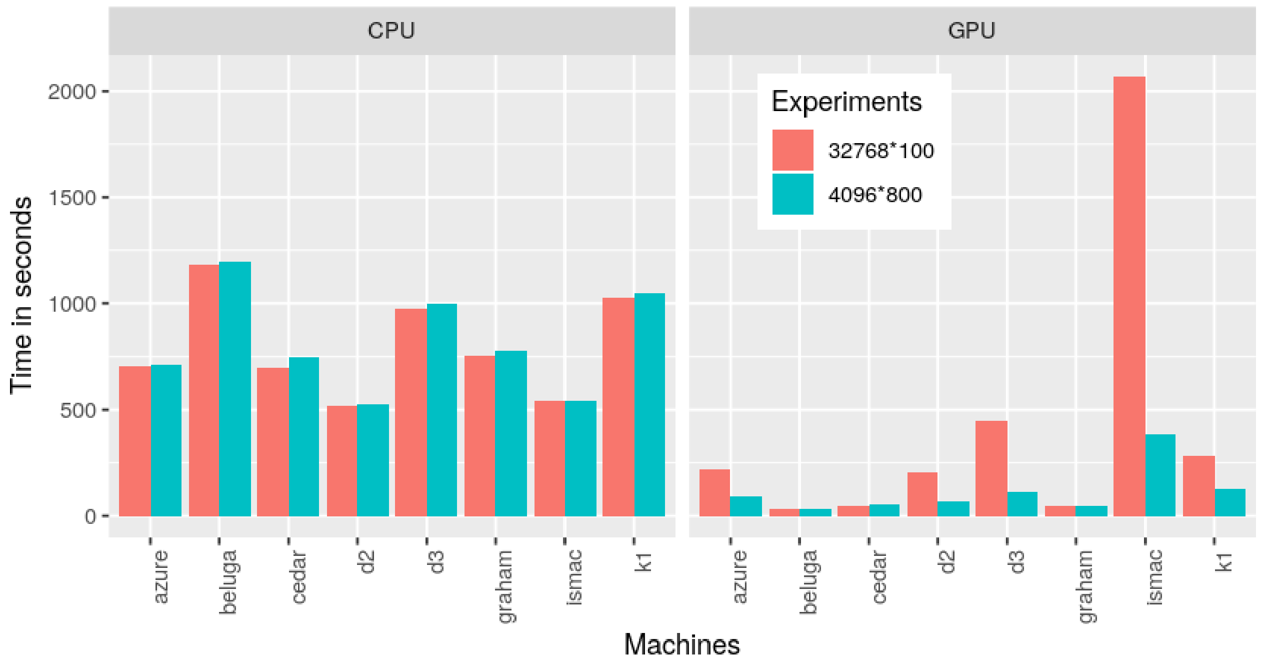


Figure 4.6: Running Time Differences, between experiments of 4096 $\times$ 800 and 32768 $\times$ 100, on different CPUs and GPUs

maximize core utilization.

In Figure 4.7, if we plot the distribution of CPU over GPU running time ratios across all different machines, we can see that, if a sufficiently powerful GPU is used – such as the those on HPCs – we would expect up to 40X performance despite the experiment setup for the two sets of experiments chosen. By contrast, the left distribution is more concentrated in the lower ratio, which means, in general, the improvement of GPU over CPU tends to be smaller with larger particle counts; that means, for current hardware, it is easier to scale with 4096 particles than for 32768 particles; with 4096 particles, we should expect constant improvement across different machines currently, but with 32768, the relative performance of GPU vs. CPU depends heavily on the machine.

Scalability

We have seen how the performance scales across different host configurations. We can focus on a few environments and do in-depth experiments there, and the results just shown will help us how such results will generalized across those machines. The next group of experiments will look at the running time with different experiment settings. Specifically, here we will list how particle and iteration counts will affect the running time and performance gains of GPU-based computation relative to CPU. The next chapter will continue on to discuss how different models induce different running times. The next set of experiments focuses on the GPU-parallelized PMCMC algorithm's scalability using the SEIR model. The basic experimental idea being pursued here consists of focusing on one testing environment at a time and running the model while varying

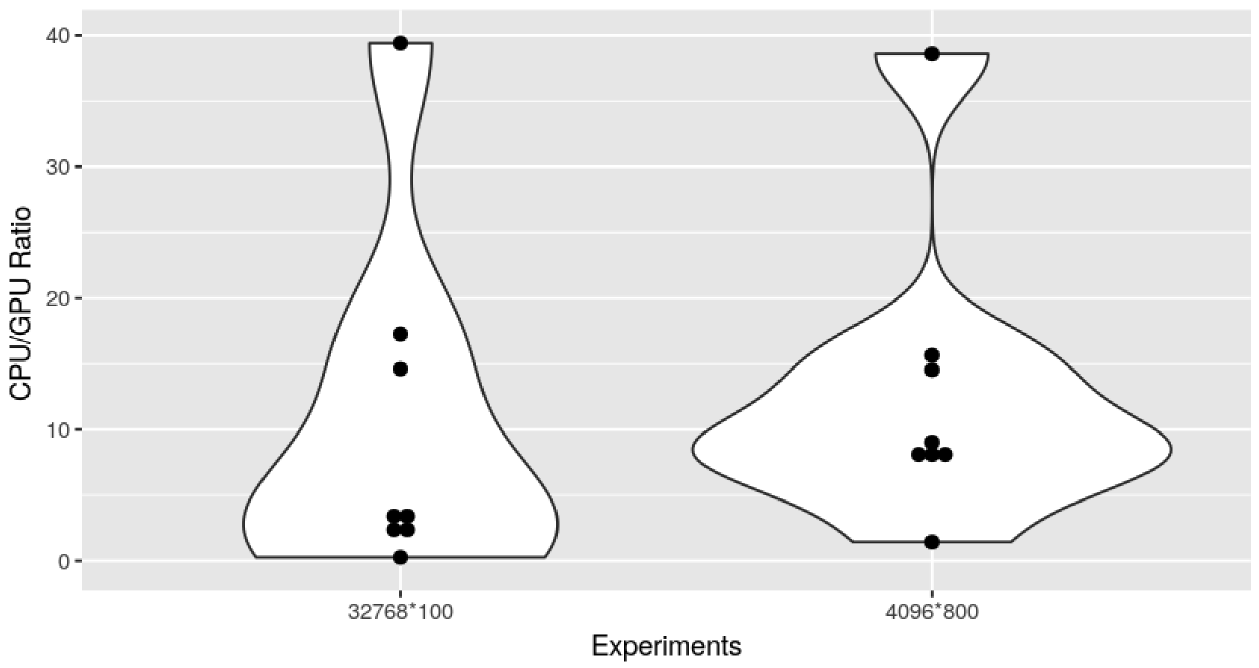


Figure 4.7: Running Time Distribution: Violin Plot for Speed Up Ratio Across All Machines

critical algorithm parameters related to running time, such as retaining invariant the count of particles and varying the iteration count. The scaling of the parameters selected always takes place in the context of successive doubling. When displaying the results on a $log2$ scale, we can easily observe how the running time scales over different combinations on particles and iterations. The testing environment for this set of results was the *Kepler* cluster at the Department of Computer Science, University of Saskatchewan. The machine has the following specifications: Each machine is equipped with two NVIDIA Tesla K20c, each holding 2496 CUDA cores and 5GB GDDR5, but the experiments are using only one such GPU. Each machine has two sockets for two server-oriented CPUs, each of which is a Intel Xeon CPU E5-2620 v2 at 2.10GHz base speed with six cores and 12 threads each, and the experiments for the CPU version of PMCMC is set to run on a single CPU core. An automated script runs the PMCMC model with $particles \in [256, 512, 1024, 2048, 4096, 8192, 16384, 32768]$ and $iterations \in [100, 200, 400, 800, 1600, 3200, 6400, 12800]$; we omit the runs with $[16384] \times [12800]$ and $[32768] \times [6400, 12800]$ for the sake of plotting. The running time experiments are performed at Git commit **bbc413b**[6].

In Figures 4.8 and 4.9, the running time was plotted on a $log2$ scale and group the results based on iteration and particle count. These figures are organized in converse ways. Figure 4.8 organizes the information more coarsely with respect to iteration count, and (for a given iteration count), more finely by particle count. Note that the results for the configurations of 16384 particles with 12800 iterations and 32768 particles with 6400 and 12800 iterations are omitted because of the failure of launching such configuration on GPU due to memory constraints of this test machine. For the GPU accelerated code, when the iteration count is fixed

[6]GitLab Repository: `https://git.cs.usask.ca/cephil/pmcmc_dynamic_modeling`

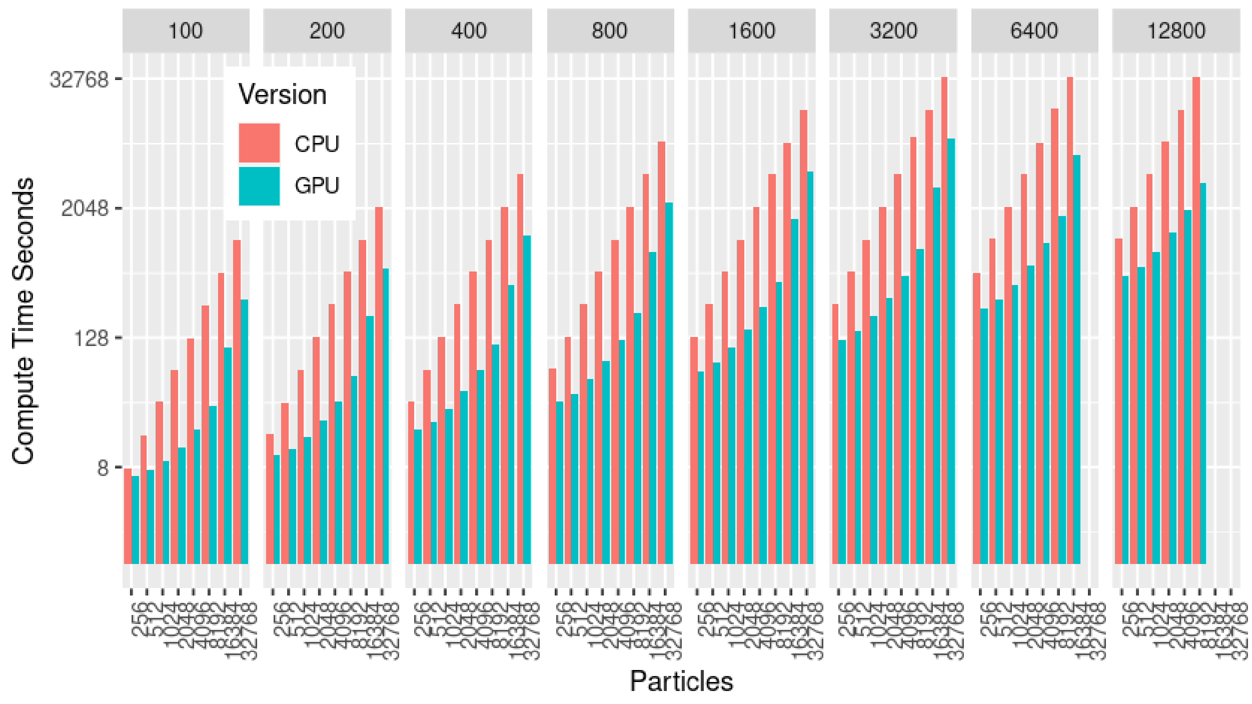


Figure 4.8: Running Time in Seconds, with Different Iterations (grouped in the categories at the top), on *log*2 Scale on Machine Kepler

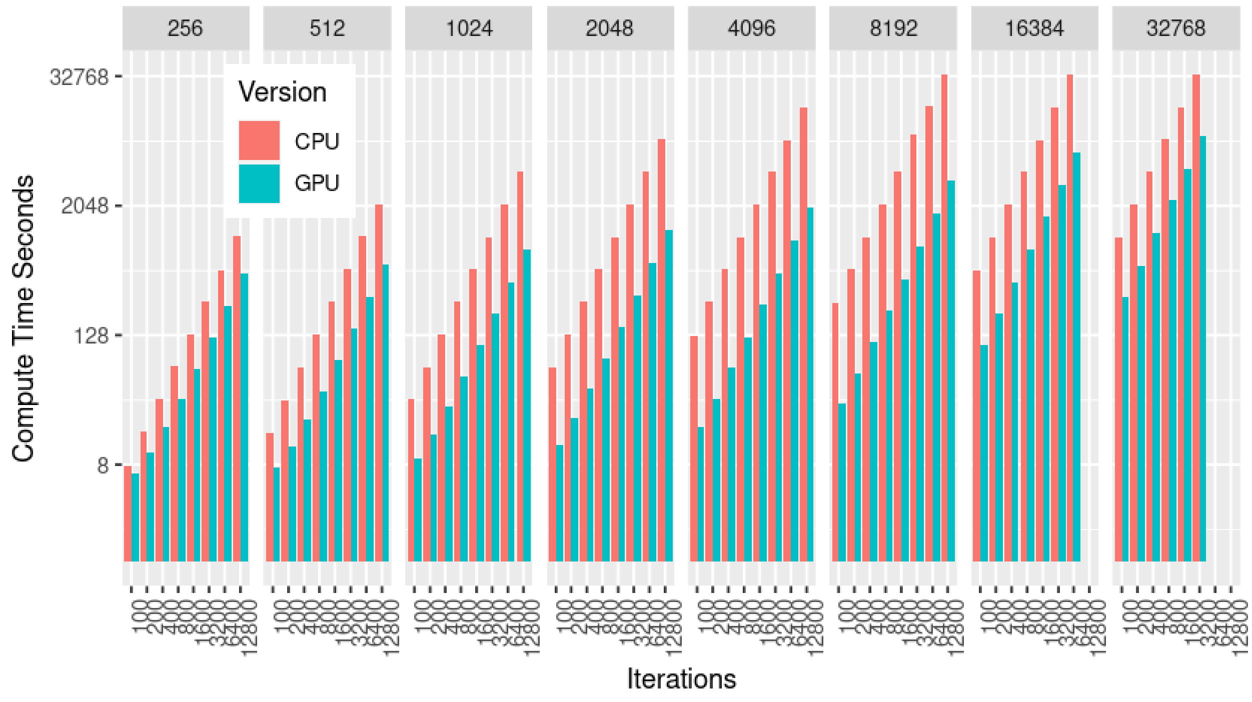


Figure 4.9: Running Time in Seconds, with Different Particles (grouped in the categories at the top), on *log*2 Scale on Machine Kepler

within one column, a rising slope is going up means the running time is super-linearly increasing with the particle count; that is, doubling the particle count more than doubles the running time. By contrast, for the CPU version, when doubling particle count, the running time increases proportionally; this reflects the linear trends on the $log - log$ plot. This tells us that the GPU version gradually loses its advantage when particles are far bigger than the count of CUDA cores available on this testing machine. The non-parallel CPU code is not affected by this resource bottleneck. 4.9 organizes the information more coarsely with respect to particle count, and (for a given particle count), and more finely by iteration count, where that iteration is set to double every time. In this figure, we can see that when the number of particles fixed within one column, the slope is fixed. When the particle count is fixed, the running time is proportional to iterations for both GPU and CPU versions.

It is also interesting to plot the ratio of improvement with different experiments. As shown in Figure 4.10, the CPU over GPU running time ratios are plotted over different particles. No matter how many iterations there are, the ratios of CPU running time over GPU running time are close when the particle count is fixed; for example, with 4096 particles, the ratios range from 7 to 9. With lower particle counts this range is substantial; for 256 particles results the improvement ratios fall between 1.18 to 2.22. This can be explained using Amdahl's law. There are computational tasks within the simulation runs that is invariant of number of iterations and particles, such as the initial setup of observation array and other values. Those fall into the part of the computations that can't take advantage of the parallelization, namely, $1 - P$. When the count of particles is small, the part of the computations that can take advantage of the parallelization P is relatively small; increasing iterations will increase P since it will divide the initial setup among all the iterations. This results in a smaller computing time per particles on GPU as iteration increases; and since the CPU computing time per particle is not varying with number of iterations, we can then observe the changing of CPU over GPU running time with lower particle counts such as 256. This effect is also elevated by the fact that this testing machine has slower CPU and disk I/O with HDD instead of SSD, causing P to be further smaller. As the count of particles increases, initially the ratio of CPU to GPU runtime increases, reflecting a successive greater speed advantage of GPU computation; at 2048 particles, GPU is about **10** fold faster than CPU runs on this testing machine. However, as particle count increases further, the ratio ceases to increase, and reaches its peak when the particle count is far greater than the count of CUDA cores (here, 2496 cores). Above that particle count, the ratio starts to decrease, with the advantages of GPUs of CPUs decreasing markedly with increasing particle count. The abnormal high value in this plot for 8192 particles with 3200 comes from anomalous CPU computing time; the per particle computing time are 14% higher than the mean values; additional repeating of experiments on this machine shows that this is an anomaly potentially caused by competing system resources on the same host at the time of experimenting.

We repeat the same experiments on HPC *Cedar* clusters, which have GPUs with more CUDA cores, and as seen in 4.11, as the particle count increases, the improvement ratio increases. Again note that the device memory of this testing environment is not sufficient for launching the configuration of 32768 particles with

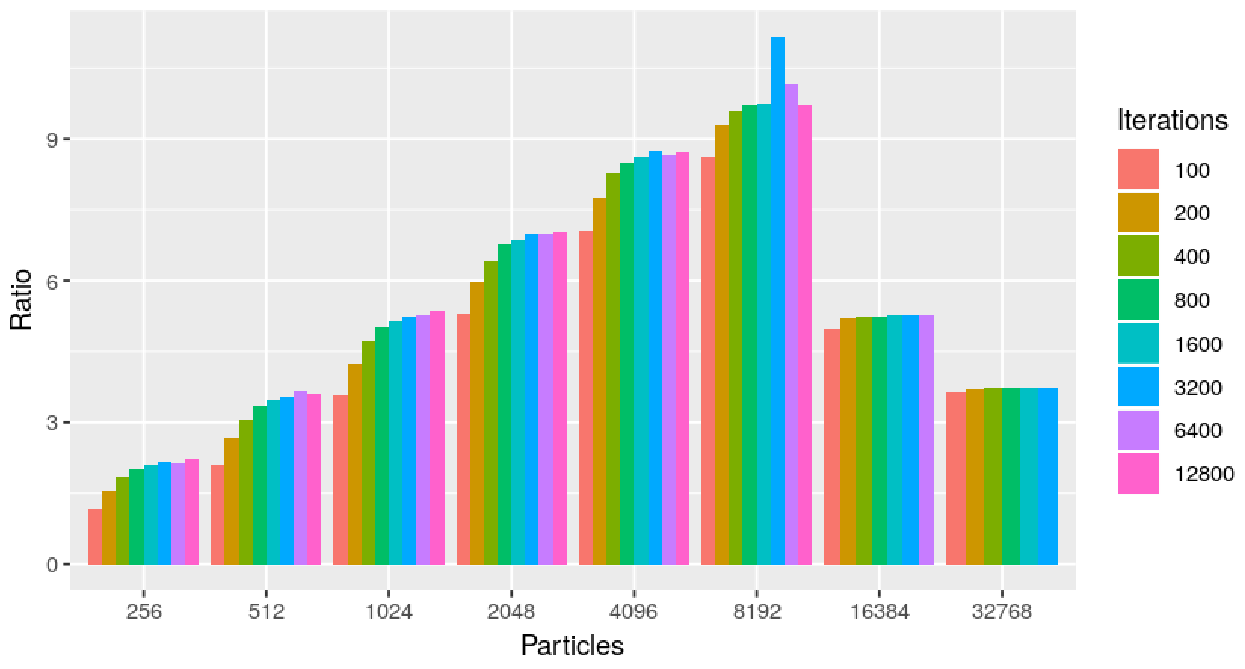


Figure 4.10: Running Time CPU over GPU Ratio, with Different Iterations on Testing Machine Kepler

12800 iterations and thus the ratio for this configuration is not presented in the plot. At 16384 particles, GPU is **18** fold faster than the original version of the CPU code. From the trending, we can see that the ratio will again eventually stop its increase with particle count and reach its peak when the particle count is far greater than the count of CUDA cores (here, 3584 cores). We believe the improvement ratio will decrease, based on the observations from other machines; comparing to the smaller scale server *Kepler*, it takes more particles to reach this performance turning-point on this HPC machine because of the hardware difference between them. This is again similar to the results observed in the previous section. An interesting trend is that even though the GPU version is always faster than the CPU versions, the improvement will increase initially as the particle count increases. This reflects a growing capacity to parallelize computation across such particles. But after a certain point, the increasing stops. This reflects the point when all the cores and GPU memory are fully utilized.

4.4.4 Streaming Particle Markov Chain Monte Carlo with SEIR Model

With the PMCMC enabled SEIR model substantially accelerated for broad sets of configurations, the opportunity is now available to provide valuable results much faster. This acceleration supports our goal of designing a streaming system for PMCMC leveraged simulation models. On the basis of the acceleration noted here, for many models and configurations, we would anticipate a sufficiently brief execution time to allow a PMCMC enabled model — like its Particle Filtered counterpart — to operate in an online fashion, consuming the data points live and provide refreshed results incrementally. Unfortunately, as we mentioned

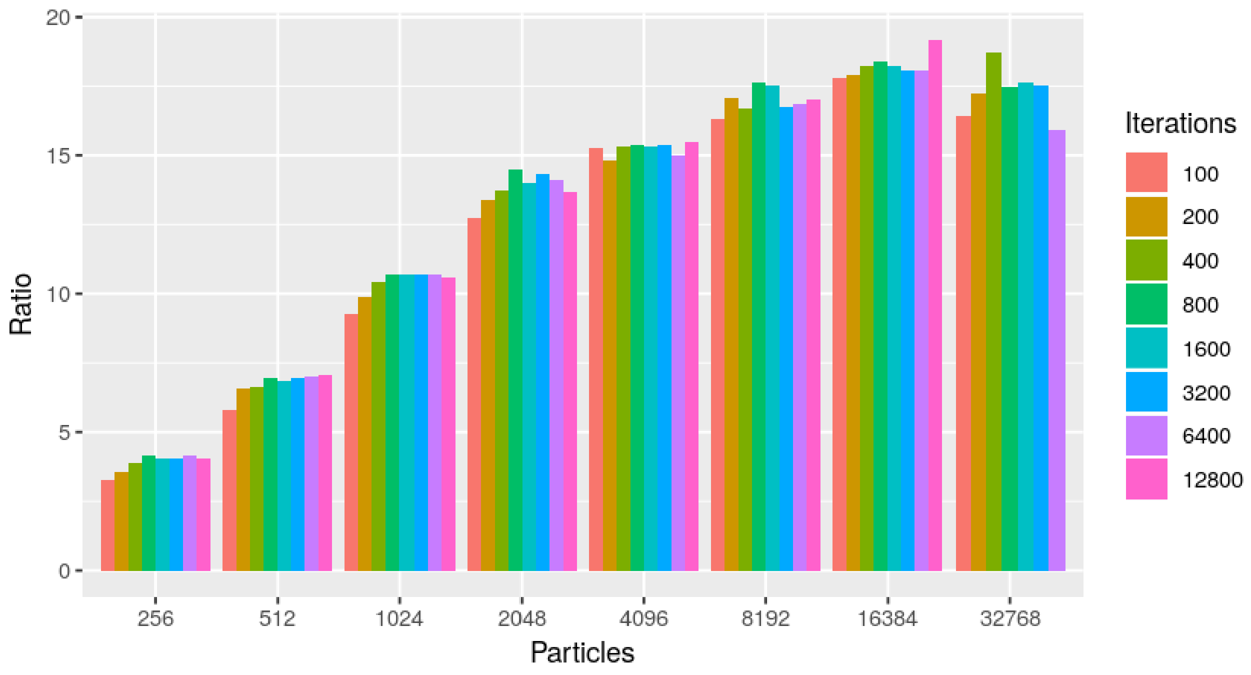


Figure 4.11: Running Time CPU over GPU Ratio, with Different Iterations on Testing Machine Cedar

before, because of the iterative nature of the PMCMC method, the algorithm requires the entire length of the data needs to be available at each step of the MCMC process. Typically, that means whenever a new point is available, the sampled distributions of the model states and parameters become outdated. By default, we would have to restart the PMCMC process from the beginning and explore the time series with a new data point.

