Message Passing Computational Methods with Pharmacometrics Applications

by

Balazs Nemeth

In partial fulfillment of the requirements for
The Doctoral Degree of Sciences

Computer Science

Promotor:	Prof. dr. Jori Liesenborgs
Co-Promotors:	dr. Tom Haber Prof. dr. Wim Lamotte

Universiteit Hasselt
August, 2020

Abstract

A pharmaceutical company needs to invest in the costly and tightly regulated multi-year drug development process early on. While many compounds are considered initially, only a few make it to the final phase where the newly developed drug is made available to the wide population. Time and effort is lost on the majority of candidate compounds as these turn out to not be efficacious and no return on investment is made. For this reason, pharmaceutical companies are interested in methodologies and approaches to better detect the ones that have more promise as soon as possible. In addition, the exclusivity granted by a patent is limited in time, and speeding up the process translates into material financial gains. Model Based Drug Development, a quantitative approach where gathered data is leveraged in an online fashion to improve decision making, has been suggested as one way to optimize the process. The domain of Pharmacometrics, a key component of Model Based Drug Development, is concerned with modeling human-compound interactions. One of the challenges in Pharmacometrics is that the computational requirements of the models preclude agility and timeliness. Currently, modelers switch between different projects to avoid stalls as typical computational runs can take up to weeks, but arguably this hampers swift modeling due to the long feedback cycle.

Recent computing systems have seen a surge in the number of explicitly exposed parallel resources due to the limits being reached in single processor systems. Leveraging the computational resources of these systems is far from trivial. Pharmacometrics, like other branches of computational science, is a multidisciplinary field. Scientists that specialize in the models are rarely equipped with the right Computer Science background to write efficient computational codes. Therefore, the common approach is to rely on software packages that provide a toolbox of mathematical and statistical methods. However, contemporary packages lack in efficiency when deployed on a parallel system. This motivates the need for new approaches like those explored in this thesis.

In the context of computational modeling, there are two prominent strategies for parallelization. First, computations on models are fit using an iterative optimization routine where multiple processors can be kept busy within each iteration by evaluating multiple candidate parameters concurrently. This part of the computation is referred to as the back-end. While this approach can hide the parallel constructs within the routine improving the usability of these routines for scientists from other domains, it requires the optimization routine to be designed to run in parallel. However, this might not always be feasible.

Second, in the front-end a single candidate parameter can be evaluated in parallel if permitted by the dependency structure of the model, a strategy suitable both for more sequential optimization routines as well as parallel optimization routines where it further improves performance. Even if a task can be decomposed into smaller concurrently executable tasks, doing so manually is tedious, error-prone and requires the right parallel computing background. Arguably, the scientist concerned with building these models is in an even worse position; their expertise is probably not in parallel computing and more automated approaches are preferable.

This thesis proposes novel ways to leverage parallelism in both the front-end and the back-end. In the former, two approaches are presented to parallelize evaluations without any input from the user. In the latter, changes to two existing state-of-the-art Markov Chain Monte Carlo samplers are presented that allow to better deal with large parallel systems in the message passing paradigm. Improvements for samplers running with data-bound models are explored as well.

One of the main properties of Pharmacometrics models is that computation time required for parts of the model depends highly on the choice of model parameters. For this reason, common approaches fail to perform well in this regard as they assume a more uniform execution time across evaluations. By neglecting this property, idle times are introduced resulting in poor use of available resources.

Performance gains observed by the presented techniques vary greatly from 10% all the way to many hundred fold reductions in execution time. It is important to note that these improvements depend not only on the targeted algorithms, but also on the computation models and on the platform. Nevertheless, the ideas are presented at a reasonably abstract level to support generalizations to other domains and computational models.

Acknowledgments

I have to start by thanking Tom Haber, one of the co-promoters of this dissertation, who has been an invaluable help both during discussions as well as contributing to the work presented in this PhD. Without his help, this thesis would not have been possible. Next, I wish to express my gratitude for all the input from Jori Liesenborgs, the promoter of this thesis. Even when I requested comments on my work close to deadlines, he was willing to find time to point out where clarifications were required. Third, I would like to thank Wim Lamotte, the second co-promoter, as it is due to his encouragement that I have started this PhD. It still surprises me how he can provide meaningful and accurate remarks while not being deeply involved with the research.

I am also indebted to Frank Van Reeth and Ingrid Konings for their unfailing assistance and Johnson & Johnson for funding the PhD research. It is due to these parties that I was able to travel around the world to present my work. I also need to mention the Flemish Supercomputer Center, known by its acronym VSC, and the Exascience Lab for access to their computational resources. Finally, hardware donated by Intel was of great help during early stages. These computational resources have made it possible to study how techniques behave on larger parallel systems.

Next, I am grateful to all the people who have helped in any shape or form, be it in providing valuable feedback during the last four years, or reading my work to improve the language.

Finally, I would like to thank all the peer reviewers for assessing the publications that were part of this PhD. Their feedback was valuable in improving the final published results. This includes the members of the jury who have reviewed this dissertation and have found time to attend the defense.

Contents

List of Figures

Glossary

API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
BSP	Bulk Synchronous Parallel
DAG	Directed Acyclic Graph
DES	Discrete Event Simulation
EA	Evolutionary Algorithm
ESS	Effective Sample Size
ETF	Earliest Task First
FCFS	First-Come-First-Serve
FDA	Food and Drug Administration
FEL	Future Event List
GPU	Graphics Processing Unit
HMC	Hybrid Monte Carlo
HPC	High-Performance Computing
ILP	Instruction Level Parallelism
IMH	Independent Metropolis-Hastings
LPT	Longest Processing Time
MBDD	Model-Based Drug Development

MC	Multiple Chain
MCMC	Markov Chain Monte Carlo
MH	Metropolis-Hastings
ML	Machine Learning
MLE	Maximum Likelihood Estimation
MP	Multiple Proposal
MPI	Message Passing Interface
ODE	Ordinary Differential Equation
OS	Operating System
PD	Pharmacodynamics
PDE	Partial Differential Equation
PK	Pharmacokinetics
PMX	Pharmacometrics
PRNG	Pseudorandom Number Generator
RR	Round-Robin
SAEM	Stochastic Approximation Expectation Maximization
SIR	Susceptible-Infected-Recovered
SMC	Sequential Monte Carlo
SMT	Simultaneous Multithreading
SSA	Static Single Assignment
SUNDIALS	SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers
WBC	White Blood Cell

Chapter 1

Introduction

1.1 Motivation

Drug development is a costly multi-phase process that spans multiple years. It is strictly regulated by government agencies like the Food and Drug Administration (FDA). A pharmaceutical company that develops a new drug needs to invest billions to adhere to regulations. Figure 1.1 provides a rough estimate of the timing and scale of this process. On average, the process takes around 10 years [Cio+14]. Initially, thousands of candidate compounds are considered most of which do not reach the final fourth phase [Hay+14]. A study in the years 2013-2015 noted that the majority of failures is attributed to the lack of efficacy [Har16] resulting in loss of time and money.

Even with a patent application filed early on in the drug development process, the protection offered by the patent starts to end at the beginning of the final phase when the return on investment is made. Some time extensions to the patent are possible to increase exclusivity up to 25 years, but coverage in the fourth phase remains limited. It is therefore of great financial importance that the process proceeds as swiftly as possible. Recently, Model-Based Drug Development (MBDD) was touted as a way to increase agility in this process [Mil+13]. The goal is to feed data gathered earlier in the development process to steer and adapt later phases. The faster the computations can finish on these models, the more agile the process becomes and the less time and effort is wasted.