In this work, without introducing alternative computational statistics algorithms, we propose a simple method for the new runs to take advantage of posterior estimates from the previous PMCMC runs with shorter time series. This section further examines the effectiveness of that approach, seeking to investigate if such variation in restarting the PMCMC process can save execution time, help us secure valuable results earlier, while avoid entirely discarding data from the previous runs. The method we are proposing here has a simple conceptual basis. We know that at the starting of a brand new Markov Chain chain, the initial value of parameters needs to be selected; conceptually, this is drawn from a prior distribution. While this value is drawn from a prior distribution, it is critical that the value of the posterior distribution for that value be greater than 0, as that values appears in the denominator of a ratio of probabilities for when assessing whether to accept the next candidate sample. Given that the posterior distribution cannot be expressed in closed form, finding a parameter vector with non-zero posterior value is not always easy. Drawing the proposed value from a naive prior distribution will often lead to a drawn value with a posterior probability density of 0. With a naive distribution, often this requires iteration until a value with non-zero probability is found. Empirically, for PMCMC enabled public health System Dynamics models, it could take several rounds of searching before a non-zero likelihood value is found. After the value is available, many iterations are then needed before the chain becomes stable and starts to explore the target's higher density area. Achieving convergence to sampling from the distribution is of key importance for the meaningfulness of the sampling, and an important enabler is achieving an acceptance rate well above 0 – with authors having argued for seeking acceptance rates in the area of 23% [116] through as high as 80%. Because the initial samples are so heavily biased by the vagaries of the initial draw, it typically requires many iterations to reach higher posterior density regions of the distribution. As a result, the initial period of the chain is not typically considered to contribute samples representative of the posterior distribution. Thus, nominal samples gathered during from this period are typically thrown away when calculating the result, This period before the chain is judged to be well-mixed is therefore commonly referred to as the chain's burn-in period. The longer the burn-in period, the more iterations in total will be needed for each run, which directly affects the running time and time needed to draw valuable conclusions from the samples.

While methods exist to shorten the period, simply increasing the 'step size' delta will not always work. Again, from our experience, this burn-in period can be long for the models built for public health research. The prior distribution from which parameter values are originally drawn can have a large impact on the number of iterations required to achieve convergence of MCMC (and, by extension, PMCMC) algorithms. Given the arrival of a new observation, we sought to use the final parameter vector sampled from the posterior distribution from the previous PMCMC run (considering strictly previous observations) as the starting parameter vector for the new run considering this additional observation. We called this process "carrying samples", and sought to use it to enhance the quality and speed of convergence in our streaming adaptation of PMCMC. It bears emphasis that the sample from the posterior value carried over to serve as the posterior value in the new PMCMC will in general be a vector of parameter values and not simply a single scalar value. This approach scale readily to handle the case where several new time points become available. Thus, for example, 10 more new data points could become available and yet still use the previously sampled value as the starting point as the draw from the prior distribution to start the new PMCMC chain.

As the time series length increases, finding a good proposal becomes difficult. Now, plot all the densities for all chains, and we can see that the posterior gets closer to the true value used for synthetic data. When taking advantage of the previous sample, the acceptance decreases less. To get the same 50% acceptance rate, fewer and fewer burn-ins need to be thrown away, making the inference much efficient. In this section, we investigate the benefits of using this approach to combine PMCMC enabled compartmental models with streaming time series. The experiments use acceptance rate as the deciding metric, and are set up as follow. We initialize the experiment by tuning the parameters to get a close to 65% acceptance with 50 initial data points. Then the acceptance rates are collected when running with successively more extended time series to simulate the data points become available incrementally in the real world. Specifically, we start a new PMCMC run with two additional data points each time. Each experiment is repeated 20 times, and statistics are collected. A total of three experimental cases will be compared side by side: The first experimental case starts from a random *log_C* value that is far from the true value used in synthetic data generator. Such a discrepancy between the initial and true value commonly obtains in real-world applications, where only a rough estimation of the true value is known, and frequently the initially sampled value will be far away from the true underlying value. The second experimental case instead uses the true value used in the generator as the starting point for the PMCMC algorithm. This is considering helping the algorithm and the best possible case for parameter inference: The true value can be considered a sample from the high-density area of the posterior to help the algorithm start the search from a high-density area, which reduces the burn-in time required. For both of the above scenarios, as new data points become available, the algorithm starts from that selected starting point. The final experimental case is a method that we propose in order to take advantage of the streaming configuration. With the same starting point as the first case for the initial run of the PMCMC model with the small count (50) of observations, now we let the individual run within one deployment to take advantage of the previous run in the series by using the previous sample as the starting point. The proposal distribution used for Random Walk MMH MCMC algorithm used with the simulation model with PMCMC framework based on a candidate drawn from a normal (Gaussian) distributed perturbation from the most recently sampled parameter vector. Taken successively, such perturbations give rise to a (posterior-influenced) random walk amongst parameter vectors. We can use the true value and full length to test the convergence speed. Having done so, we then set up different streaming chains and test the convergence speed of the final run with the observation vector of full length. With different iteration settings, we can see that for lower counts of iterations, basing the runs the prior distribution on the sample from the previous run's posterior distribution can be beneficial, regardless of the random walk standard derivation. Use of this sample allows the parameter vectors sampled in the PMCMC algorithm to move slowly towards the posterior, even when given recourse to few observations. This can help real-world applications, where now we do not need to wait for many data points to be available to run the algorithm, and can also lead to the need to discard fewer burn in samples – even for the final run – collectively saving a significant amount of time.

Note that implementations of algorithms for MCMC evaluation and diagnosis are available in different CRAN[7] R packages; we used CODA[8] and FitR[9] for effective sample size computation and visualization. In Figure 4.12, the acceptance rates for the three experimental cases along the successive streaming runs are collected and plotted. Compared to the ordinary case where every longer run starts from a randomly guessed location $v = 7.2$ (depicted in the top right subplot), our proposed method, shown in the top left subplot, starting from the same locations for the very first PMCMC run with 50 data points, has a higher and stabler acceptance rate, especially in the vicinity of the end of the series. As can be seen from the bottom right plot, our proposed method is very close to the case where each single PMCMC run is started from the true location with $v = 5.85$.

The Effective Sample Size for the three cases in comparison is also presented in Figure 4.13; the 25 lines

[7]The Comprehensive R Archive Network, https://cran.r-project.org/
[8]CODA: Output Analysis and Diagnostics for MCMC: `https://cran.r-project.org/web/packages/coda/index.html`
[9]FitR: Toolbox for fitting dynamic infectious disease models to time series, `https://rdrr.io/github/sbfnk/fitR/`

represent the change of ESS with respect to burn-in for the 25 individual PMCMC runs with varying time series lengths. This kind of plot aid decisions regarding the optimal cut-off time for burn-in. The ESS is a measurement of independent samples following burn-in. When the burn-in period is short, the samples will tend to exhibit stronger autocorrelation. Such samples can provide useful information about the mixing performance; when the burn-in period increases, the ESS will eventually start to drop as useful samples are thrown away. A good value for cut-off could be where the ESS reaches it maximum across parameters. Now looking at the three cases' experiment results, it is clear for the middle subplot – where true value is used as the initial value for PMCMC runs – that the peak of ESS occurs at the start . When we start from the true value, all samples are valuable, and no burn-in period is needed; this is the optimal case, as we discussed before. The right subplot represents the ordinary case where random location is used, and no sample is carried. ESS indicates that a significantly long burn-in period is needed for every PMCMC run in this series before ESS reaches its peak. By contrast, for our proposed method, just the initial PMCMC run needs to suffer this prolonged burn-in; successive runs can be the optimal case as using the true value. Note that in this set of experiments, this result is possible because the very first PMCMC run can efficiently explore the target distribution, and thus the final sample of the first run can offer much valuable information for the second PMCMC run and all later ones as well. If convergence to the target distributions frequently requires an iteration count beyond what is explored in the PMCMC runs — for example, given an abbreviated running time between each data point or marked covariation between parameter values — the first few runs may not reach a location that could provide the optimal ESS for the successive run at 0 burn-in. However, even this situation, use of this streaming approach should still reduce the burn-in period needed.

4.5 Summary

CUDA C/C++ is an extension of standard C/C++ language that offers heterogeneous computing ability. The host code that runs on the CPU and system memory, and the device code, which is designed to be executed on GPU devices, can be mixed following the C++ language semantics and style. This strongly increases the accessibility of transforming code from the C/C++ codebase to CUDA compatible code. Ideally, when undertaking similar tasks, I would recommend that the first step is to transfer the code to C/C++ first, as this increases the confidence of the migration to GPU and makes measuring the performance gain possible in small steps.

In this chapter, we have investigated an implementation of the GPU accelerated PMCMC algorithm with System Dynamics models, demonstrated one application of the implementation using an SEIR-structured such model. We discussed the experimental settings, the establishment of the testing environment, and presented the results of testing the SEIR application in different experiments. The implementation process described primary considerations when moving to GPUs and shared lessons learned in the process. The experiment results demonstrate significant improvement in terms of speed and scalability for the GPU accelerated version

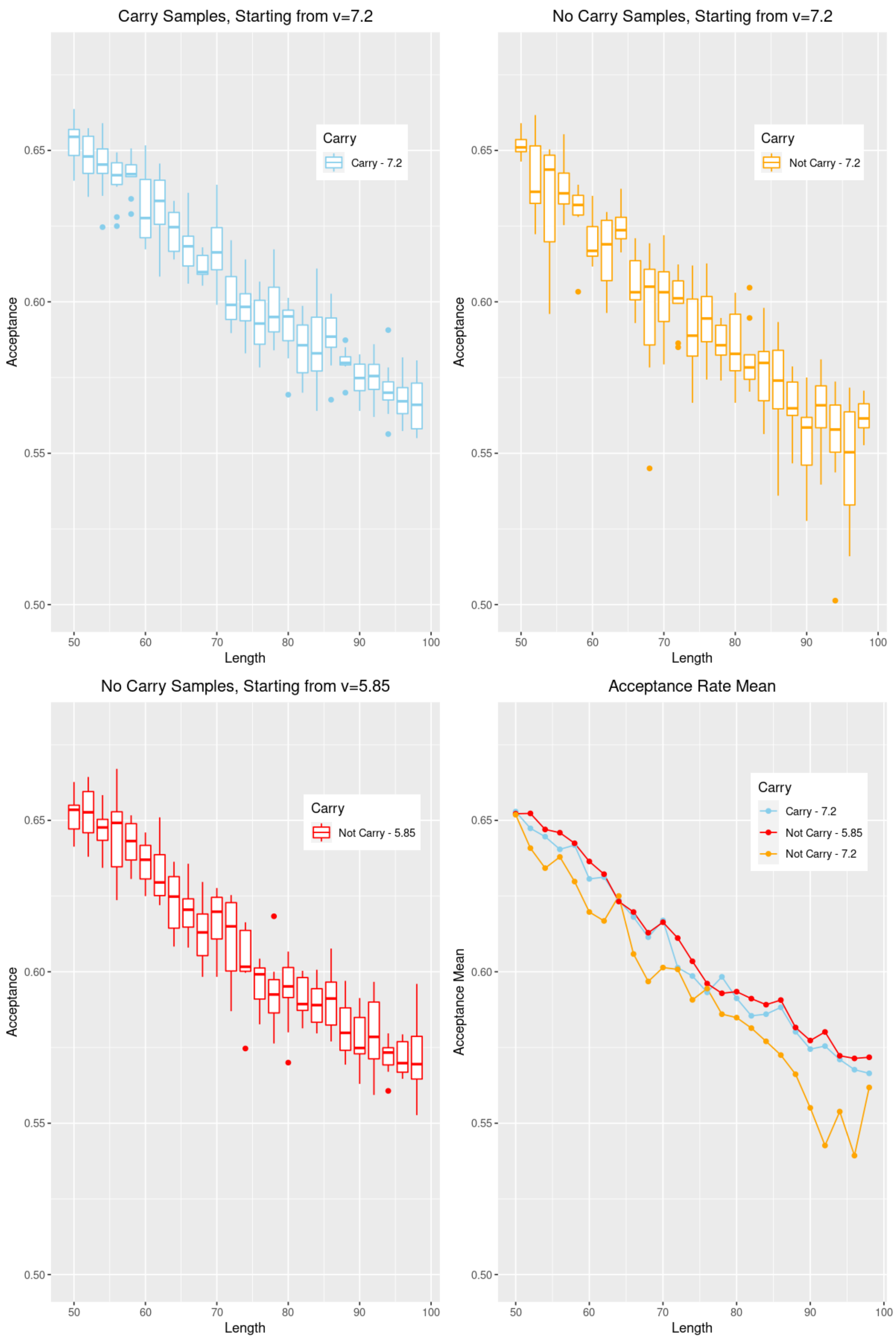


Figure 4.12: Decreasing acceptance rate

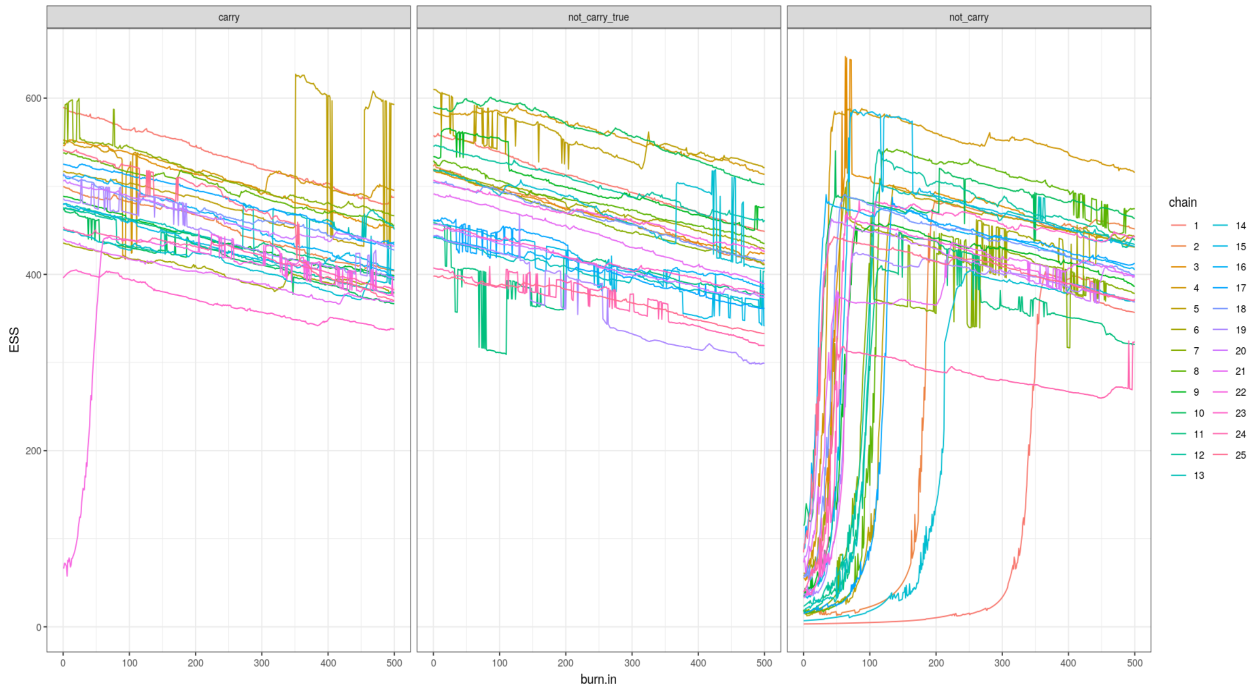


Figure 4.13: ESS for different situations

compared to the CPU implementation. Furthermore, with the accelerated framework greatly enhanced the accessibility of testing multiple chains with streaming ability on PMCMC models. Taking advantage of this accessibility, we successfully carried out some preliminary experiments to test the benefit of streaming PMCMC with System Dynamics models. Although the results are encouraging, additional experiments are needed. To complete the task, we will introduce another GPU implementation application, but with a substantially more complicated model using real-world streaming data as input.

Table 4.6: Running Time Across Different Machines for the SEIR Model

4096 Particles, 800 Iterations			
Testing Machines	Mean CPU Time	Mean GPU Time	CPU/GPU Ratio
Azure	710s	92s	7.7
Beluga	1198s	31s	38.6
Cedar	751s	52s	14.4
D2	530s	69s	7.7
D3	999s	111s	9.0
Graham	773s	49s	15.8
Ismac	544s	381s	1.4
K1	1050s	124s	8.5
32768 Particles, 100 Iterations			
Testing Machines	Mean CPU Time	Mean GPU Time	CPU/GPU Ratio
Azure	706s	220s	3.2
Beluga	1180s	30s	39.3
Cedar	696s	48s	14.5
D2	521s	202s	2.6
D3	976s	450s	2.2
Graham	758s	44s	17.2
Ismac	589s	2070s	0.3
K1	1030s	286s	3.6

5 Application of CUDA PMCMC On an Influenza Model

5.1 Introduction

Bayesian computation methods enable the ready combination of empirical data with models to simulate communicable disease outbreaks; the availability of information-rich and easy-to-access social media and other online data further enhance such models' predictive accuracy. [1] found that adding Google Search volume data on related topics could substantially elevate predictive accuracy of an influenza model used with Particle Filtering method beyond what is possible using clinical data alone. It also demonstrated how multiple data sources could be used together in an System Dynamics model to characterize influenza spread and support the hypothesis that the fear of getting infected is also contagious. Work described in this chapter extend this model with the powerful PMCMC method to build a complete case study for the mechanisms introduced in this thesis with this Particle Filter-grounded, fear-included, multiple-data sources influenza model. Those will include:

- Adapt the same model to Particle MCMC, with one more social media data sources, to better estimate the parameters and understanding the underlying states;
- Experiment and compare how the CUDA accelerated code can speed up this model;
- Demonstrate how the model runs with online Particle MCMC, by using a varying length of time series;
- Describe the complete streaming solutions for this model, including all data sources collecting adapters.

This chapter is organized as follows: first we will describe the model and modifications, and then evaluate the model inference performance and perform the speed tests; the implementation of the streaming system for this model as well as the its effect on this model will be demonstrated.

5.2 Using Particle MCMC in Influenza Model Inference

5.2.1 Model Description

As shown in Figure 5.1, the influenza model with fear spreading schema adapted from [117] has eight stocks in total. Besides the commonly known susceptible, exposed, infectious and recovered stocks, four more stocks were introduced to simulate the proportion of the population who is anxious about the pandemic, or influenced by and eventually become self-isolated from the general population. Those four extra stocks are:

- I_F, or Scared population, is the stock for individuals that have developed anxiety of the spreading of the disease, which is influenza here;

- R_F, or Removed due to fear, is the stock for individuals that removed them from the circulation of the disease transmission due to the effect of fear towards influenza; as seen in the diagram, individuals will be in this state after they become scared first; once recovered from the removed state, they will again become susceptible;

- I_{FP}, or Scared & Infectious, describing those that are infected by both the fear and pathogen;

- R_{FP}, or Removed due to Fear & Infection; individuals in this state will not contribute to the spreading of the influenza virus further as they have removed themselves from the circulation.

The model described in [1] was built in AnyLogic with Particle Filtering and focused on the Canadian populations of Quebec and (separately) Manitoba, respectively. The model is formulated as follows:

$$\begin{aligned}
\frac{dS}{dt} =& -\beta(1-\alpha)\frac{\epsilon}{N}SI_P - (1-\beta)\alpha\frac{c}{N}SI_P - \beta\alpha\frac{c}{N}SI_F \\
& -\beta(1-\alpha)\frac{c}{N}SI_{FP} - (1-\beta)\alpha\frac{c}{N}SI_{FP} - \beta\alpha\frac{\epsilon}{N}SI_{FP} \\
\frac{dE}{dt} =& \beta(1-\alpha)\frac{c}{N}SI_P + \beta(1-\alpha)\frac{c}{N}SI_{FP} + \beta(1-\alpha)\frac{c}{N}SI_P + \beta(1-\alpha)\frac{c}{N}SI_{FP} - \frac{E}{\tau} \\
\frac{df_p}{dt} =& \frac{E}{\tau} - \alpha\frac{c}{N}I_PI_P - \alpha\frac{c}{N}I_PI_F - \alpha\frac{c}{N}I_PI_{FP} - \lambda_PI_P + HR_{FP} \\
\frac{dI_F}{dt} =& (1-\beta)\alpha\frac{\varepsilon}{N}SI_P + \alpha\frac{c}{N}SI_{FP} + (1-\beta)\alpha\frac{c}{N}SI_{FP} - \beta\frac{c}{N}I_{FP} \\
\frac{dI_{FF}}{dt} =& \beta\alpha\frac{\varepsilon}{N}SI_P + \beta\alpha\frac{\varepsilon}{N}SI_{FP} + \beta\frac{c}{N}I_FI_{FP} + \alpha\frac{c}{N}I_F \\
& I_F + \alpha\frac{c}{N}I_PI_F + \alpha\frac{c}{N}I_{FF} + \alpha_NI_{FP} - \lambda_{FP}I_{FP} \\
\frac{dR_F}{dt} =& \lambda_FI_F - HR_F \\
\frac{dR_{FP}}{dt} =& \lambda_{FP}I_{FP} - \lambda'_PR_{FP} - HR_{FP} \\
\frac{dR}{dt} =& \lambda_PI_P + \lambda_PI_{FP} + \lambda'_PR_{FP}
\end{aligned} \tag{5.1}$$

The parameters description and original values are listed in Table 5.1. This is a medium sized model in terms of observations count, observation length, stocks and state variables. Large to extra-large models can have up to 100 to 200 stocks and state variables, ten or more time series and thousands of available data points, especially when the time series is daily data[7].