Pharmacometrics (PMX) is a key component of MBDD. It is a field where mathematical and statistical methods are applied to pharmacology to develop an

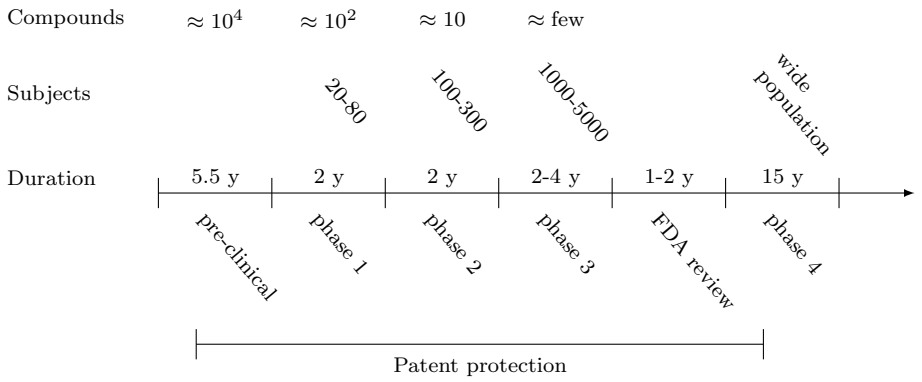


Figure 1.1: An overview of the drug development process that typically spans multiple years. Progressively less compounds make it to each subsequent phase while the number of subjects ramps up. More time spent in the early phases reduces the patent protection coverage in the fourth phase when the compound is made available to a wider population and the return on investment is made.

understanding of Pharmacokinetics (PK), the study of how an organism affects a compound, and Pharmacodynamics (PD) which focuses on the converse. In PMX, modelers often resort to pooling data from subjects into a single hierarchical model [OF14]. Finding parameters of these models is compute intensive; the PK and PD model that describe the interaction between each subject and a compound requires numerical integration. In addition, these models are often evaluated in a Bayesian context where samples from the posterior distribution are collected. Compared to the more classical approach of finding a single set of parameters, this drastically increases the number of model evaluations and hence the total computational requirements. This dissertation considers some of the computational aspects of PMX, required at different stages of the process given in Figure 1.1.

Famously, Moore's law predicted an exponential increase in transistor count in processors, but this increase is only loosely coupled with computational speed [Cha20]. The well-known memory wall [WM95; Dre07], i.e. the growing discrepancy between memory and processor speeds, together with the diminishing returns of Instruction Level Parallelism (ILP) [Gra$^+$03], and the power wall [EGC16] hinders this relationship. To keep up with the predicted perfor-

mance trend, recent computing systems have seen a surge in parallel resources as a means to avoid these limits. In contrast to parallelism implicitly exploited by a compiler like ILP, parallelism is explicitly exposed in systems moving the burden to the software designers or even the users.

1.2 Problem Statement and Research Goals

Efficient use of parallel resources is a daunting task. Without the right expertise, resource utilization tends to be low. In distributed memory systems, this is further aggravated by the need to keep not a few, but hundreds or thousands of processors busy to maintain high utilization. Scientists are rarely equipped with the right Computer Science background, a situation not only specific to PMX, but to other scientific computing domains as well. Therefore, the contemporary approach is to turn to software packages. Currently, the prominent tools in use for PMX are NONMEM [Erm+19] and more recently, Stan [Car+17] and nlmixr [Fid+19]. These implement powerful methods in the mathematical and statistical sense, but often lack in terms of parallel resource utilization and are therefore unsuited when results are either to be fed back into the drug development process in due time or rapid feedback in the modeling process is desired. Hence, novel approaches, where the details of large and complex parallel systems are abstracted away, are required.

This thesis assumes that scientific computing software, like software for PMX, can be structured into two components. The first component, referred to as the front-end, consists of the scientific model. In one interpretation, it is a standalone classical computer program that takes as input a set of parameters θ, and outputs a score that represents the quality of θ. This score is typically obtained by performing scientific simulations. The second component, referred to as the back-end, implements iterative methods [Mur10] and samplers. It takes as one of its inputs a scientific model and it either searches for good parameter choices or tries to assess the uncertainty in the parameters. Since the Bayesian approach is prominent in PMX, only samplers are considered here. While the focus is on PMX models, it is important to note that most of the explored approaches generalize to other scientific domains as well where observations are uncertain or incomplete requiring probabilistic techniques such as econometrics and astrophysics. For example, a model from epidemiology is used to evaluate performance improvements in Chapter 6.

The goal is to design methods leveraging parallelism in both components that perform well in a distributed setting without requiring any further input

from the scientists. This avoids distracting the scientists from their goals, while at the same time, maintaining high system utilization. The relatively high communication latency and the number of resources that need to be coordinated make this a challenging task. Two interesting properties of models in PMX are taken into account transparently where possible by the parallel methods. First, the amount of computation required for simulation varies with model parameters and second, the typical approach of scaling up work to reduce the overhead introduced by parallelism in relative terms is not viable since the models, including the data, are fixed.

Note that the computational aspect of PMX is targeted as a whole; the distinction between the front-end and back-end is mainly useful to structure the software. Depending on the specifics of the PMX models, the number of participants in the drug trail and the parallel system characteristics, some of the presented methodologies could result in more performance improvements than others. Throughout the drug development process, the models can change as more knowledge is gathered. For these reasons, methodologies do not target specific parts of the process. Instead, they are presented as standalone improvements instead while combining them into a single system that combines their strengths is itself a research challenge that is left as future work.

1.3 Contributions

Depending on the problem at hand, it might be better to allocate parallel resources in the front-end, in the back-end or in both. Therefore, the dissertation contributes methodologies for both components. Contribution **C1** considers PK and PD models specifically. Contributions **C2**-**C4** target the back-end and contributions **C5** and **C6** target the front-end.

C1 PK and PD dynamics need to be simulated for the length of the drug trails. When a drug is administered repeatedly, simulation time increases. The first contribution shows that by detecting the repetitive structure of the dynamics, these requirements are radically reduced while only sacrificing a small amount of accuracy. Methods to automatically determine the detection sensitivity to strike the best balance between accuracy and speed are included as well. While this technique does not directly target distributed systems, parallelism can still be used to correct for inaccuracies.

C2 Due to the amount of data to which models are fit, the front-end can become data-bound leading to low operational intensity. The second contribution

considers how to reduce wait-time by performing useful computations during otherwise stalled cycles. PMX models are mostly compute bound, but the presented work can also be extended to hide latency in a distributed setting.

C3 An important aspect that hinders usability of samplers is that convergence properties depend on the choice of proposal distribution. A typical setup is to draw proposals from a normal distribution. With this setup, a d dimensional problem requires tuning $\mathcal{O}(d^2)$ parameters to specify the shape of the proposal distribution, a daunting task without knowledge of the shape of the target distribution. This has led to the development of an affine invariant sampler that performs equally well regardless of any affine transformation of the target density, but this sampler performs poorly in a distributed system. Therefore, the third contribution shows that the dependencies in this sampler are more loose than might initially seem by exploiting the deterministic nature of Pseudorandom Number Generator (PRNG) streams. Building on this observation, a distributed version of the sampler is introduced where both the wait time and the number of messages exchanged are reduced.

C4 In PMX, the posterior density often has multiple modes due to non-linearity. The Sequential Monte Carlo (SMC) sampler can be configured to initially smooth out the posterior for easier exploration. It is a state of the art sampler that explores the target at multiple positions simultaneously allowing some parallelism while still preserving the density. However, with load imbalance, the collection of information after each step introduces wait-time. The fourth contribution is a speculative sampler that relaxes scalability limits caused by load imbalance. By observing that the decision taken based on the collected information is easy to predict well, computation can continue tentatively. If the tentative results turn out to be wrong, computation is rolled back to ensure that results are unaffected by speculation.

C5 One way to leverage parallelism in the front-end is to require scientists to specify the models with predefined parallel building blocks, but this limits expressiveness. The fifth contribution introduces a mapping between parts of the computation Directed Acyclic Graph (DAG) and the graphical model representation of a model. This mapping enables applying a parallelization of the graphical model to the computational tasks in the DAG. In double-blind placebo-controlled clinical trials where some subjects receive a placebo

while others receive the actual treatment, load imbalance is exacerbated further. Since the graphical model expresses the hierarchical structure of these models, this approach lends itself well to deal with differing simulation time of subjects. The goal is to deal with larger models prominent in later stages of drug development as each subject is modeled separately, while, at the same time, taking into account load imbalance.