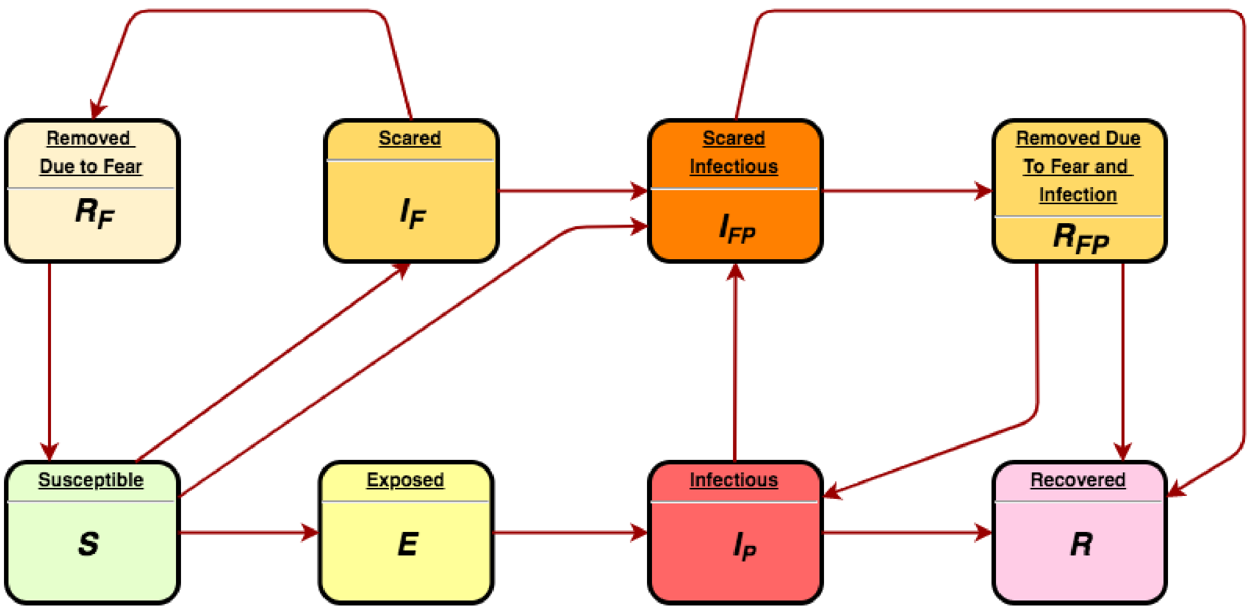


Figure 5.1: Stock-and-Flo Diagram for the Influenza Model (Adapted from [1])

5.2.2 Modification of the Model for PMCMC Inference

We modified the model to study the population of Saskatchewan and added one more data source: The count of Twitter tweets including words of interest. The weekly Google Search volume was collected for this province, and the clinical data was available weekly[1] for the flu season from October to March each year. The likelihood $L_Incidents$ is calculated using the negative binomial distribution comparing daily incident case counts from empirical data and the sum of all daily flows into Infected stocks. Similarly, $L_GoogleSearch$ is using negative binomial distribution comparing daily Google searching data and to a coefficient times the sum of all flows into Fearful stocks in the model. It bears noting that we consider searching as a behaviour similar to incident case counts in the second that both make use of information on the flows accumulated over a single day (one time unit). A new infected will count once toward this week's empirical data and a newly *fearful* individual will count, for example, 3 Google Search to this week's search volume.

In addition to these likelihood functions, the model included another likelihood - $L_twitter$ that reflects twitter mentions of influenza. By contrast to the above likelihoods that considered flows, the design of the model considered posting on social media such as tweeting as adhering to a different pattern; as long as an individual is in I_fp and I_f stock, we would consider that they will keep posting, for example, once a day. Based on this consideration, we consider the Twitter counts as proportional to accumulated stock values. Reflecting that reasoning, the model used a negative binomial density was used to compare Twitter empirical data to a model value given by a coefficient times the sum of the values of stock I_fp – representing individuals who are both fearful and infected – and I_f, namely those who are fearful but remain uninfected.

[1]Sask. Weekly Influenza Surveillance Reports: `https://tinyurl.com/SKFluReports`

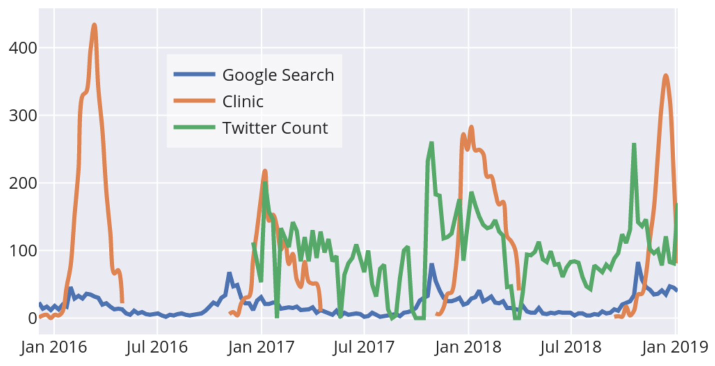


Figure 5.2: Empirical Data for Influenza in Saskatchewan

The product of the three likelihoods was used when calculating the total likelihood.

5.2.3 Data Description

The original model uses the empirical data collected from the provinces of Quebec and Manitoba's local health authorities, respectively and from online sources. In this work, the same data set is collected for the province of Saskatchewan; the collecting methods will be discussed later. The empirical data, shown in Figure 5.2, includes clinic reported data, Google search trends and Twitter tweets, including the list of keywords of interest for this geographical population. The counts were only reported for the clinic data during the flu season, which generally goes from November to March and has no data in between. While all historical data can be retrieved for Google search volume on Google Trends, the historical Twitter counts were only available after December 2017 from previously harvested data[2]. While the numbers represent the actual counts for the clinic data and Twitter data, the Google Trends data has been normalized to indicate the relative search volumes. The correlations analysis in Figure 5.3 shows that those two online activities and social media, such as searching and tweeting, have strong correlations with the clinic data. They usually arrive at the peak value right before the flu season, and higher values in those indicators will lead to higher clinic reported numbers from the recent three years of the data. The correlation plot between the data also demonstrates a high correlation between the two online indicators, which leads us to have increased confidence about those two in uncovering the true and whole numbers about influenza epidemiology during the flu season in Saskatchewan. A subset of continuous data points was used in the experiments (20 weekly data points of 2017-2018 flu season).

5.2.4 Model Inference Results

The use of PMCMC with the adapted model allows inference of parameters and states over the period of time based on the empirical data collected. There are also dynamic variables associated with the Particle

[2]Data collected from Twitter API in CEPHIL lab

Table 5.1: Parameters for the Influenza Model Adapted from [1] and Adjusted for Saskatchewan Population

Parameter	Notation	Original Value for Quebec	Value for Saskatchewan
Probability of infection transmission given exposure	β	0.04	0.04
Probability of fear transmission given exposure	α	0.02	0.02
Mean latent time	γ	Uniformly distributed (2,4), unit days	Uniformly distributed (2,4), unit days
Mean time to recovery	μ	7 days	7 days
Total population of province	N	7,843,475	1,098,352
Rate of recovery from fear	H	0.2, unit per day	0.2, unit per day
Rate of removal of self-isolation from fear	λ_F	Dynamic, unit per day	Dynamic, unit per day
Rate of mean to time to recovery of going from SScared Infected" to "Recovered" via "Removed due to Fear and Infection"	λ'_{FP}	Dynamic, unit per day	Dynamic, unit per day
Rate of removal to self-isolation from fear and pathogen	λ_{FP}	$\frac{1}{\mu\lambda'_{FP}}$	$\frac{1}{\mu\lambda'_{FP}}$
Rate of recovery from infection with pathogen	λ_P	$\frac{1}{\mu}$	$\frac{1}{\mu}$
Rate of recovery from removed due to fear and infection	λ'_P	$\frac{1}{\mu(1-\lambda'_{FP})}$	$\frac{1}{\mu(1-\lambda'_{FP})}$

Table 5.2: Initial Values for the Influenza Model Adjusted for Saskatchewan Population

Names	Initial Values
Susceptible Stock Mean	700,000
Susceptible Stock Standard Deviation	10,000
Exposed Stock	10
Infectious Stock	10
Scared Stock	10,000
Scared Infectious Stock	0
Removed Due To Fear Stock	0
Removed Due To Fear and Infected	0
Twitter Adjustment Parameter Disperson	2.0
Non Cases Reported Dispersion Parameter	50.0
Non-cases Google Dispersion Parameter	25.0

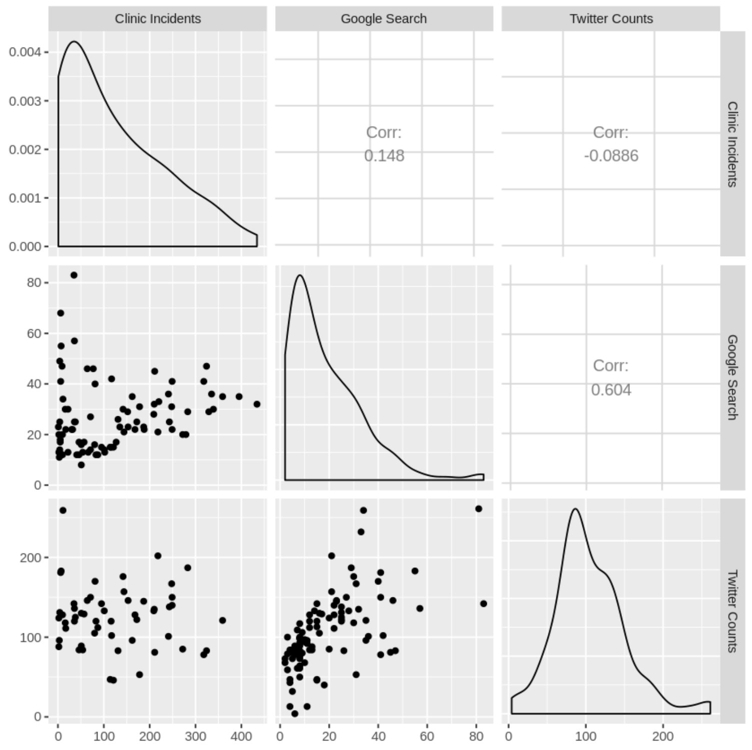


Figure 5.3: Correlation Plot: Empirical Data for Influenza in Saskatchewan

Filter process. The simulation experiments' outputs provide insights and predictions about the underlying condition. In Figure 5.5 and 5.4 the samples stock values are shown in 2D histogram as a distribution over MCMC iterations. The x-axis shows model times, as the model runs with a time unit of weeks and a total of 20 weeks of data. The y-axis is the values of the dynamic stocks; for stocks such as susceptible and exposed, the y-axis shows the number of individuals in those stocks. The stock's density represents the distributions of the particles, where particles are used to approximate the statistic possibilities. Because the stock values are not directly comparable to the empirical data, the empirical data are not shown in this plot set. In Figure 5.6 the dynamic variables are presented in the 2D histogram as well. For example, we can see that the removal rate from scared and infected, shown at the top right, was changing over time while the removal rate from scared only, shown at the top left, was kept at a concentrated value most of the time. Figure 5.7 depicts the distribution (across MCMC iterations) of sampled states over time as a 2D histogram. The x-axis shows model time (in weeks), as the data extends over 20 weeks. The y-axis is the values of the model state variable. The density represents the distributions of the particles, where particles are used to approximate the statistic possibilities. The state values are directly comparable to the empirical data, and the empirical data are shown plotted on top of the density. Essentially, the sampled values' distributions represent the density of the model's belief of the current state, while the empirical data at that step will update the belief using Bayes' theorem. We can see that the empirical data is mostly falling in the highest posterior density region, which suggests that the model performs well in inferring accuracy.

5.3 Performance of the GPU Accelerated Influenza Model

5.3.1 Hardware Variations and Scalability

This section will present and examine the performance patterns of the GPU accelerated influenza model with PMCMC in terms of running speed and resource usage. Those would demonstrate the achievement of the accelerated implementation and its potential in saving researchers time with all different kinds of hardware. Each set of experiments has been repeated 20 times to decrease the effects of resource competition of other applications. The ODE integration step is 0.01.

First, we would like to test and compare the overall running time between the original CPU code and the new GPU implementation in different test machines when running with this influenza model with a more extensive set of stocks and flows. With baseline settings for the performance experiment, with 4096 particles and 800 MCMC iterations, as shown in Table 5.3, the GPU-accelerated version is faster across testing machines. The fastest was when running on *Beluga*, benefiting from the largest CUDA core count, abundant memory and fastest memory bandwidth. By contrast, the higher CPU clock speed and larger system memory contributed to *D2* achieving the fastest CPU-version running speed. Comparing to the results shown in the previous chapter with the simpler SEIR model, the running times for the testing machine *iMac* is not included. The limited 512MB on device memory for *iMac* makes it unable to launch the GPU codebase

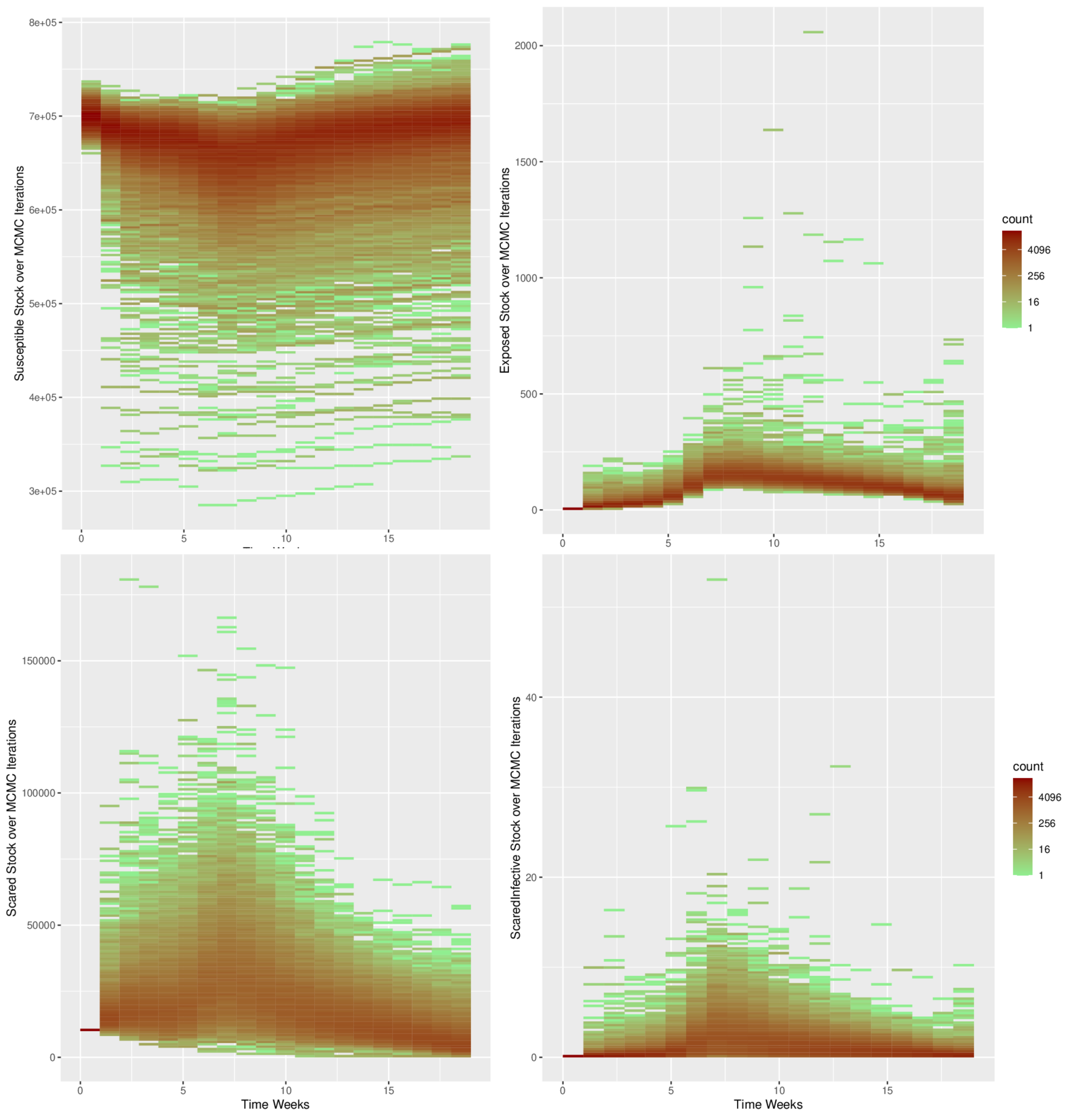


Figure 5.4: Sampled Stock Values (S, E, I_F, I_{FP}) of the Influenza Model.

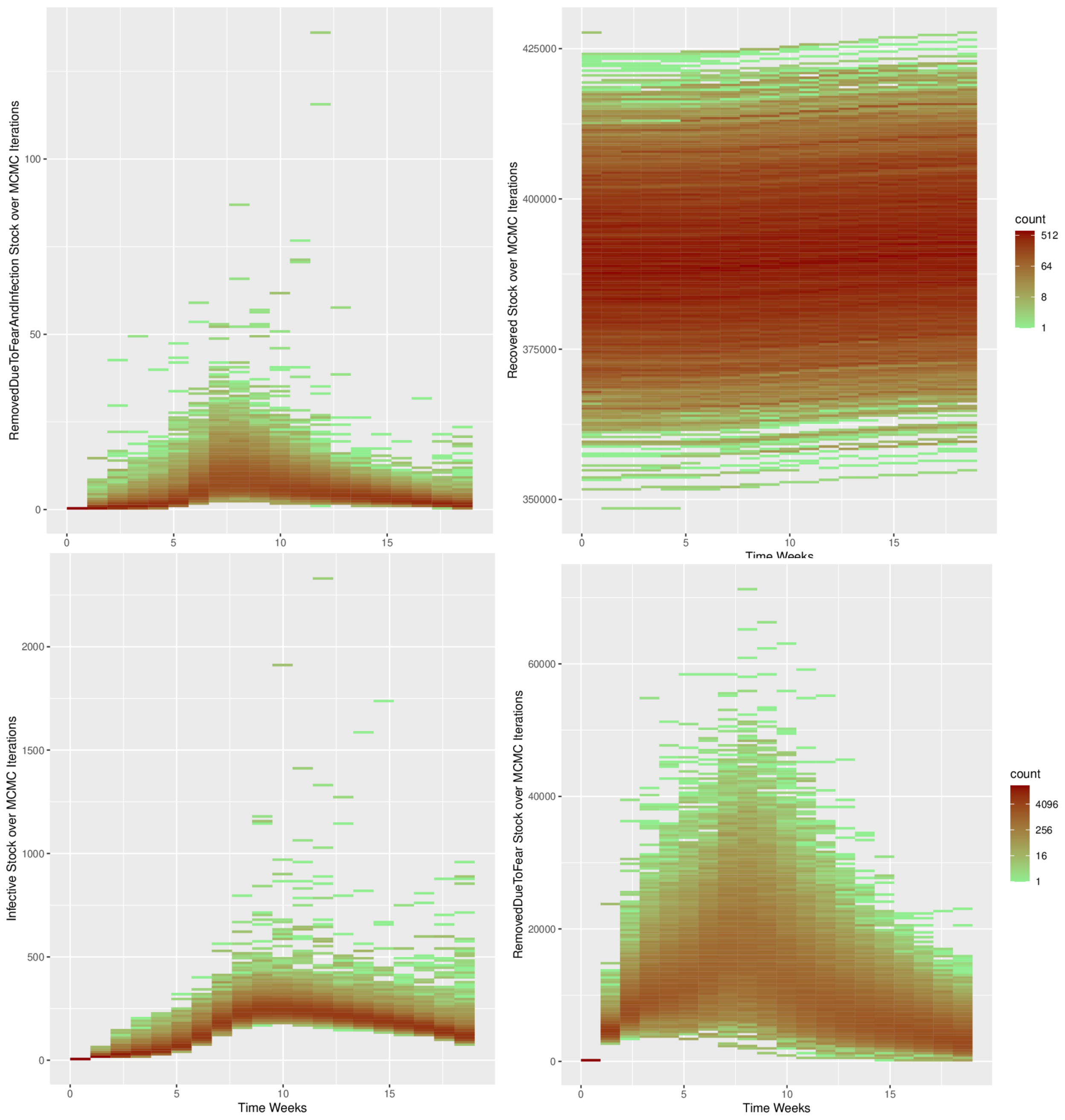


Figure 5.5: Sampled Stock Values (R_F, R, I_P, R_{FP}) of the Influenza Model.

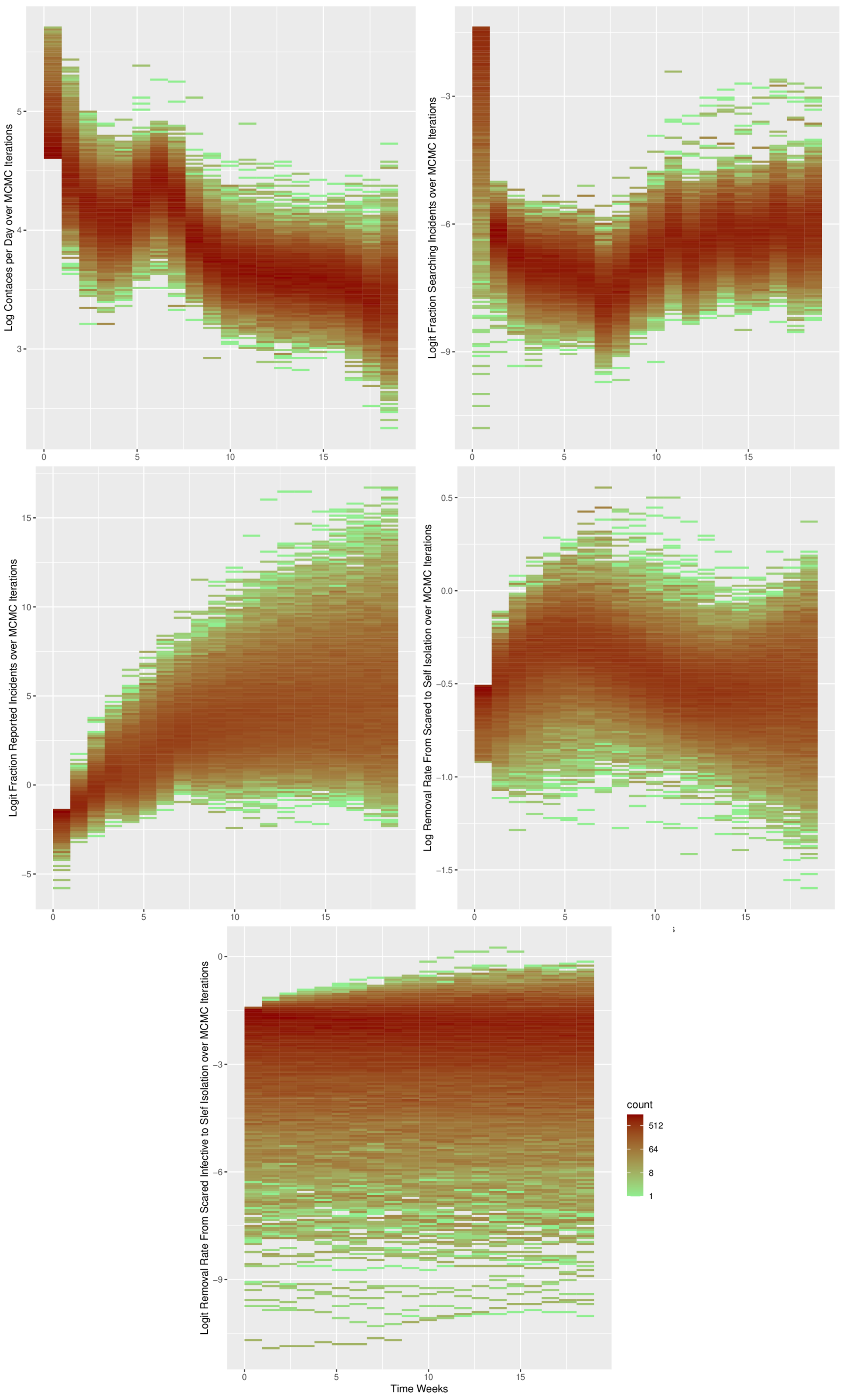


Figure 5.6: Sampled Dynamic Variables of the Influenza Model.

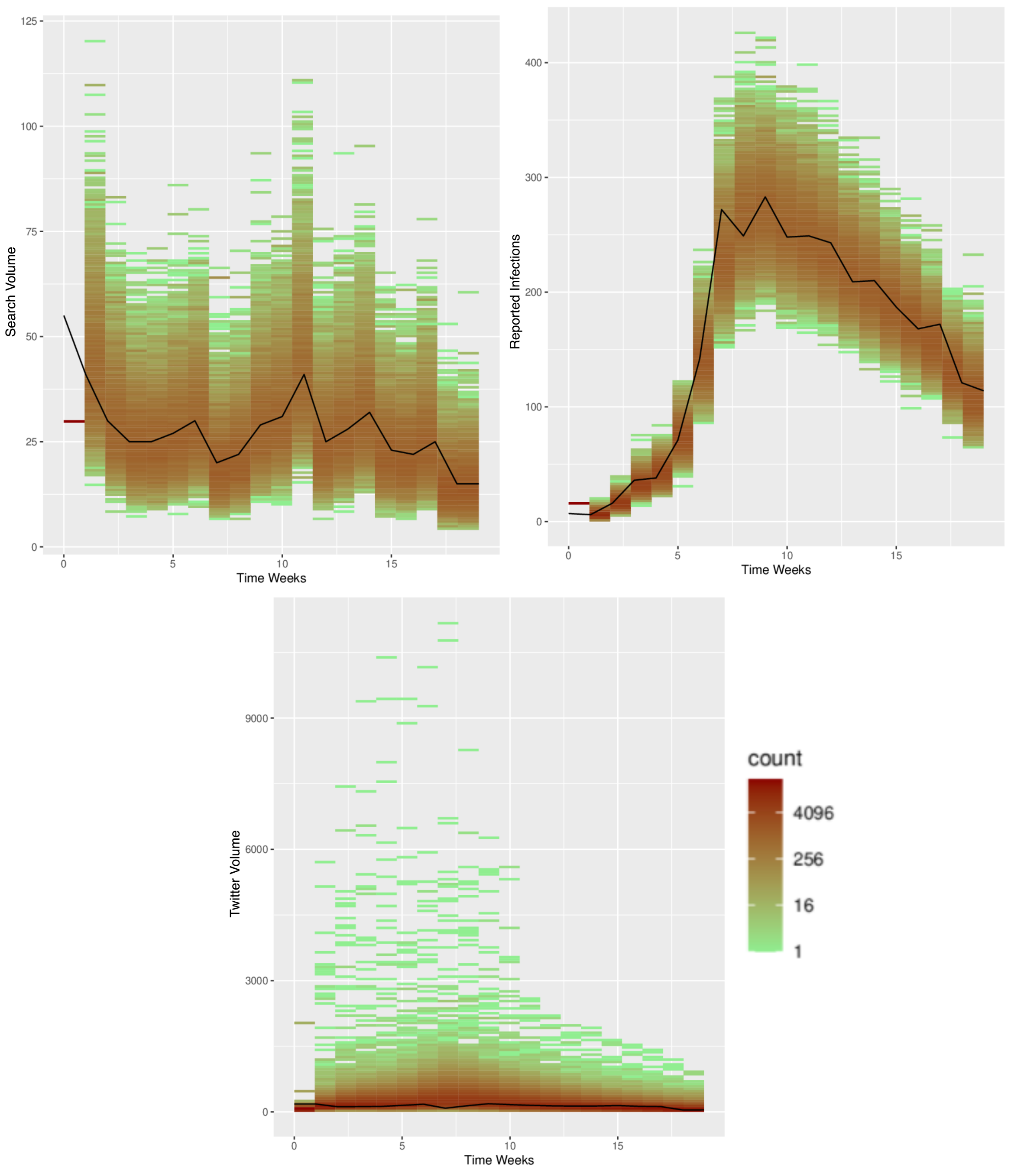


Figure 5.7: Empirical Data Compared to Corresponding Model Posterior Distribution.

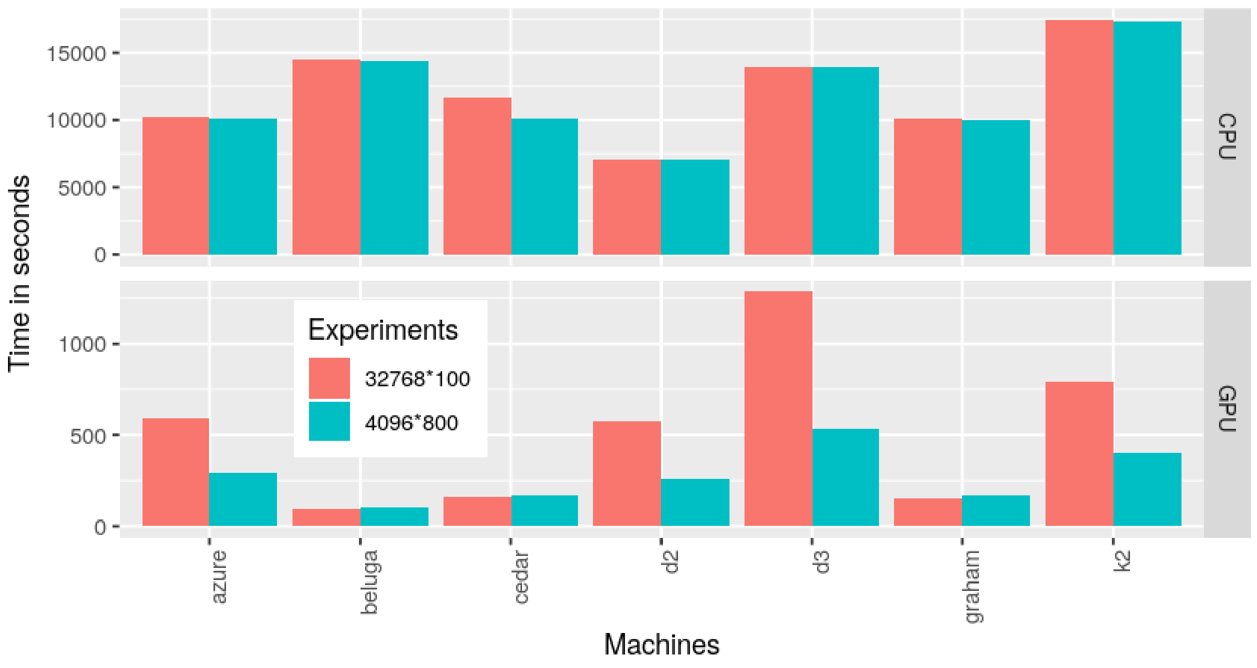


Figure 5.8: Running Time Differences Between 4096X800 and 32768X100 configurations on Different Machines' CPU and GPU.

for the influenza model, as the memory space is not enough to hold the entirety of the stocks and dynamic variables vector with a reasonable amount of particles.

Scaling up the count of particles by a factor of 8 to 32768 and reducing MCMC iterations down by the same factor to 100, we evaluated the relative impact of those configurations that matter the most to the algorithm inferring ability. The running times are compared for each machine between the two configurations and presented in Figure 5.8. They exhibited similar execution time running sequentially, reflecting the similar amount of tasks are involved. The running time is compared for each machine between the 4096×800 and 32768×100 and presented in 5.8 to see the difference between these two settings. With parallel code, the difference was unnoticeable for a relatively capable host and device. The count of particles would decide the task size at each MCMC iteration, and those tasks would be launched together and scheduled to run by the device. The not-so-different running time suggested that our implementation handled it well and did not require extra overhead to process a larger count of particles. On resource-restricted machines, such as *D2*, *D3*, and *K2*, the difference between the performance of these GPU configurations is far more significant. It is possible that the host and device bus speed prevented transferring a larger count of particles efficiently; the limited count of cores means it would take longer to finish threads synchronization when needed.

From a different angle, we are interested in seeing how much the GPU implementation can benefit the user on each type of testing machine; that is, we want to check the CPU version running time over each machine's GPUs. Figure 5.9 displays the CPU over GPU run time ratio distribution across different testing environments. For current hardware, it is easier to parallelize a low particle count, yielding a constant propor-

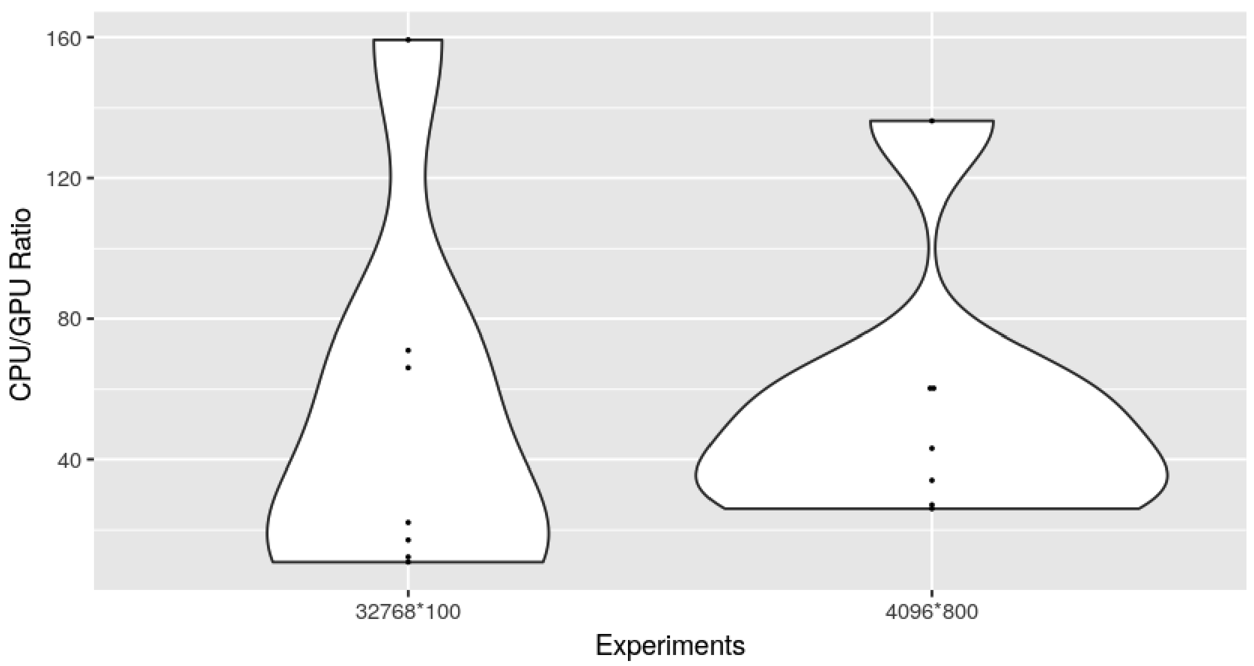


Figure 5.9: Running Time CPU Over GPU Ratios Distribution: Violin Plot for Speed Up Ratio Across All Machines

tional improvement for low particle counts. Even though a larger particle count yields smaller improvements on most lower-end GPU devices due to the overhead associated with mapping different groups of particles directly to the GPU cores, high-end GPUs can be fully utilized by all those particles, achieving up to 160× speedup.

Next, we investigate the scalability when keeping invariant the testing environment. Figure 5.10 shows the running time in seconds for both parallel and non-parallel versions when changing the count of iterations and (separately) particles. The values for the GPU running time (red) are small and can be hard to compare. Arranging the results by ascending particle count and grouping them by iteration count on a $log2$ scale as in Figure 5.11, it is clear that the CPU version of the code exhibits a log-linear escalation in running time with iteration counts. For the GPU implementation, the running time increases faster when the count of particles increases within each iteration group. This reflects the fact that as the count of particles increases, the task size at each MCMC iteration grows and extra time is required for host-device communication and thread synchronization, which scales non-linearly with the task size. For the GPU implementation, within each iteration group, the running time increases faster when the count of particles increases. This is expected, as with the number of particle getting larger and larger, the task size at each MCMC iteration getting bigger and extra time is needed for host-device communication and thread synchronization, which typically not linear to the size of the task. If we group the same results by particle counts and then arrange them in the increasing order of iterations, using the same $log2$ scale, as in Figure 5.12, we observe the perfect $log2$ increments within each particle group. Both the CPU and GPU code should scale linearly when we change

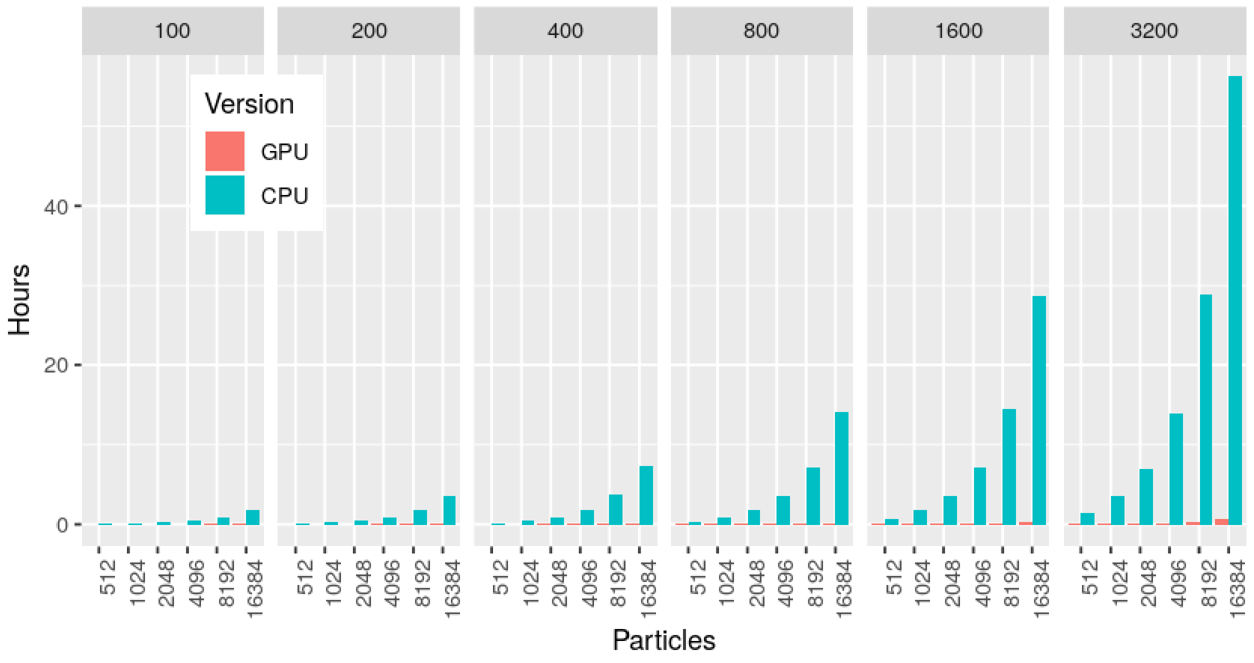


Figure 5.10: Running Time in Hours When Varying Different Counts of Iterations and Particles for Influenza Model on Testing Machine Cedar

the iterations, as the parallelization is happening within each iteration but not across iterations.

Across the particle and iteration counts parameter settings, we can observe substantial speedup; the actual improvements secured depend on the configuration. To further explore the performance pattern, we then carry out the same set of experiments on another machine, *D1*. Though we use the same set of particle counts and iterations, the hardware is limited this time and should present a different pattern, especially using the combinations at the end of the range. The running times are displayed in 5.14 first; note that the combination of 16384 × 6400 is omitted for the sake of presentation, but the result for this is easy to infer giving the context. Again, though we can observe the GPU implementation's significant improvements across all different settings, the actual ratios are different when the particle counts are different. Figure 5.13 and 5.15 shows the CPU over GPU running time ratio with different particles and iteration counts on individual testing machines, *Cedar* and *D1*, as samples of HPC and personal workstations are grouped by the count of particles. Note that the result for the configuration of 16384 particles and 6400 iterations is not in Figure 5.15 because of the failure of launching such configuration on GPU due to memory constraints of this test machine. The performance gain (improvement ratio) remains close to constant across different iteration settings within a given particle count. By contrast, with increasing particle count, the improvement ratio initially rapidly increases and then reaches a plateau at which hardware is fully utilized. For *D1* compared to the first testing machine, the peak improvement ratio was reached much earlier, with around 2048 to 4096 particles, due to hardware limitations, from memory size, cores and streaming processors, to the clock speed of the cores and the memory. On this more resource-constrained machine, the performance improvement secured by GPUs

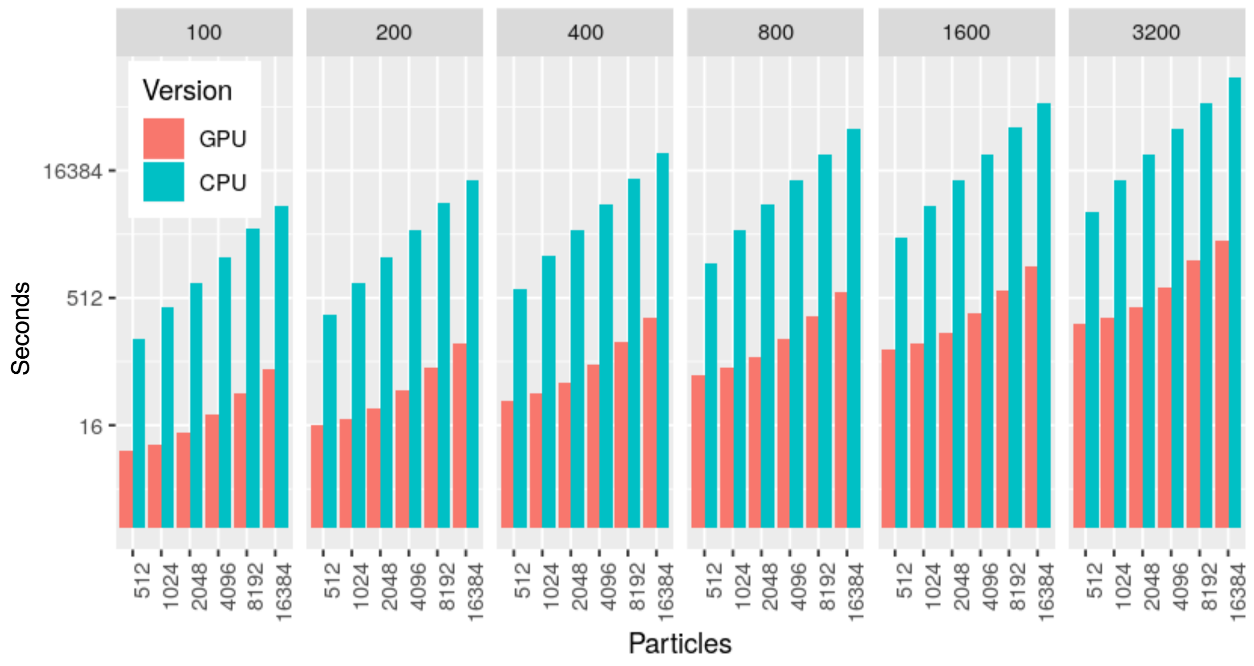


Figure 5.11: Running Time in Seconds when varying different counts of iterations and particles for Influenza Model, on *log*2 Scale on Testing Machine Cedar

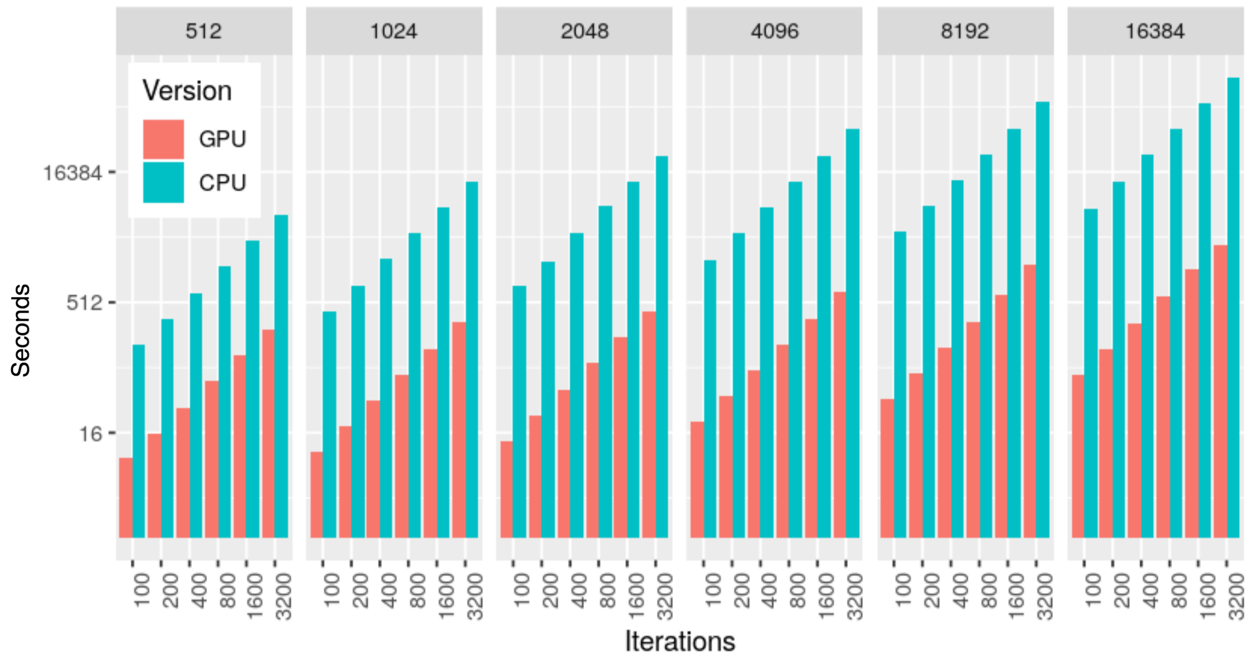


Figure 5.12: Running Time in Seconds, with Different Particles for Influenza Model, on *log*2 Scale on Testing Machine Cedar

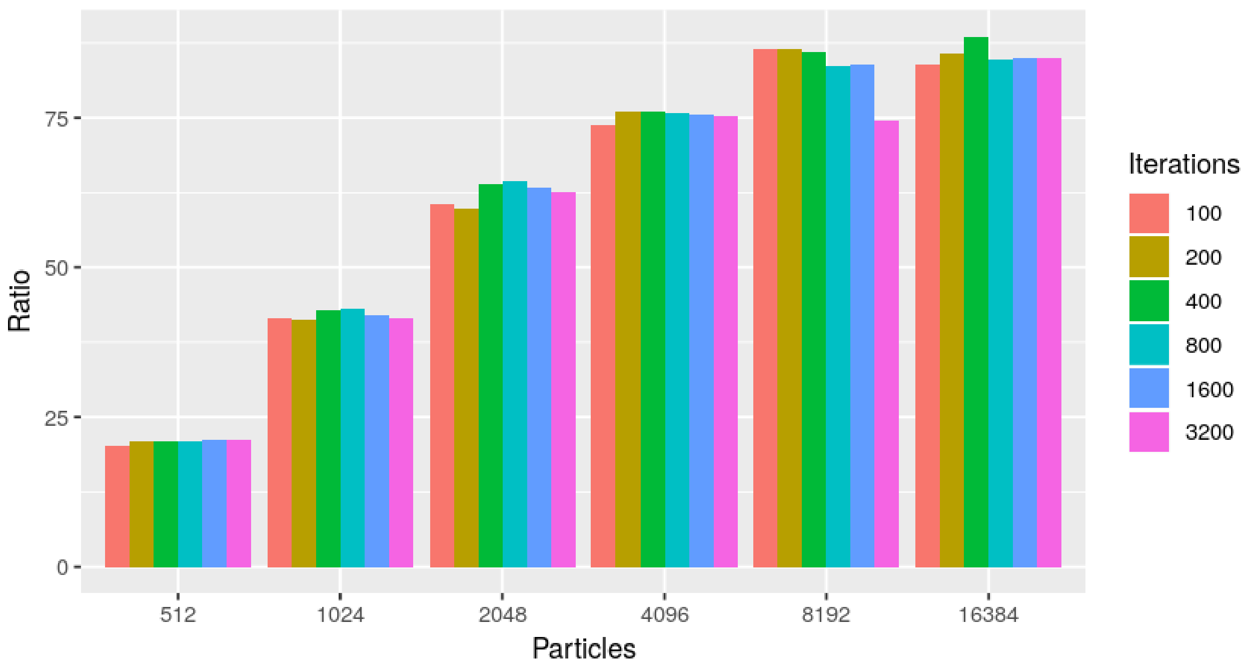


Figure 5.13: Running Time CPU Over GPU Ratio, with Different Iterations, On Testing Machine Cedar

starts to decrease due to a struggle in handling the excess of concurrent tasks. With the maximum particle count (16384), this overhead leads to an improvement ratio the same as that secured when the device is underutilized with just 512 particles.

5.3.2 Profiling and Tuning

We revisit the CPU profiling method used before in the previous chapter when assessing the original algorithm before moving to GPU. Recall that the Flame Graph describes how many times a specific functional frame is on the stack, and we observed the calculation of model stocks and flows take the majority of time in the non-parallel version of the code. In Figure 5.16, when generating the Flame Graph again with the new GPU codebase with influenza model, we can see that the CPU profile tool *perf* is not able to capture the GPU function invocations, but result can still offer some insight on how the functions are called. For the CPU usage, it is clear that **cudaMemcpy** and its children function taking the majority of the CPU time by staying on top of the stack in the flame graph. Those memory-copying origin from two functions: the function for resampling and the function for starting the state variables update. This distribution of CPU execution time is expected as those are the two functions that have most of the device/host communications within a PF step of the PMCMC process; after copying the memory the GPU takes over the computation tasks.

CUDA toolkit includes instruments and profiling tools in command-line interface (CLI) and as GUI application. They can track function calls, memory usage, processor usage and help understand execution

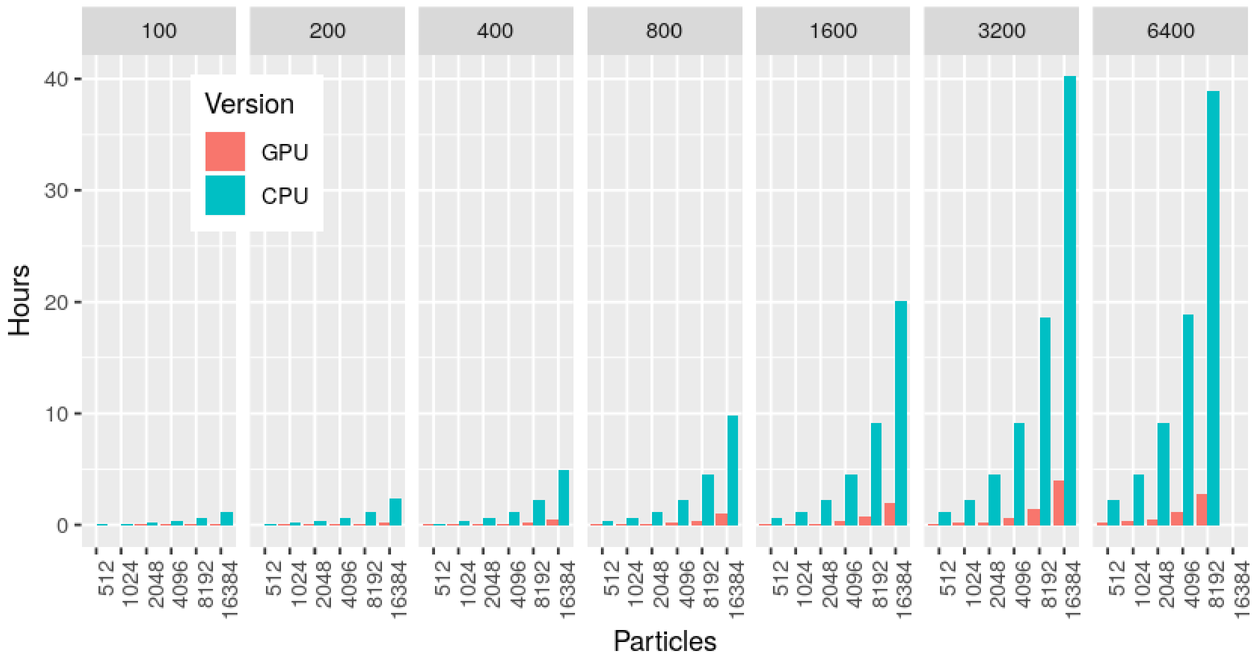


Figure 5.14: Running Time on D1, CPU vs GPU, with different iterations

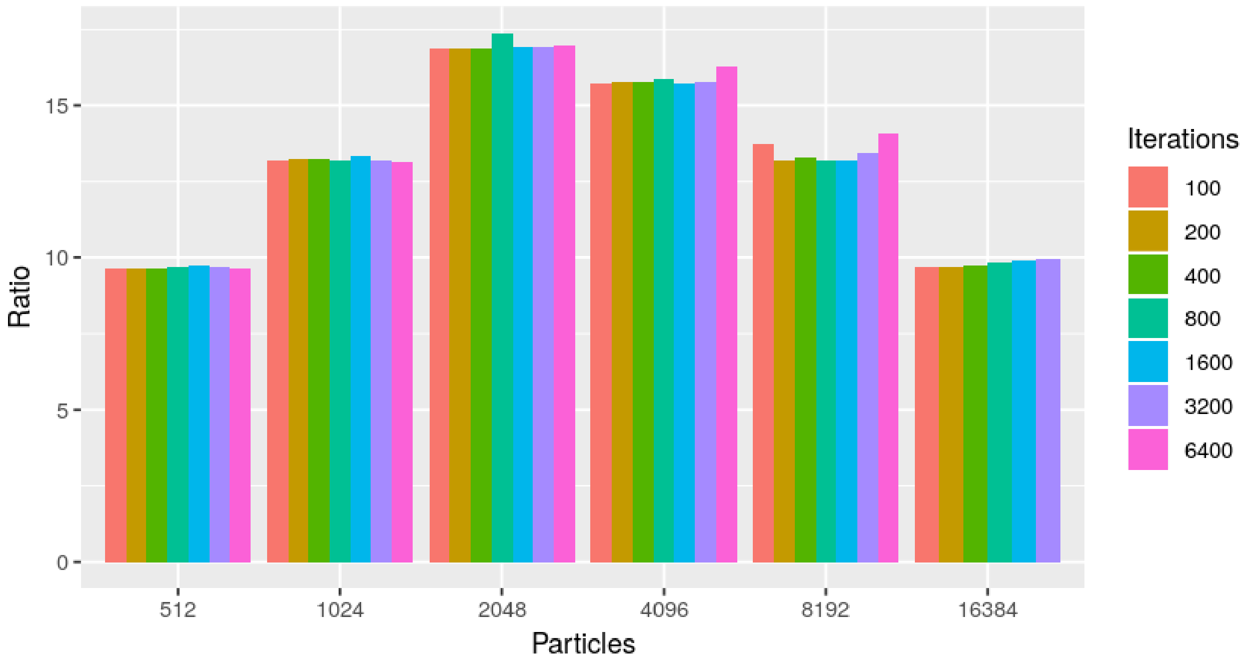


Figure 5.15: CPU over GPU Running Time Ratio on Testing Machine D1

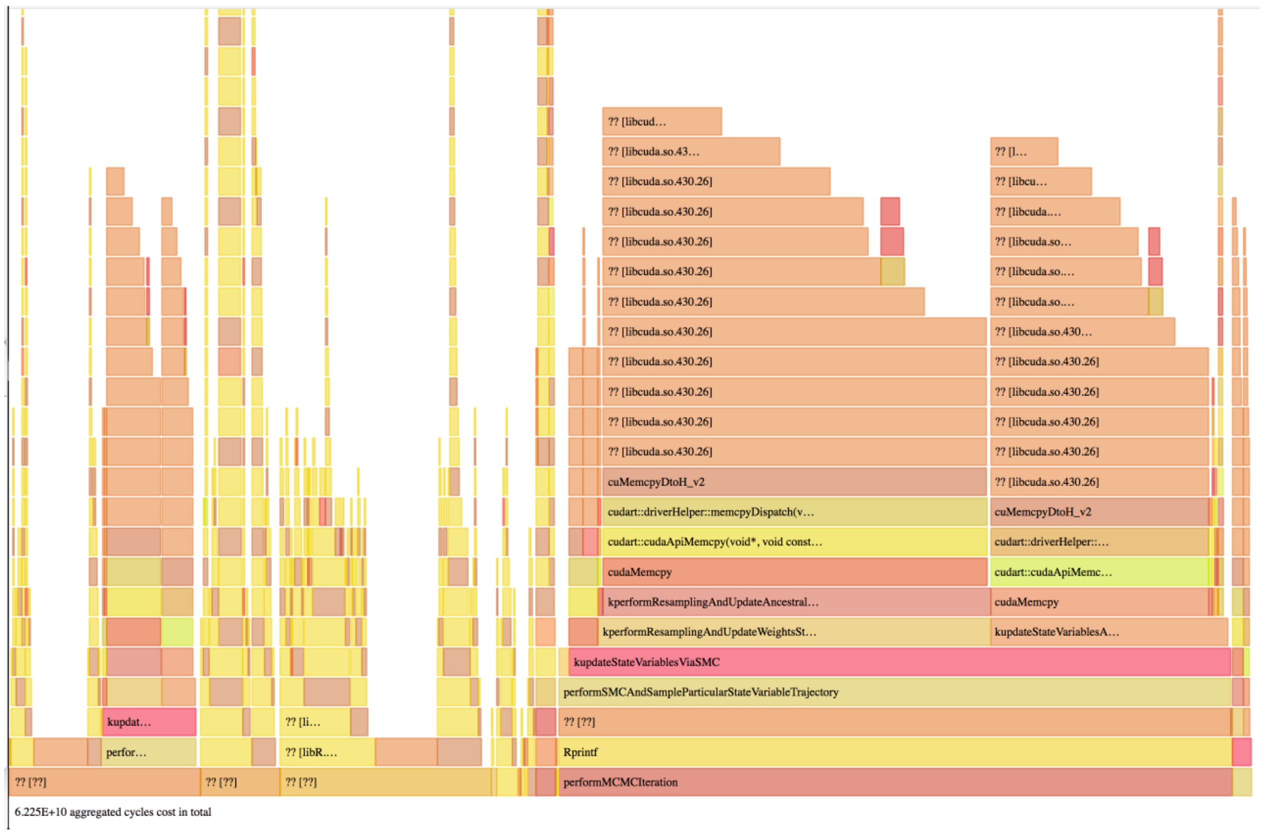


Figure 5.16: Flame Graph of Profiling Results for GPU Runs

pattern and reveal potential bottlenecks in the code. After examining the CPU calling stack and time usage for the CUDA implementation, now we can use the profiler comes with CUDA, **nvprof**, and its visualization tool **NVVP**, to look at the GPU time and resource consumption by analyzing the CUDA kernel calls. In Figure 5.17, we show the **NVVP** visualization of 45 seconds model execution, on testing device **D2** with a device of limited memory size, bandwidth, and core count. Unlike the Flame Graph, the x-axis is representing the time into the execution and each function percentage of time consumption; the calling stack is not shown. Each row on the y-axis represents the timeline of kernel function execution; the percentage number showed on the title indicates the percentage of execution time, while the colored bins on the timeline indicate the start and end time of each kernel function. First, the *MemCpy* section includes three items: host-to-device (HtoD), device-to-host (DtoH), and copying-within-device (DtoD); the sampling shows that the *HtoD* and *DtoH* copying happens before and after calculations of each time step on GPU. In the zoomed-in view at the bottom, the GPU was performing calculations until 5.15 seconds from the start, and then performs a DtoH memory copy to move the results back to CPU. After updating the weights, a HtoD copy taking place from 5.19 seconds to 5.21 seconds sends the values to GPU and calculation is started of particles for another time step. Second, under the *Compute* section, it is indicated that the most time-consuming function is the one that computes the multinomial draws, which corresponds to the resampling. The second most one is the numerical integration of differential equations, which corresponds to the state variable update; the memory copy operations for those two functions are ranked at the top of the CPU Flame Graph just shown. The rest of the kernel functions take no significant time. This points to directions for future work where the focus can be put on optimizing the distributed multinomial sampling and the integration of the equations. Third, the profiling results demonstrate a clear interleaving of computation and communication, suggesting that there is room to improve the occupancy of the GPU resources by overlapping computation and communication. Because of the formulation of the Particle MCMC algorithms, while the calculation of each particle's evolution is independent, the calculation of each time step needs to be performed in order. One way of achieving such overlapping in the future could be adapting pre-fetching in the MCMC process [118, 79] variation. Since there are only two possible outputs for each MCMC iteration — namely, accept or reject the current sample — we can essentially speculatively pre-calculate several iterations of the algorithm. With this approach the GPU can perform computation for one MCMC sample while copying the data for another sample to increase the occupancy of the device. Such parallelization via pre-fetching also provides for easy adaption of multiple GPUs and multiple CPU cores for further speed-up.

As discussed previously, when running CUDA kernel functions on GPU, there are several more layers of resources we need to consider. Different GPUs have a different number of streaming multiprocessors. Each such SM contains a different count of streaming processors and CUDA cores based on generations and computation compatibility. When launching a CUDA kernel, it is required to specify the grid and block size. The block size indicates how many concurrent calls each SM would run at a time. Usually, this value is a number between 1 to 1024; if the block is specified using three dimensions grid, then its product should

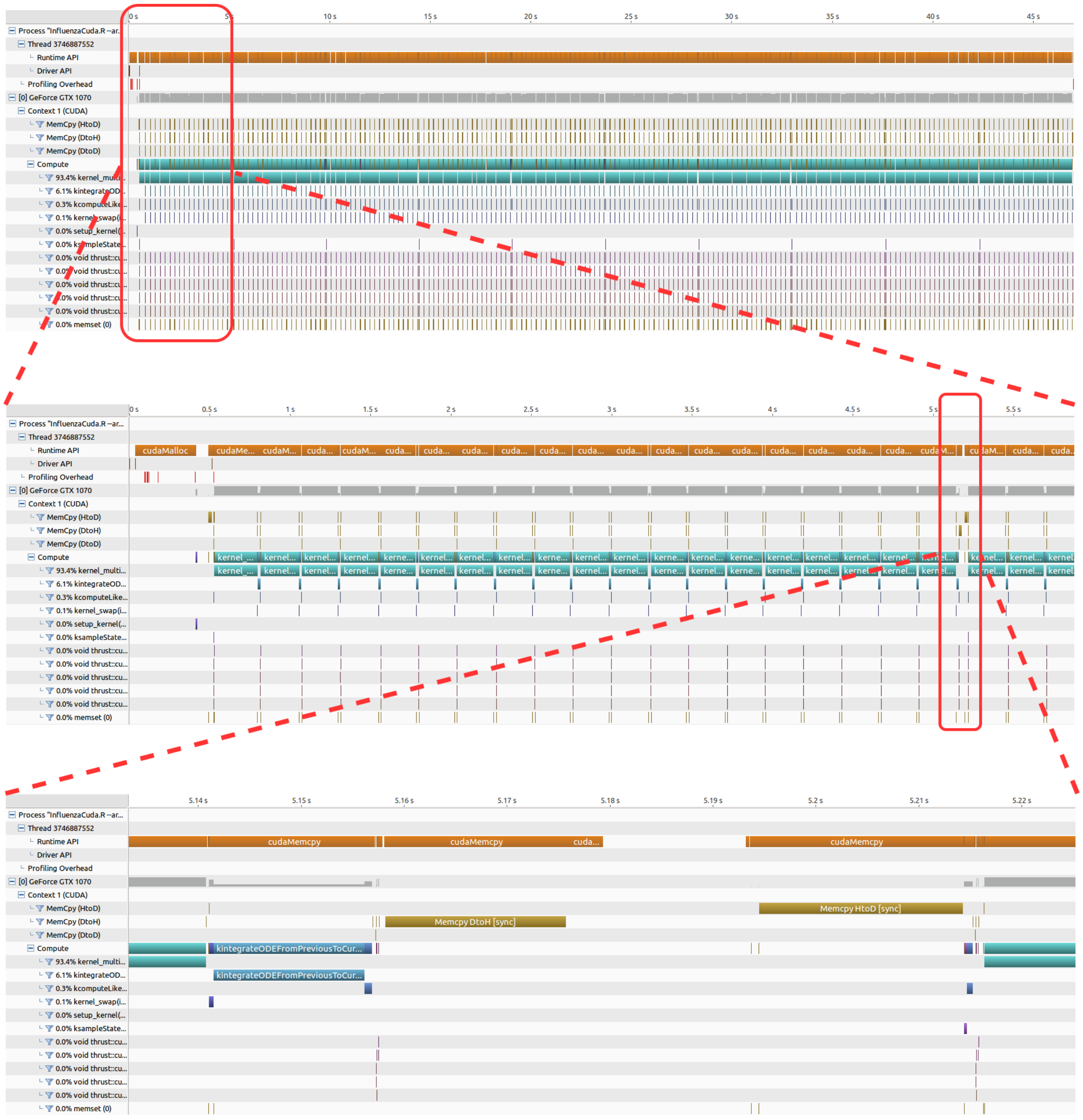


Figure 5.17: CUDA NVVP Visual Profiling Results for the GPU Experiment.

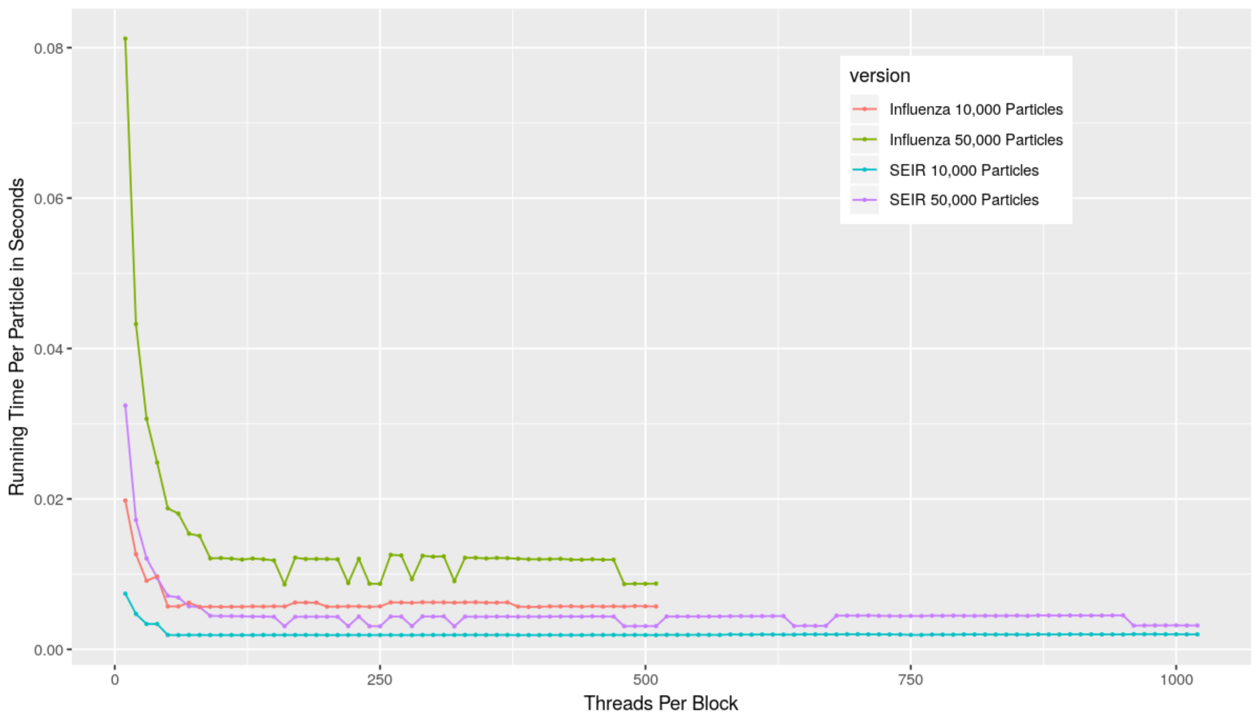


Figure 5.18: Running Time When Varying Thread Per Block On The GPU

be less than 1024. Besides the block size's hardware limit, the number of concurrent calls to launch is also limited by other resources such as memory; if each kernel call takes a more considerable amount of memory space on the SM, then less will be able to be launched at a time. To better understand the accelerated MCMC program's performance, we also test the model with a different block size setting. We want to point out that this is an adjustable parameter that depends on the GPU device type, and it should be calibrated per device. The experiment is set up as follows. A total count of particle is selected as N; for each thread per block $Thread_{size}$, a multiplying that is closest to N is selected to pass down as the total particles $N_{selected}$; e.g., when $N = 10,000$, then $Thread_{size} = 100$ would have $N_{selected} = 10,000$ and $Thread_{size} = 300$ would have $N_{selected} = 9,900$. This can guarantee the balance when execute each block on the device.

The experiment results are collected and presented in Figure 5.18. On the x-axis, we have increasing threads per block count, from 1 and up to the hardware limit 1024; on the y-axis, the running time normalized by the particle counts is shown for each choice of the block size. Four configurations are tested and displayed here for the two models with 10,000 and 50,000 particles, where each point represents one running time per particle given the threads per block. When comparing 10,000 and 50,000 particles, it is clear that no matter the model, the 50,000 takes more time because of other overhead costs such as memory allocation, copying and CPU code. The 50,000 one also requires a larger thread size before reaching a stable running time. For the influenza model, since the kernel function is much more complicated because of the count of stocks and flows, the program failed to finish with a thread size larger than 520. When choosing thread per block parameters, we should avoid smaller thread counts such as those below 50. Also, the larger the total tasks to launch, the bigger thread per block we should choose, as we can see that from the two 50,000 lines. Though choosing a number close to the hardware limit does not affect efficiency, one should know that other hardware resource limits, such as the memory size available for each thread, can prevent a kernel task from launching.

5.4 Streaming Particle Markov Chain Monte Carlo with Influenza Model

By varying the time series length for the influenza model, we can test the effect of streaming input for PMCMC for complicated models. Similar to before we set up the streaming test for the SEIR model, the main matrices taken into considerations include MCMC diagnostic, the acceptance rates, and the visualization of the trace plot of the MCMC. We want to experiment and observe whether executing the model with streaming configuration can result in the simulation's MCMC process to convergent quicker and reach the desired acceptance rate with fewer burn-in iterations and thus take less time overall to finish.

5.4.1 Streaming System for Influenza Model

To set up the complete streaming environment for the PMCMC influenza model connecting to live and real data sources using the streaming system described in chapter 3, we would need to design and implement data collecting adapters for each of the three data sources. We will describe the adapters in detail; all the services, adapters and models are managed by Docker Compose file. The same streaming controller was used for the influenza PMCMC model to connect and consume the live data sources.

Google Search Trends for Flu-Related Keywords Google Trends[3] offers easy-to-access online services for querying, filtering and comparing population search interest based on time and regions. While all data can be generated and downloaded from the web page to collect real-time trend information for flu-related search keywords automatically for the streaming system, an adapter is built using Node.js. This adapter will collect and publish the time series to the message channel in the central hub of the streaming system. Node.js was chosen for this task because the task is relatively simple and requires less computation power, and Node.js provides us with an easy way to deal with the web requests for the communication with the Google Trends web service. Specifically, a Node.js library, **google-trends-api**[4] was used to interact with the Google Trends web service and fetch the data based on query parameters specified. For us, the parameters include the searching keywords related to influenza, geolocation, and time. Data are fetched for a time span of one month at a time to maintain the consistency of normalization of query results, and weekly data are calculated incrementally based on the fetched results.

[3]Google Trends: `https://trends.google.com/trends`
[4]Google Trends API Node.JS Library: `https://github.com/pat310/google-trends-api`

Twitter Keyword Counter Using Apache Storm For the time series for Twitter keywords related to influenza, the harvesting and counting app is built on Apache Storm for the best performance and scalability. Compared to the Google Trends collecting task, the computation and stability requirements are higher, and extra attention is needed when designing the adapter. The harvesting service using Twitter4J library [5] to get Twitter tweets as a real-time sample stream from the Twitter official sampled streaming API[6]. The counting service will then scan over the tweets and accumulate to report the number of instances containing the keywords of interest; the Twitter sampled streaming will emit roughly 1% of all tweets as documented by Twitter. The Apache Storm adapter's topology has a set of *spouts* connect to the Twitter sampled streaming API, and then passing through a pipeline contains tokenizing, counting, and windowed accumulating *bolts*. Finally, the data is sent to the central hub of the streaming system using a client *bolt*.

Clinic Cases from Government Report To collect the clinical empirical case count for the province, we build a streaming system adapter following the specifications in chapter 3 that fetch and process data directly from the public-facing webpage. The adapter uses a Python script to download the weekly report in the format of an image from the aforementioned Government of Saskatchewan website[7], and then corp the embedded image in the report precisely to get the area that contains the usable data. The adapter will then using the OCR library Tesseract [8] through its Python binding Pytesseract [9] to recognize the number and publish the message channels to be consumed by the streaming controller of the influenza model. Figure 5.19 summarizes the streaming system adapter for collecting clinic reported cases.

Streaming Setup for the Simulation Model To have the GPU accelerated the PMCMC model working with the streaming system proposed in previous chapter to consume the streaming time series, We implemented the model as follows when running the experiments. The first thing needed is an early stop mechanism to control the running of the PMCMC; the algorithm will continue to loop and explore the target until new data become available since longer chains usually provide better inference results. Thus, when a new data point becomes available, the current run needs to be stopped immediately and move to the run with the new data. Using POSIX signal to work with R's C extension proven to be the best feasible method. Also we need to make sure only include finished iterations when plotting the graphs. The second step is to write a Python supervisor to monitor and consume the new data points. The controlling script is also able to save the values of the final sample from the previous run that can be used as the starting point for the new run with more data points.

[5]Twitter4J, A Java library for the Twitter API, `http://twitter4j.org/en/`

[6]Twitter Sampled Stream: `https://developer.twitter.com/en/docs/labs/sampled-stream/quick-start`

[7]Saskatchewan Weekly Influenza Surveillance Reports: `https://www.saskatchewan.ca/government/government-structure/ministries/health/other-reports/influenza-reports`

[8]Tesseract Open Source OCR Engine: `https://github.com/tesseract-ocr/tesseract`

[9]Pytesseract: `https://pypi.org/project/pytesseract/`

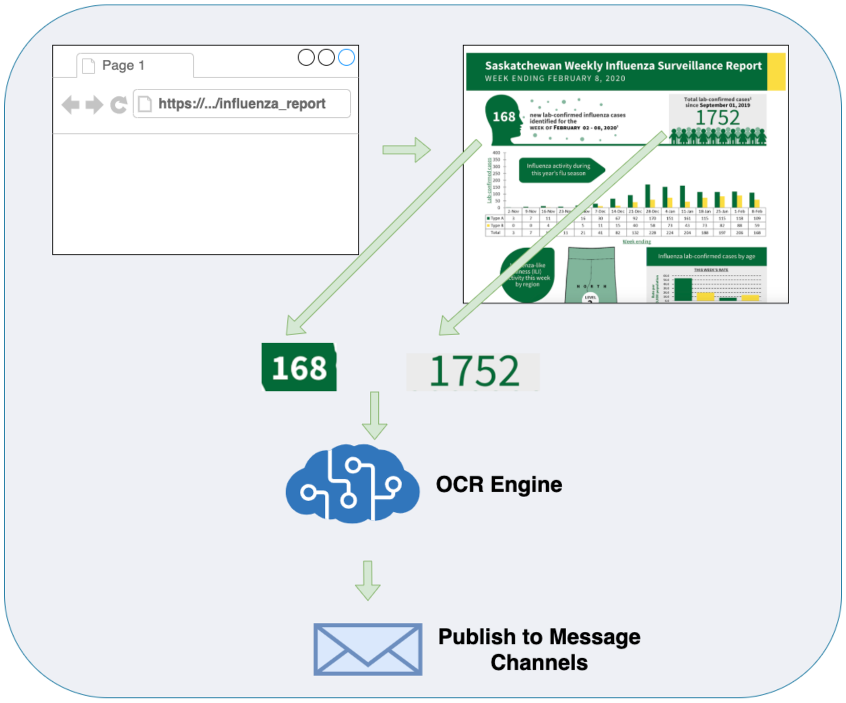

Page 1
https://.../influenza_report
Saskatchewan Weekly Influenza Surveillance Report
WEEK ENDING FEBRUARY 8, 2020
168
1752
168
1752
OCR Engine
Publish to Message Channels

Figure 5.19: Streaming System Adapter that Collects Clinic Cases from Government of Saskatchewan Website

5.4.2 Experiment Results

The experiment setup is similar to that undertaken in the previous chapter. To summarize, we want to explore the benefits of carrying the final parameter vector sampled from the posterior distribution for the previous PMCMC chain considering a shorter time series so as to serve as the initial parameter vector sampled from the prior distribution for a PMCMC chain including additional, newly received, data points. As in the previous chapter, we will be seeking to asses benefit measured in terms of acceptance rates and based on other MCMC diagnostics. The two situations for comparison are as follows:

- Carrying samples, which means when a new data point becomes available, the new MCMC chain is started using an initial MCMC parameter vector that is the final sampled vector from the previous MCMC chain that runs without the new data point;

- Not carrying samples, which is the ordinary case where a new MCMC chain needs to be initialized from a preset location, and does not use any samples or information from the previously conducted shorter time series runs.

The results comparing these situation with the influenza model considered in this chapter are displayed in Figure 5.20. The top two plots show the acceptance rates over ten repeated experiments, while the bottom one compares the mean value of acceptance rates over the repeated experiments for both settings. Note that we start each experiment with time series including ten time points and increase the time series length one by one with successive observations until all 20 weekly data points are available; to initialize the experiments, the single PMCMC variable in the vector is set to 7.2. MCMC chains are all of length of 3000; this length is considered to be a short chain — normally more iterations are needed for this kind of simulation models. The short chains can help evaluate the impact of the starting points, but also could result in unstable acceptance rates; those are the outliers in the box plots. This set of experiment results shows that, though the differences between the two settings' overall acceptance rates are within the accuracy of the model, carrying samples can reduce the number of outliers, especially for longer time series.

Next, if we look into details regarding results from one experiment with a streaming run from time series of size 10 to time series of size 20, as in Figure 5.21, we can see that the evolution of the effective sample size exhibits pronounced differences between experiments involving carrying over samples vs. not carrying over the such samples. The upper two plots display the trace of PMCMC sampled parameter values for carrying and not carrying the samples, where the x axis is the iteration and y axis is the parameter value. The lower plots show the ESS with respect to the length of burn-in period for the two configurations, where the x axis is the number of iterations used as burn-in period for this chain and y axis is the ESS value. On the left panel of that lower plot, the ESS is relatively high when carrying the samples, no matter how small the count of burn-in iterations employed. That means that now there can be little to no need for a burn-in period, and carrying the samples can help the MCMC chains have a larger number of effective samples, which means the chain can more quickly converge to mixing well within the chain. The right panels of the lower plot shows when the samples are not carried over to inform the more extended time series runs. It is clear that a burn-in period of 50 to 150 iterations will be needed for each of the individual chains, which results in the slower reaching of the desired status for all the lengths of time series. The trace plot in Figure 5.21 shows similar results. On the left side, where samples are carried over, only the first chain needs to find its way to the true value. By contrast, on the right panel, each MCMC chain will use the initial period to move towards that value.

Regardless of whether we are considering the carrying and non-carrying configurations, for each repeated experiment, the final chains run consider all observations. Figure 5.22 depicts the trace plot, density and ESS precisely for these final runs of each repeated experiment. The density plot on the top right shows that while those two will reach similar final distributions of sampled parameter values, the carrying one does not have a long tail on the right side. The carrying and not carrying configurations should converge to the same distribution because those two are only differ in the starting point and by the definition of the PMCMC algorithms, under broad assumption and with sufficient samples, they should all have the same equilibrium distribution. The ESS plot shows similar patterns to those exhibited above, wherein the non-carry configuration needs around 100 burn-in iterations. By contrast, carrying samples can eliminate this period for most of the final runs. However, when carrying over the sample, we observed one case exhibiting a low ESS. This likely represents a situation where the chain attains a parameter vector offering an excellent value of the posterior at the start, and exhibits difficult in finding better parameter vector; as a result, the

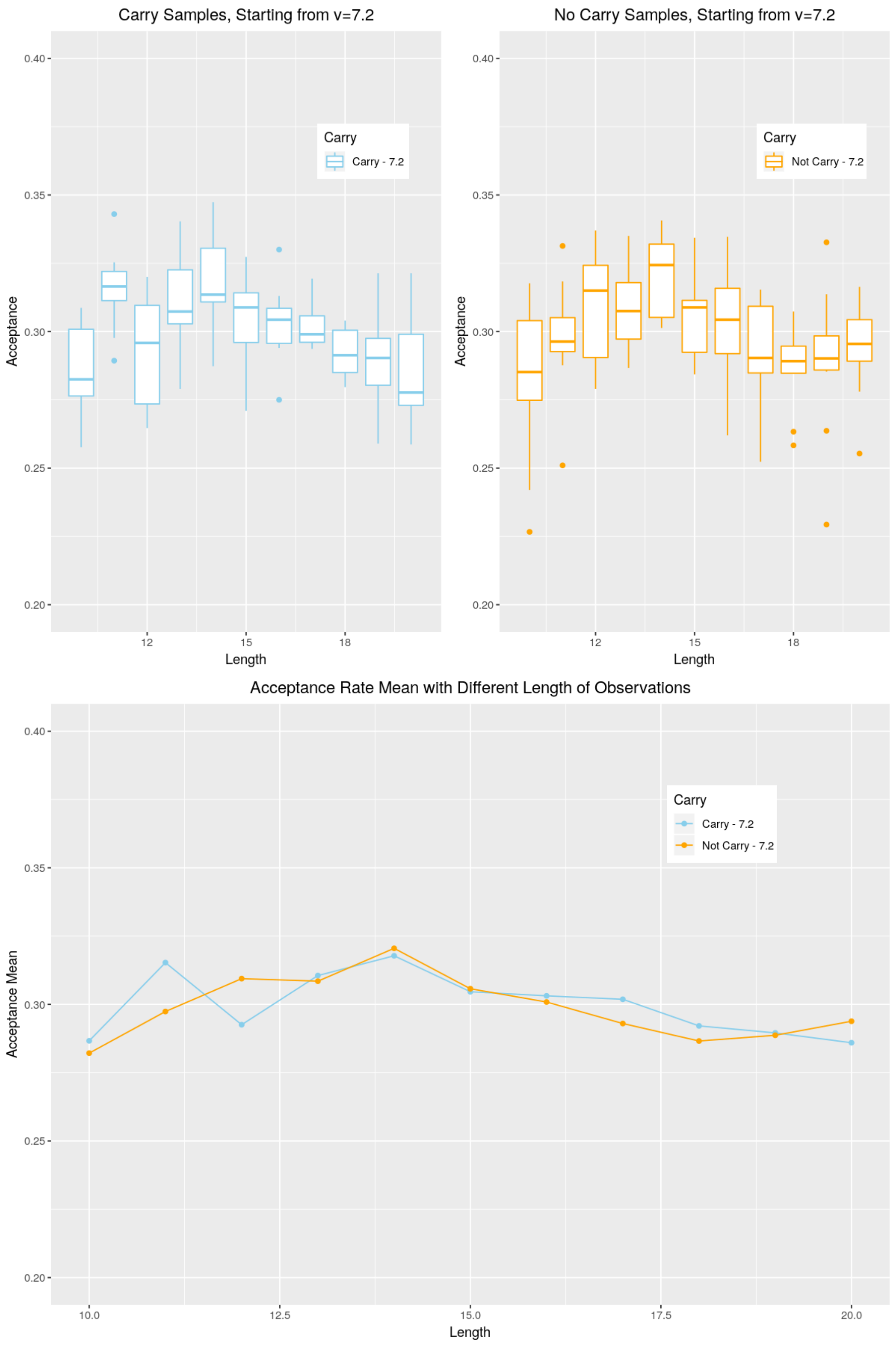


Figure 5.20: Comparing the Overall Acceptance Rate for the Influenza Model for Carrying and not Carrying Samples

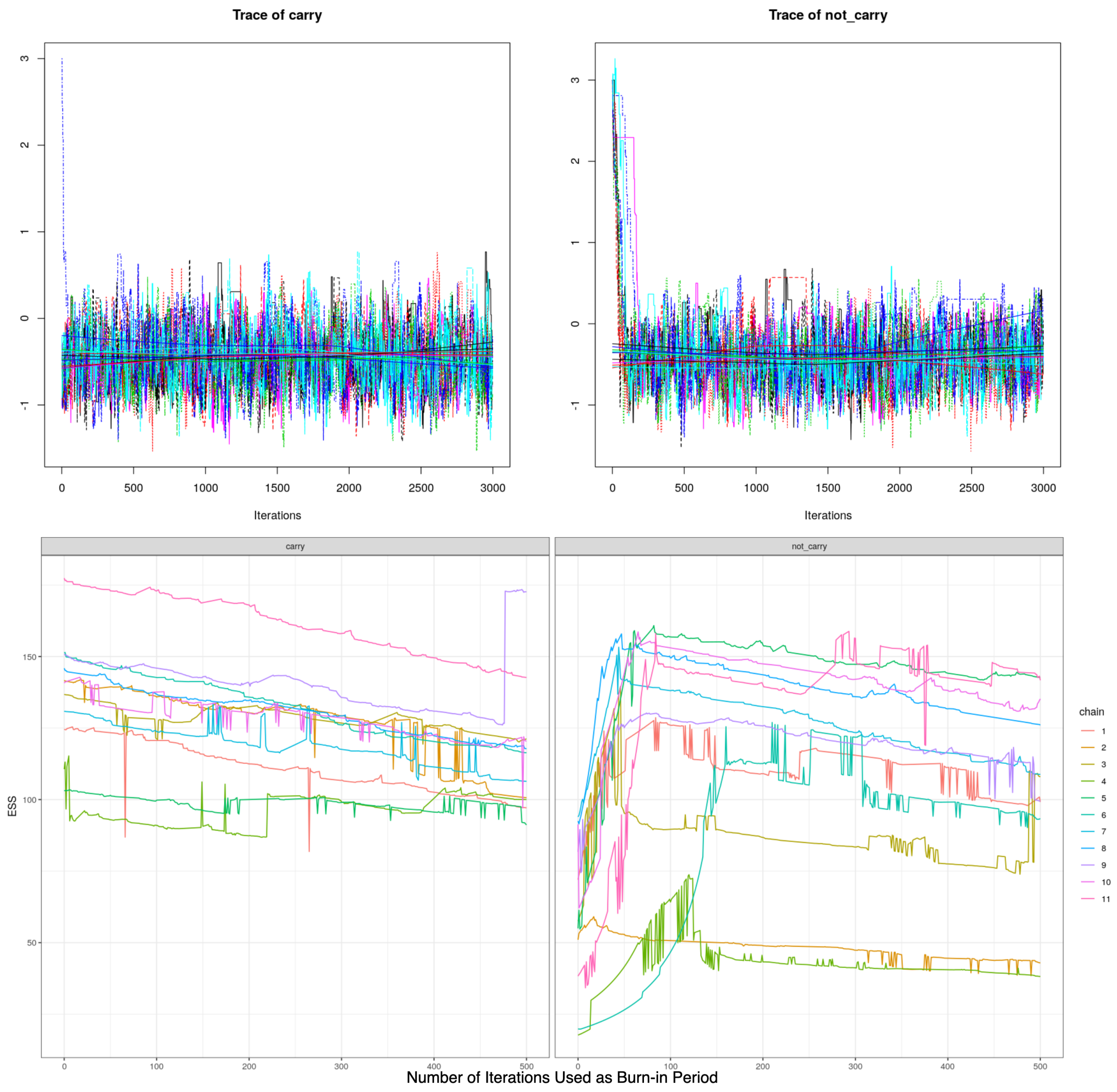


Figure 5.21: MCMC Trace for the Influenza Model and Effective sample size for the Influenza Model, for one streaming series

chain exhibits lower sampling variability and ESS from the start. The density of the MCMC trace shows a similar density for both configurations after those 3000 MCMC iterations. However, the non-carry runs start with markedly different values on the left side of the density plot as the results of the run's initial period. While carrying samples can reduce the burn-in and establish good ESS starting from the earlier stage of the MCMC runs, it is not unexpected to see that even with carrying samples, certain chains may still suffer poor performance because of other experiment variables. The notable variation in results obtained from runs even for the carry-over case is reflected in the ranges of ESS exhibited on the left side of Figure 5.22.

Finally, the Gelman-Robin convergence diagnostic for the multi-chain convergent test is shown in 5.23. The median and 97.5% quantile of potential scale reduction factors (PSRF) are presented. The x-axis represents the iterations into the MCMC chain, and the y-axis represents the potential scale reduction factors after those iterations; the 97.5% quantile is needed as the reduce factors are estimated based on the finite length of the MCMC chain but not an exact solution. High reduction factors mean bad convergence while it is usually considered good ones and reached the stationary distribution at values around 1.2 or lower. For the carrying sample one, the PSRF starts lower at three and reduces to around 1.2 with about 100 iterations. Also, it takes much more iterations for the non-carrying one to start to see lowering PSRF; the value stays high within the first 100 iterations and takes another 60 iterations before it reaches an acceptable value as the carrying one. This confirms the benefit of carrying samples in the current experiment settings: the one with carried samples has notable improvements in multi-chain convergence.

In this section, by collecting experiment results of streaming influenza PMCMC and examining those from different angles, we can notice how the carrying sample method can affect the MCMC process and has the potentials to improve the PMCMC with system dynamic simulation models algorithm's performance. On top of the already 30 folds time-saving CUDA implementation, this could further shorten the running time and resources needed for carrying out those experiments. It is worth noting that the results and experiment setup are preliminary, and certainly, much more comprehensive explorations are needed before we conclude this carrying samples method.

5.5 Discussion of the Results

5.5.1 Performance Summary

As illustrated by the performance experiments of the two distinct models in the previous two chapters, we can observe many similar performance patterns. The CUDA GPU implementation can offer much more improvement for a model with complex structure and complicated ordinary differential equations. Also, by showing GPU profiling results, we want to communicate how to verify whether the implementation is optimized and find potential future improving directions. By presenting the performance patterns, as all those experiments on a variety of testing environments and type of GPUs with different counts of core, it is easy to make predictions about the performance as hardware improving. With increasing count of more

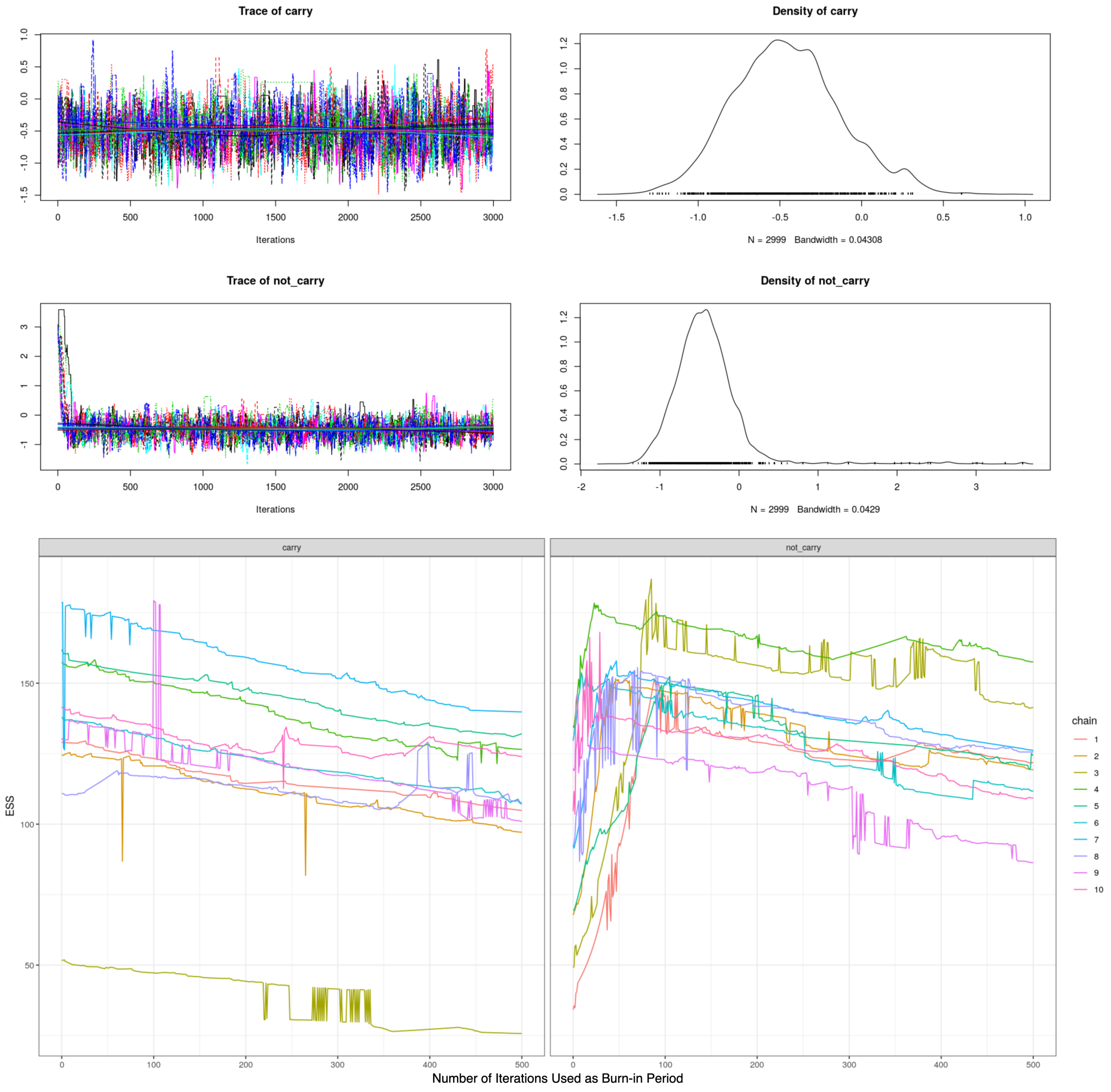


Figure 5.22: MCMC Trace, Density, and Effective Sample Size for the Influenza Model, for all final length

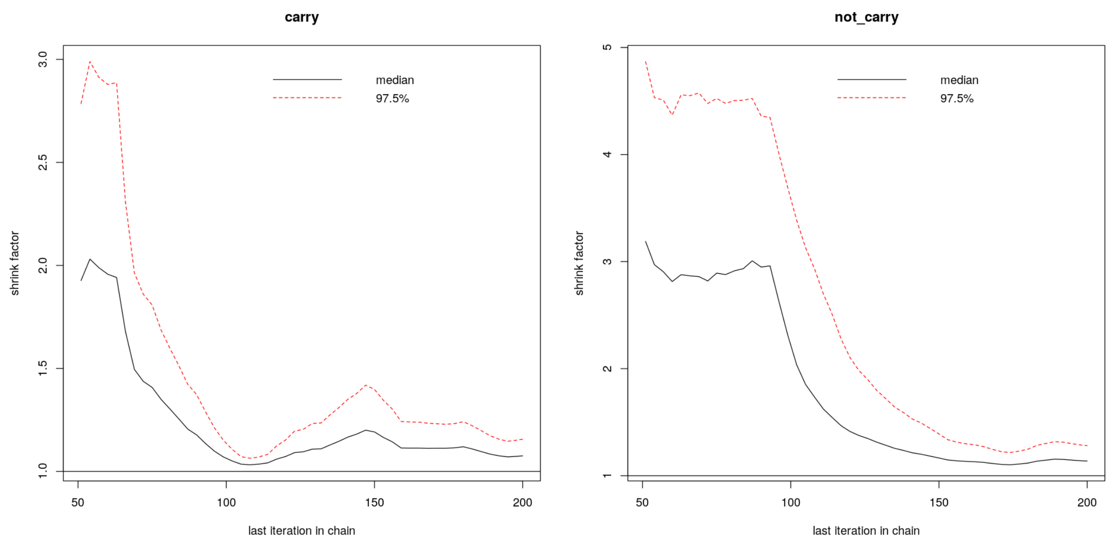


Figure 5.23: Gelman Diagnose for the Influenza Model on all final lengths

advanced cores and faster, larger memory available in the future than the current hardware, we will be able to scale up the experiments and use more particles and get results faster.

Between the two models, as anticipated and seen in other MCMC GPU implementation, the improvement is only worth the effort with a large particle count and complicated models to mitigate the overhead rooted from the communication and memory[8]. Another notable observation from our experiment is that the scalability of the CUDA-accelerated PMCMC. The design philosophy of CUDA provides an abstracted higher-level programming interface independent of the actual scale and layout of the parallel cores in the underlying hardware. By doing this, when new and improved GPUs become available, an exemplary and scalable implementation of an application can take advantage of the increasing in cores and memory right away without any significant change of the code. Our testing results, carried out among several generations of NVIDIA CUDA-enabled GPUs, shows that the implementation can indeed follow this.

We would also like to mention the comparison of performance in similar implementations briefly. One such implementation is described in [119] with several examples available. We can compare the running time per particle per simulation step per MCMC iteration and compare the speedup ratio from CPU to GPU. For *libbi*, with GPU, the running time is 15 seconds for 500 PMCMC iterations with 8192 particles and running with only two simulation time steps, with integration step size 0.05, with one stock and two parameters using the Lorenz '96 model, on NVIDIA Tesla C2075. It also has a running time of 250 seconds using one thread on the CPU. For our implementation, on the slowest machine we tested, the SEIR model's running time with four stocks, one parameter, 4096 particles, 800 iterations and 32 time steps, and 0.01 integration time step is about 400 seconds. Those two models are comparable in terms of size and complicity, and the results are summarized in table 5.4.

5.5.2 Streaming Particle Markov Chain Monte Carlo Summary

The performance improvement of the PMCMC code with System Dynamics simulation models enables us to experiment with variable-sized time series input data to the algorithm. The novel approach of carrying samples from experiments running with shorter time series data and using the samples as the starting points for the experiments with longer data has been evaluated using different metrics, including acceptance rates, visually checking of trace plots, effective sample size and Gelman-Rubin convergence diagnostic.

The effect of running the PMCMC algorithm with an incremental length of time series data has been explored to mimic PMCMC in an online learning setup. This has not been done before as far as the authors know. With two different compartmental simulation models, this work demonstrates the generalization of the approach. Though there are differences between the different models presented in the previous two chapters. those preliminary evaluations indicate that there are potential benefits with incremental time series data for running long PMCMC simulations, and the degree of the effect can correlate to the complexity of the system.

Table 5.3: Running Time Across Different Machines for the Influenza Model

4096 Particles, 800 Iterations			
Testing Machines	Mean CPU Time	Mean GPU Time	CPU/GPU Ratio
Azure	10117s	296s	34
Beluga	14405s	105s	137
Cedar	10127s	167s	60
D2	7067s	260s	27
D3	13922s	533s	26
Graham	10022s	166s	60
K2	17290s	400s	43
32768 Particles, 100 Iterations			
Testing Machines	Mean CPU Time	Mean GPU Time	CPU/GPU Ratio
Azure	10191s	594s	17
Beluga	14505s	91s	159
Cedar	11656s	163s	72
D2	7098s	576s	12
D3	13925s	1284s	11
Graham	10054s	152s	66
K2	17419s	786s	22

Table 5.4: Comparison with Similar Bayesian GPU Accelerated Implementations

Implementation	Model	Unit Running Time (Per PMCMC iteration, Per Time Step, Per Particle)	CPU Time over GPU Time
Current	SEIR	[3.815 x 10-7, 3.815 x 10-6]	[2x, 38x]
Libbi	Lorenz '96	1.831 x 10-6	16x

6 Conclusion

6.1 Synopsis

Low-latency streaming computing and analytics have become increasingly desirable in many research fields, and offer great potential for modern public well-being and social science research. With a large and growing number and diversity of informative but voluminous datasets available to researchers, efficiently and effectively processing the data and developing solutions to solve real-world problems are more challenging than before. Common barriers include time- and finance-inefficient manual steps within the workflow, long developing and configuration periods, and the efforts required dealing with software and systems designed to only work for specific problems one at a time, and much more. Streaming data analysis with simulation models and Bayesian statistic methods has demonstrated great potential in recent public health research. Powerful Bayesian methods enable simulation models to respond to unfolding time series of data and constantly reground themselves to produce more accurate and relative results. However, such methods usually come with a heavy computational burden and require long developing and experimention cycles and expensive computation hardware.

This thesis's work focused on combining streaming processing, System Dynamics models, and GPU accelerated Bayesian statistical methods of Particle MCMC and applied them to real-world problems. After reviewing the current literature on streaming processing and Bayesian methods with simulation models, several distinctive key research questions were identified to improve and accelerate the process. A specification of the algorithms was provided, followed by a discussion of the essentials of GPU programming. Chapter three introduced the available modern tools, methods, software, and hardware that can be used to tackle those issues. On the basis of researching and reviewing existing solutions and software, a design of a streaming analysis system was presented. This was followed by case studies and evaluation. Chapter 4 characterized the GPU-accelerated PMCMC algorithm with the dynamic models. Leveraging the GPU-accelerated MCMC implementation, Chapter 4 further investigated the implications of streaming, incremental runs of iterative Bayesian methods on state inference within the context of public health simulations. A complete real-world application of influenza transmission was discussed in Chapter 5. That chapter also offered additional performance evaluation to demonstrate the improvement and performance patterns associated with the CUDA-accelerated application.

The streaming system design proposed satisfied all the requirements outlined; the architecture was presented along with strategies for maintenance, expansion, and operation. Streaming processing pipelines are

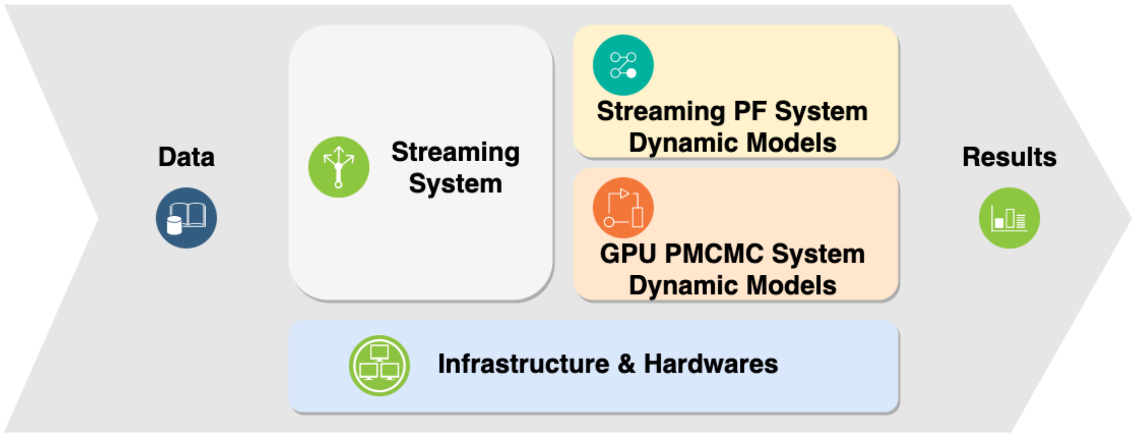

Data
Streaming System
Streaming PF System Dynamic Models
GPU PMCMC System Dynamic Models
Results
Infrastructure & Hardwares

Figure 6.1: Architectures and GPU-Based Parallelization for Online Bayesian Computational Statistics and Dynamic Modeling

extremely valuable – and likely essential – for achieving automation in modern research and deliver simulation models as services at scale. The data volume available will only grow and reach a level that for which manual processing is no longer feasible. The proposed solutions for each aspect have been presented clearly and in a straightforward fashion. Security, automation, and efficiency will always be at the centre when implementing methods and applications.

By studying two different kinds of Monte Carlo methods – namely, the non-iterative Sequential Monte Carlo in the form of Particle Filtering, and iterative Particle Markov Chain Monte Carlo – this thesis explored strategies for applying streaming data to different Bayesian algorithms. The GPU acceleration of Particle Markov Chain Monte Carlo not only demonstrates techniques for and the gains to be expected by speeding up computationally expensive methods using commonly available hardware, but also enables us to investigate questions that were hard before, such as those regarding the influence of the incremental size of data, to how to expand PMCMC applications with online learning ability. This implementation also illustrated how to use modern software design to boost public health and social data science productivity. Figure 6.1 offers an overall summary of these factors.

6.2 Contributions

Notable contributions of this work include:

- Environmental scan of many software and hardware supports for streaming interfaces to support combinations of dynamic models with Particle Filtering or Particle MCMC.

- The design of an easy-to-use streaming platform for use with those technologies, and offering a familiar interface to researchers;

- Within the streaming system design, a systematic articulation of the considerations for creating a good streaming system, and guidelines as to how to choose frameworks and tools to build such platforms efficiently;

- Illustrated the use of Particle Filtering and Particle MCMC with dynamic models in real-time projection and prediction;

- The first published effort combining large ordinary differential equation-based System Dynamics models with Particle Markov Chain Monte Carlo on graphic processing units;

- A novel approach and preliminary investigation of incremental data input in an online streaming configuration for large ODE-based System Dynamics models with Particle MCMC;

- Demonstration of the performance characteristics of running large-scale compartmental simulation models on GPU on various hardware configurations. The breadth of such configurations examined are such that the resulting findings can confer strong insight into the potential performance gain to expect for different experiment setups, such as the count of particles or the length of the chain, and future scalability in the future if the count of available GPUs is doubled. Such results also offer insights into the effect of models of different sizes.

- Proposed a novel approach to incrementally ingest new data points into the PMCMC algorithm, so as to enhance early mixing of the MCMC chain involved in PMCMC.

- Presented PMCMC applications with compartmental models for joint state and parameter estimation and investigated the effect of variable time series length in the context of this capacity to incrementally ingest new data points into PMCMC.

The performance experiments on various testing environments make it relatively easy to predict performance improvement as hardware advances. One core design philosophy of CUDA that provision of a higher-level abstract programming interface independent of the actual scale and layout of parallel cores in the underlying hardware can allow an efficient and scalable implementation of an application to immediately take advantage of additional cores and memory and absent need for any major code changes. Our testing results, carried out among several generations of CUDA-enabled GPUs, show that the implementation fulfills such aspirations: We can scale up the experiments and particle count for more timely receipt of accurate results.

The capacity to mesh diverse high-velocity datasets with theory serves as a valuable accelerator for social and behavioural modeling. PMCMC's capacity to derive a joint posterior distribution over the latent state of a system over time together with the value of associated static parameters, marks it as a particularly powerful and increasingly prominent method capable of integrating diverse longitudinal behaviorally relevant data with theory captured in the form of a dynamic model. In this work, we provided a novel implementation of the GPU-accelerated PMCMC algorithm with dynamic modeling, demonstrated it with an influenza transmission simulation enriched with social media and online activity data, and performed and reported on various experiments investigating the performance pattern of our work on diverse testing environments. This work made possible prominent provincial and national-level reporting during the COVID-19 pandemic, and has materially facilitated effective adoption of PMCMC to build state-of-the-art simulation models taking advantage of today's abundance of high-velocity social and behavioural data available online, through mobile devices, and elsewhere, while reducing the time needed to develop, tune and deliver such models with a well-considered parallel implementation on accessible and widespread commodity graphical processing units.

6.3 Limitations and Future Work

The work presented in this thesis is intended to offer effective and comprehensive solutions for problems outlined but supports diverse possibilities for improvements and extensions. The streaming software system can be extended in the future to improve performance, flexibility and usability. Several promising approaches bear investigation to further enhance the performance of the CUDA implementation of the PMCMC. Additional experiments and research are also in order to fully understand the impact of streaming PMCMC.

It is a frequent and increasingly common use case to have several simulation models deployed and executed concurrently, focusing on different research questions and scenarios. In the wake of the capacity to support streaming processing demonstrated within this work, we can anticipate a growing number of models to be executed simultaneously and continuously, and support for this capacity within the system designed and implemented in this thesis could further enhance the system's value. Responsive to this, the simulation model's streaming system can be expanded with functions for model orchestration and operation to address the existing system's limitations with respect to deploying, managing, and monitoring those long-run instances. Such a system can support easily submitting and rolling out new models as tasks for models that request frequent changes, such as upon the occurrence of new interventions such as public health orders, or changes in data collecting patterns. The running of the resulting model would then be carried out automatically. Similar to a Kubernetes clusters, if the system can deploy on top of multiple computing nodes, model running tasks could then be distributed over the cluster, in the context of load balancing to guarantee efficiency; further priority levels could then be assigned to the tasks based on the user-assessed importance of the model. With additions such as this, the system can evolve further to achieve the capacity to serve as an always-on, minimal delay, quick response streaming system for simulation models; the system's modular design enables many additions and extensions to be implemented and integrated with much less effort. An important limitation of the current system include the omission of the capacity to handle certain contexts that were judged to occur infrequently. One example applies in cases where the new records arrive faster than the Particle Filter algorithm's computation-limited update rate. Given the high dependence of particle-filtering runtime on the particle count, this is expected to represent a particular concern when a higher particle sample size is required, in the context of a larger model, or (as is frequently required) when both apply. Though our solution can temporarily cache data points in case of ingesting spikes, variations on the algorithm such as Real-time Particle Filter [23] also offer promising avenues towards mitigating this problem.

There are also many future research directions for advancing CUDA GPU accelerated System Dynamics simulations with the PMCMC method. First, other similar methods, such as SMC^2 [120] can also be used with SIR type compartmental models, whilst supporting similar GPU CUDA approaches for acceleration. While the work in this thesis sought to provide a broad set of testing results across different hardware profiles, horizontal scaling of the hardware has yet to be tried and merits investigation. Potential performance gain can be achieved with multiple GPUs on one host communicating over the bus or on different nodes communicating over the network, which increases the allocation of GPU cores and memory to the simulation models and mitigate the resource limitation of a single GPU; we have shown that both resources can strongly impact the running time. In terms of GPU memory, the trend of using host memory directly has started with the increasing bandwidth accompanying PCIe 4.0, and special instructions and buses in CPU and motherboard design; for example, with AMD fifth generation of Ryzen CPU and RDNA2 GPUs, the GPU can have virtually unlimited onboard memory because of the capacity to share memory with the host. SLI is a technology that enables the computation to be distributed to two or three GPUs on a simple computer node; recently, we also have NVLink, which was initially designed for deep learning and parallel image processing tasks. Our final GPU implementation's profiler showed that large portion of GPU work can be found in numerical integration of the ODE for the dynamic models, which points us to a potential direction where we can further speed up the program. One opportunity for speeding up the numerical integration would be to explore the use of the Runge-Kutta method as a replacement for the Euler method. There are many variations of MCMC that can be easily parallelized [121]. One of the particular interests is the prefetching[118, 79] variation. Since there are only two possible outputs for each MCMC iteration, we can essentially speculatively precalculate several iterations of the algorithm depending on the number of parallelizing computational cores available. A coarser-grained opportunity would be to run multiple MCMC chains in parallel on different cores or computational nodes. In the GPU and CUDA context, this can be translated into running on GPU clusters.

The system proposed is not perfect and has much room for future expansion. The hope is that by communicating the design philosophy and findings along the process, more and more of the routes for expansion can be integrated into the current workflow. The effect of streaming Particle Markov Chain Monte Carlo should be thoroughly explored with more simulation models and experiments.

6.4 Final Remarks

This thesis has proposed, designed and implemented a system to take advantage of recent specialized hardware with cutting edge performance power, parallel programming paradigms, virtualization and Cloud Computing to create a next-generation research support for health science and social science data science based around combinations of Bayesian computational statistics methods together with dynamic models; benchmarks confirm that the system demonstrates strong performance advantages relative to sequential execution. Variants of this system have played a strong role in supporting provincial and two lines of national-level reporting during the COVID-19 pandemic; those results and the modularity and extensibility of the platform suggest a strong prospect for broader use of these methods in coming years. The system will help to deliver robust and responsive informative simulations to aid quick and reliable decision-making from still-fresh empirical data while open for future extensions and opportunities.

References

[1] A. Safarishahrbijari and N. D. Osgood, "Social Media Surveillance for Outbreak Projection via Transmission Models: Longitudinal Observational Study," *JMIR public health and surveillance*, vol. 5, p. e11615, May 2019.

[2] NVIDIA, "CUDA C++ Best Practices Guide - Design Guide," Nov. 2019. Accessed on: Feb. 20, 2020. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`.

[3] V. Palanisamy and R. Thirunavukarasu, "Implications of big data analytics in developing healthcare frameworks – A review," *Journal of King Saud University - Computer and Information Sciences*, Dec 2017.

[4] N. J. Gordon, D. J. Salmond, and A. F. Smith, "Novel approach to nonlinear/non-gaussian Bayesian state estimation," *IEE Proceedings, Part F: Radar and Signal Processing*, vol. 140, no. 2, pp. 107–113, 1993.

[5] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[6] C. Andrieu, A. Doucet, and R. Holenstein, "Particle Markov chain Monte Carlo methods," *J. R. Statist. Soc. B*, vol. 72, no. 3, pp. 269–342, 2010.

[7] X. Li, B. Keeler, R. Zahan, L. Duan, A. Safarishahrbijari, J. Goertzen, Y. Tian, J. Liu, and N. Osgood, "Illuminating the Hidden Elements and Future Evolution of Opioid Abuse Using Dynamic Modeling, Big Data and Particle Markov Chain Monte Carlo," *SBP-BRiMS*, 2018.

[8] Y. B. Erol, Y. Wu, L. Li, and S. J. Russell, "Towards Practical Bayesian Parameter and State Estimation," *CoRR*, vol. abs/1603.08988, 2016.

[9] P. Clifford, "On-line inference for data streams," in *Statistical Problems in Particle Physics, Astrophysics and Cosmology - Proceedings of PHYSTAT 2005*, pp. 243–251, Imperial College Press, 2006.

[10] L. Duan and N. Osgood, "GPU Accelerated PMCMC Algorithm with System Dynamics Modelling," in *Social, Cultural, and Behavioral Modeling*, Lecture Notes in Computer Science, pp. 101–110, Cham: Springer International Publishing, 2021.

[11] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, pp. 1087–1092, Jun 1953.

[12] W. K. Hastings, "Monte carlo sampling methods using Markov chains and their applications," *Biometrika*, vol. 57, pp. 97–109, Apr 1970.

[13] S. Geman and D. Geman, "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 6, pp. 721–741, 1984.

[14] K. P. Murphy, *Machine learning : a probabilistic perspective.* MIT Press, 2012.

[15] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking," *IEEE Transactions on Signal Processing*, vol. 50, pp. 174–188, Feb 2002.

[16] Y. Ho and R. Lee, "A bayesian approach to problems in stochastic estimation and control," *IEEE Transactions on Automatic Control*, vol. 9, no. 4, pp. 333–339, 1964.

[17] V. Kostakos, D. Ferreira, J. Goncalves, and S. Hosio, "Modelling smartphone usage: A markov state transition model," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, (New York, NY, USA), pp. 486–497, Association for Computing Machinery, 2016.

[18] K. Kreuger and N. Osgood, "Particle filtering using agent-based transmission models," in *Proceedings - Winter Simulation Conference*, vol. 2016-February, pp. 737–747, Institute of Electrical and Electronics Engineers Inc., Feb 2016.

[19] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte carlo localization for mobile robots," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, vol. 2, pp. 1322–1328 vol.2, 1999.

[20] N. Osgood and J. Liu, "Towards closed loop modeling: Evaluating the prospects for creating recurrently regrounded aggregate simulation models using particle filtering," in *Proceedings of the Winter Simulation Conference 2014*, pp. 829–841, IEEE, Dec 2014.

[21] A. Safarishahrbijari, T. Lawrence, R. Lomotey, J. Liu, C. Waldner, and N. Osgood, "Particle filtering in a SEIRV simulation model of H1N1 influenza," in *Proceedings - Winter Simulation Conference*, vol. 2016-February, pp. 1240–1251, Institute of Electrical and Electronics Engineers Inc., Feb 2016.

[22] X. Li, A. Doroshenko, and N. D. Osgood, "Applying particle filtering in both aggregated and age-structured population compartmental models of pre-vaccination measles," *PLOS ONE*, vol. 13, p. e0206529, Nov 2018.

[23] C. Kwok, D. Fox, and M. Meil, "Real-time Particle Filters," *Proceedings of the IEEE*, vol. 92, no. 3, pp. 469–484, 2004.

[24] Y. Du, J. Liu, F. Liu, and L. Chen, "A real-time anomalies detection system based on streaming technology," in *2014 Sixth International Conference on Intelligent Human-Machine Systems and Cybernetics*, vol. 2, pp. 275–279, Aug 2014.

[25] S. Amini, I. Gerostathopoulos, and C. Prehofer, "Big data analytics architecture for real-time traffic control," in *5th IEEE International Conference on Models and Technologies for Intelligent Transportation Systems, MT-ITS 2017 - Proceedings*, pp. 710–715, Institute of Electrical and Electronics Engineers Inc., Aug 2017.

[26] E. Al Nuaimi, H. Al Neyadi, N. Mohamed, and J. Al-Jaroodi, "Applications of big data to smart cities," *Journal of Internet Services and Applications*, vol. 6, no. 1, pp. 1–15, 2015.

[27] M. Yu, C. Yang, and Y. Li, "Big data in natural disaster management: A review," *Geosciences*, vol. 8, no. 5, 2018.

[28] S. Ayvaz and M. Shiha, "A scalable streaming big data architecture for real-time sentiment analysis," in *Proceedings of the 2018 2nd International Conference on cloud and big data computing*, ICCBDC'18, pp. 47–51, ACM, 2018.

[29] A. Kejariwal, S. Kulkarni, and K. Ramasamy, "Real Time Analytics: Algorithms and Systems," *Proceedings of the VLDB Endowment*, vol. 8, pp. 2040–2041, Aug 2017.

[30] B. Ellis, *Real-time Analytics : Techniques to Analyze and Visualize Streaming Data*. Wiley, 2014.

[31] V. D. Ta, C. M. Liu, and G. W. Nkabinde, "Big data stream computing in healthcare real-time analytics," in *Proceedings of 2016 IEEE International Conference on Cloud Computing and Big Data Analysis, ICCCBDA 2016*, pp. 37–42, Institute of Electrical and Electronics Engineers Inc., Aug 2016.

[32] E. D. Valle, S. Ceri, F. van Harmelen, and D. Fensel, "It's a Streaming World! Reasoning upon Rapidly Changing Information," *IEEE Intelligent Systems*, vol. 24, pp. 83–89, Nov 2009.

[33] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "Stream: the stanford stream data manager (demonstration description)," in *Proceedings of the 2003 ACM SIGMOD international conference on management of data*, SIGMOD '03, pp. 665–665, ACM, 2003.

[34] S. Chandrasekaran and M. J. Franklin, "Streaming Queries over Streaming Data," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pp. 203–214, Elsevier, 2002.

[35] W. Raghupathi and V. Raghupathi, "Big data analytics in healthcare: promise and potential," *Health Information Science and Systems*, vol. 2, p. 3, Dec 2014.

[36] L. M. Lee and S. B. Thacker, "Evolving Challenges and Opportunities in Public Health Surveillance," in *Principles and Practice of Public Health Surveillance*, Oxford University Press, 3 ed., 2010.

[37] A. Belle, R. Thiagarajan, S. M. R. Soroushmehr, F. Navidi, D. A. Beard, and K. Najarian, "Big Data Analytics in Healthcare.," *BioMed research international*, vol. 2015, p. 370194, Jul 2015.

[38] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani, "Deep learning for iot big data and streaming analytics: A survey," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 2923–2960, 2018.

[39] M. M. Rathore, A. Ahmad, A. Paul, J. Wan, and D. Zhang, "Real-time Medical Emergency Response System: Exploiting IoT and Big Data for Public Health," *Journal of Medical Systems*, vol. 40, p. 283, Dec 2016.

[40] L. R. Nair, S. D. Shetty, and S. D. Shetty, "Applying spark based machine learning model on streaming big data for health status prediction," *Computers & Electrical Engineering*, vol. 65, pp. 393–399, Jan 2018.

[41] N. Mustafee, J. H. Powell, and A. Harper, "RH-RT: A data analytics framework for reducing wait time at emergency departments and centres for urgent care," in *Proceedings - Winter Simulation Conference*, vol. 2018-Decem, pp. 100–110, Institute of Electrical and Electronics Engineers Inc., Jan 2019.

[42] I. Celino, D. Dell'Aglio, E. Della Valle, Y. Huang, T. Lee, S.-H. Kim, and V. Tresp, "Towards BOTTARI: Using Stream Reasoning to Make Sense of Location-Based Micro-posts," pp. 80–87, Springer, Berlin, Heidelberg, 2012.

[43] M. J. Paul and M. Dredze, "You Are What You Tweet: Analyzing Twitter for Public Health," in *Fifth international AAAI conference on weblogs and social media*, 2011.

[44] A. Culotta, "Towards detecting influenza epidemics by analyzing Twitter messages," in *SOMA 2010 - Proceedings of the 1st Workshop on Social Media Analytics*, (New York, New York, USA), pp. 115–122, ACM Press, 2010.

[45] S. Munn, K. Y. Ni, and J. Xu, "Learning network dynamics from Tumblr®: A search for influential users," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10354 LNCS, pp. 204–213, Springer Verlag, 2017.

[46] B. M. Kuehn, "Twitter Streams Fuel Big Data Approaches to Health Forecasting," *Jama*, vol. 314, no. 19, pp. 2010–2012, 2015.

[47] A. Kumaresan, D. Liberona, and R. K. Gnanamurthy, "A Case Study on API-Centric Big Data Architecture," in *Knowledge Management in Organizations*, Communications in Computer and Information Science, pp. 459–469, Cham: Springer International Publishing, 2017.

[48] A. Haldorai and A. Ramu, *Cognitive social mining applications in data analytics and forensics.* Advances in social networking and online communities book series, 2019.

[49] R. Karim, R. Sahay, and D. Rebholz-Schuhmann, "A Scalable , Secure and Realtime Healthcare Analytics Framework with Apache Spark," no. November, pp. 2–3, 2015.

[50] D. J. Muscatello, T. Churches, J. Kaldor, W. Zheng, C. Chiu, P. Correll, and L. Jorm, "An automated, broad-based, near real-time public health surveillance system using presentations to hospital Emergency Departments in New South Wales, Australia," *BMC Public Health*, vol. 5, p. 141, Dec 2005.

[51] J. B. S. Ong, M. I. Chen, A. R. Cook, H. C. Lee, V. J. Lee, R. T. P. Lin, P. A. Tambyah, and L. G. Goh, "Real-time epidemic monitoring and forecasting of H1N1-2009 using influenza-like illness from general practice and family doctor clinics in singapore," *PLoS ONE*, vol. 5, p. e10036, Apr 2010.

[52] K. Lee, A. Agrawal, and A. Choudhary, "Real-time disease surveillance using Twitter data," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '13*, (New York, New York, USA), p. 1474, ACM Press, 2013.

[53] A. Doyle, G. Katz, K. Summers, C. Ackermann, I. Zavorin, Z. Lim, S. Muthiah, L. Zhao, C.-T. Lu, P. Butler, R. P. Khandpur, Y. Fayed, and N. Ramakrishnan, "The EMBERS architecture for streaming predictive analytics," in *2014 IEEE International Conference on Big Data (Big Data)*, pp. 11–13, IEEE, Oct 2014.

[54] M. Sofean and M. Smith, "A real-time architecture for detection of diseases using social networks: design, implementation and evaluation," in *Proceedings of the 23rd ACM conference on hypertext and social media*, HT '12, pp. 309–310, ACM, 2012.

[55] F. Jenhani, M. S. Gouider, and L. B. Said, "Streaming social media data analysis for events extraction and warehousing using hadoop and storm: Drug abuse case study," *Procedia Computer Science*, vol. 159, pp. 1459–1467, 2019.

[56] D. Zinn, Q. Hart, T. McPhillips, B. Ludascher, Y. Simmhan, M. Giakkoupis, and V. K. Prasanna, "Towards reliable, performant workflows for streaming-applications on cloud platforms," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 235–244, IEEE, 2011.

[57] N. Feldkamp, S. Bergmann, and S. Strassburger, "Online analysis of simulation data with stream-based data mining," in *SIGSIM-PADS 2017 - Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pp. 241–248, Association for Computing Machinery, Inc, May 2017.

[58] M. Deng, M. Nalin, M. Petković, I. Baroni, and A. Marco, "Towards Trustworthy Health Platform Cloud," in *Workshop on Secure Data Management*, pp. 162–175, Springer, 2012.

[59] J. Á. Bañares, "Model-Driven Development of Performance Sensitive Cloud Native Streaming Applications," in *PNSE@ Petri Nets*, pp. 13–26, 2017.

[60] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns : Fundamentals to Design, Build, and Manage Cloud Applications.* 2014.

[61] P. Rad, V. Lindberg, J. Prevost, W. Zhang, and M. Jamshidi, "ZeroVM: secure distributed processing for big data analytics," in *2014 World Automation Congress (WAC)*, pp. 1–6, IEEE, Aug 2014.

[62] N. Naik, "Docker container-based big data processing system in multiple clouds for everyone," in *2017 IEEE International Symposium on Systems Engineering, ISSE 2017 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., Oct 2017.

[63] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, *et al.*, "The FAIR Guiding Principles for scientific data management and stewardship," *Scientific Data*, vol. 3, no. 1, p. 160018, 2016.

[64] R. Madduri, K. Chard, M. D'arcy, S. C. Jung, A. Rodriguez, D. Sulakhe, E. Deutsch, C. Funk, B. Heavner, M. Richards, P. Shannon, G. Glusman, N. Price, C. Kesselman, and I. Foster, "Reproducible big data science: A case study in continuous FAIRness," *PLoS ONE*, vol. 14, p. e0213013, Apr 2019.

[65] T. Likhomanenkoa, A. Rogozhnikova, A. Baranova, E. Khairullina, and A. Ustyuzhanina, "Improving Reproducibility of Data Science Experiments," in *ICML 2015 AutoML Workshop*, 2015.

[66] F. Ohlhorst, *Big Data Analytics: Turning Big Data into Big Money.* John Wiley & Sons, 2013.

[67] A. Endo, E. van Leeuwen, and M. Baguelin, "Introduction to particle Markov-chain Monte Carlo for disease dynamics modellers," *Epidemics*, vol. 29, p. 100363, Dec 2019.

[68] L. Wang, *Bayesian Phylogenetic Inference via Monte Carlo Methods.* PhD thesis, University of British Columbia, 2012.

[69] A. Golightly and D. J. Wilkinson, "Bayesian parameter inference for stochastic biochemical network models using particle Markov chain Monte Carlo," *Interface Focus*, vol. 1, pp. 807–820, Dec 2011.

[70] I. Nevat, G. W. Peters, and J. Yuan, "Channel tracking in relay systems via particle MCMC," in *IEEE Vehicular Technology Conference*, 2011.

[71] M. Kattwinkel and P. Reichert, "Bayesian parameter inference for individual-based models using a Particle Markov Chain Monte Carlo method," *Environmental Modelling and Software*, vol. 87, pp. 110–119, Jan 2017.

[72] S. Wang, G. H. Huang, B. W. Baetz, and B. C. Ancell, "Towards robust quantification and reduction of uncertainty in hydrologic predictions: Integration of particle Markov chain Monte Carlo and factorial polynomial chaos expansion," *Journal of Hydrology*, vol. 548, pp. 484–497, May 2017.

[73] G. Mingas, L. Bottolo, and C.-S. Bouganis, "Particle mcmc algorithms and architectures for accelerating inference in state-space models," *International Journal of Approximate Reasoning*, vol. 83, pp. 413–433, 2017.

[74] D. Lovell, J. Malmaud, R. P. Adams, and V. K. Mansinghka, "ClusterCluster: Parallel Markov Chain Monte Carlo for Dirichlet Process Mixtures," Apr 2013.

[75] S. Henriksen, A. Wills, T. B. Schön, and B. Ninness, "Parallel implementation of particle mcmc methods on a gpu," vol. 45, pp. 1143–1148, 2012. 16th IFAC Symposium on System Identification.

[76] A. Lee, C. Yau, M. B. Giles, A. Doucet, and C. C. Holmes, "On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods," *Journal of computational and graphical statistics*, vol. 19, no. 4, pp. 769–789, 2010.

[77] G. Mingas and C.-S. Bouganis, "Population-Based MCMC on Multi-Core CPUs, GPUs and FPGAs," *IEEE Transactions on Computers*, vol. 65, pp. 1283–1296, Apr 2016.

[78] M. Quiroz, R. Kohn, M. Villani, and M.-N. Tran, "Speeding Up MCMC by Efficient Data Subsampling," *Journal of the American Statistical Association*, vol. 114, pp. 831–843, Apr 2019.

[79] A. E. Brockwell, "Parallel Markov Chain Monte Carlo simulation by pre-fetching," *Journal of Computational and Graphical Statistics*, vol. 15, pp. 246–261, Mar 2006.

[80] A. Jasra, D. A. Stephens, and C. C. Holmes, "On population-based simulation for static inference," *Statistics and Computing*, vol. 17, pp. 263–279, Aug 2007.

[81] K. B. Laskey and J. W. Myers, "Population Markov Chain Monte Carlo," *Machine Learning*, vol. 50, pp. 175–196, Jan 2003.

[82] S. Williamson, A. Dubey, and E. Xing, "Parallel Markov chain Monte Carlo for nonparametric mixture models," in *Proceedings of the 30th International Conference on Machine Learning* (S. Dasgupta and D. McAllester, eds.), vol. 28 of *Proceedings of Machine Learning Research*, (Atlanta, Georgia, USA), pp. 98–106, PMLR, 17–19 Jun 2013.

[83] D. Li and W. H. Wong, "Mini-batch Tempered MCMC," *arXiv: Computation*, 2017.

[84] H. Lu, Q. Shen, J. Chen, X. Wu, and X. Fu, "Parallel multiple-chain DRAM MCMC for large-scale geosteering inversion and uncertainty quantification," *Journal of Petroleum Science and Engineering*, vol. 174, pp. 189–200, 2019.

[85] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (Savannah, GA), pp. 265–283, USENIX Association, Nov. 2016.

[86] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," Mar 2016.

[87] R. Turner and B. Neal, "How well does your sampler really work?," *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, vol. 1, pp. 73–82, Dec 2017.

[88] S. P. Brooks and A. Gelman, "General methods for monitoring convergence of iterative simulations," *Journal of Computational and Graphical Statistics*, vol. 7, no. 4, pp. 434–455, 1998.

[89] A. Gelman and D. B. Rubin, "Inference from iterative simulation using multiple sequences," *Statistical Science*, vol. 7, no. 4, pp. 457–472, 1992.

[90] J. Yan, M. Cowles, S. Wang, and M. Armstrong, "Parallelizing mcmc for bayesian spatiotemporal geostatistical models," *Statistics and Computing*, vol. 17, no. 4, pp. 323–335, 2007.

[91] J. Song, S. Zhao, and S. Ermon, "A-NICE-MC: Adversarial Training for MCMC," *Advances in Neural Information Processing Systems*, vol. 2017-December, pp. 5141–5151, Jun 2017.

[92] R. T. Cox, "Probability, Frequency and Reasonable Expectation," *American Journal of Physics*, vol. 14, pp. 1–13, Jan 1946.

[93] M. G. Bruno, "Sequential Monte Carlo Methods for Nonlinear Discrete-Time Filtering," *Synthesis Lectures on Signal Processing*, vol. 6, pp. 1–99, Jan 2013.

[94] Z. Chen, R. Barbieri, and E. N. Brown, "State Space Modeling of Neural Spike Train and Behavioral Data," *Statistical Signal Processing for Neuroscience and Neurotechnology*, pp. 175–218, Jan 2010.

[95] N. Bergman, *Recursive Bayesian Estimation: Navigation and Tracking Applications.* PhD thesis, Linköping University, 1999.

[96] C. Andrieu, N. de Freitas, A. Doucet, and M. Jordan, "An introduction to mcmc for machine learning," *Machine Learning*, vol. 50, no. 1, pp. 5–43, 2003.

[97] A. S. Talawar and U. R. Aundhakar, "Parameter Estimation of SIR Epidemic Model using MCMC Methods," *Global Journal of Pure and Applied Mathematics*, vol. 12, no. 2, pp. 1299–1306, 2016.

[98] H. Nishiura, "Prediction of Pandemic Influenza," *European Journal of Epidemiology*, vol. 26, no. 7, pp. 583–584, 2011.

[99] P. Shao and Y. Shan, "Beware of asymptomatic transmission: Study on 2019-nCoV prevention and control measures based on extended SEIR model," *bioRxiv*, p. 2020.01.28.923169, Jan 2020.

[100] J. M. Lyneis, "System dynamics for market forecasting and structural analysis," *System Dynamics Review*, vol. 16, no. 1, pp. 3–25, 2000.

[101] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[102] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009. Accessed on: Mar. 2, 2021. [Online]. Available: `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf`.

[103] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[104] D. Abdurachmanov, P. Elmer, G. Eulisse, and S. Muzaffar, "Initial explorations of ARM processors for scientific computing," *Journal of Physics: Conference Series*, vol. 523, p. 012009, Jun 2014.

[105] D. Pellerin and S. Thibault, *Practical FPGA Programming in C.* USA: Prentice Hall Press, first ed., 2005.

[106] IEEE, "IEEE Recommended Practice for Software Requirements Specifications," *IEEE Std 830-1998*, pp. 1–40, 1998.

[107] Wiegers Karl, *Software Requirements, Second Edition.* 2003.

[108] M. Andress, P. Cox, and E. Tittel, *CIW security professional certification bible.* Hungry Minds, 2001.

[109] N. Poulton, *Docker Deep Dive: Zero to Docker in a Single Book.* Nigel Poulton, 2018.

[110] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017.

[111] D. P. Rodgers, "Improvements in multiprocessor system design," *SIGARCH Comput. Archit. News*, vol. 13, p. 225–231, June 1985.

[112] B. Gregg, "The flame graph," *Communications of the ACM*, vol. 59, pp. 48–57, may 2016.

[113] K. Kaczmarski and P. Rzażewski, "Thrust and cuda in data intensive algorithms," in *New Trends in Databases and Information Systems* (M. Pechenizkiy and M. Wojciechowski, eds.), (Berlin, Heidelberg), pp. 37–46, Springer Berlin Heidelberg, 2013.

[114] B. Tang, N. L. Bragazzi, Q. Li, S. Tang, Y. Xiao, and J. Wu, "An updated estimation of the risk of transmission of the novel coronavirus (2019-nCov)," *Infectious Disease Modelling*, vol. 5, pp. 248–255, Jan 2020.

[115] X. Li and N. D. Osgood, "Applying particle filtering in complex compartmental models of pre-vaccination pertussis," *bioRxiv*, p. 598490, apr 2019.

[116] G. O. Roberts, A. Gelman, and W. R. Gilks, "Weak convergence and optimal scaling of random walk Metropolis algorithms," *Annals of Applied Probability*, vol. 7, no. 1, pp. 110–120, 1997.

[117] J. M. Epstein, J. Parker, D. Cummings, and R. A. Hammond, "Coupled contagion dynamics of fear and disease: Mathematical and computational explorations," *PLoS ONE*, vol. 3, Dec 2008.

[118] E. Angelino, E. Kohler, A. Waterland, M. Seltzer, and R. P. Adams, "Accelerating MCMC via parallel predictive prefetching," in *Uncertainty in Artificial Intelligence - Proceedings of the 30th Conference, UAI 2014*, pp. 22–31, AUAI Press, 2014.

[119] S. Funk and A. A. King, "Choices and trade-offs in inference with infectious disease models," *Epidemics*, vol. 30, p. 100383, Mar 2020.

[120] N. Chopin, P. E. Jacob, and O. Papaspiliopoulos, "SMC2: An efficient algorithm for sequential analysis of state space models," *Journal of the Royal Statistical Society. Series B: Statistical Methodology*, vol. 75, pp. 397–426, Jun 2013.

[121] B. Calderhead, "A general construction for parallelizing MetropolisHastings algorithms," vol. 111, no. 49, pp. 17408–17413, 2014.