C6 The downside of applying parallelism from graphical models to a task graph is that it might not be fine-grained enough. Therefore, the final contribution targets the front-end on a much finer level. By measuring the execution time of tasks in this model, a scheduling heuristic can be employed to reduce overall execution time. This however, leads to suboptimal performance as execution time changes. Therefore, by combining multiple schedules with an Evolutionary Algorithm (EA), a more robust schedule can be produced that performs well even as different model parameters are explored.

1.4 Dissertation Overview

This dissertation is organized in three parts. The first part provides preliminaries while the second and third parts consider back-end and front-end parallelization respectively. Each contribution from Section 1.3 has been published in one of six conference papers as the principal author. The references to these publications are included in Appendix B. Chapters 3 to 8 are based on these, although not in a verbatim manner. Some alterations have been made to support readability, to streamline notation and to avoid repetitive parts. Due to page limits, some results and figures had to be omitted from the publications. While these parts do not change the contributions of the papers, they have been included here as they provide a more complete exposition of the presented ideas. It is important to note that the parts that have been omitted from the publications have originally been peer-reviewed as well.

The first part starts with Chapter 2 introducing concepts and providing the necessary background including an introduction to hierarchical models and the Bayesian inference framework, both important when dealing with PMX models. This chapter ends with a straightforward parallelization of both a hierarchical model and a classical sampler that, at least in theory, would yield the correct results. However, the flaws are listed to motivate the work presented in the second and third part. The first part also includes Chapter 3 describing how to exploit the nearly periodic nature of repeated administration models by numerical

application of the method of averaging on the one hand and reusing previous computational effort on the other hand, resulting in an approximation of repeated administration models. Parallel computing is only of limited importance for this approximation, but it is nevertheless included to exemplify the models in use in PMX.

In the second part, Chapter 4 considers how to deal with data-bound front-ends, caused by the discrepancy between DRAM bandwidth and microprocessor speed that hinders reaching peak performance. As the proposed methodology requires factorization of the model, it cannot be considered to be confined to the back-end. However, it is included here as the main idea, improving operational intensity by performing useful computation during otherwise stalled cycles, relies mainly on the back-end. The concepts are demonstrated in the machine learning context with multi-threading, but it is applicable to a wide variety of parallel algorithms, and at different scales, including to PMX computations running on larger systems where a relatively high latency is observed for messages exchanged between remote processors. Second, Chapter 5 introduces a fully decentralized version of an affine invariant sampler. Two cases with differing communication-to-computation ratios are used during evaluation against the original master-slave solution, where a more than tenfold reduction in execution time is measured. Third, Chapter 6 introduces speculative computation into SMC samplers. Multiple test scenarios, each with different computational characteristics, are studied empirically on a compute cluster. Tests show that when decisions are predicted correctly, execution time is reduced drastically for use cases with high load imbalance. Furthermore, the maximum theoretical gain, derived from execution characteristics, is compared with the measured improvement to verify that most speculative evaluations are actually useful. If predictions are incorrect, or load is balanced, speculation has no measurable negative impact. Performance is also evaluated in a weak scaling setting on a cluster with 36 cores in each system.

The third part consists of Chapters 7 and 8. Chapter 7 proposes a methodology in which parallel parts are automatically identified by leveraging the conditional independence property in the graphical model extracted from the dataflow graph of a model specification. Finally, Chapter 8 studies how to find well-performing schedules by simulating their execution time and combining schedules produced by a heuristic through an evolutionary approach.

Chapter 9 provides conclusions and outlines some promising research ideas for future work. The dissertation ends with a Dutch summary in Appendix A.

Chapter 2

Preliminaries

As noted in Chapter 1, drug development is a costly multi-phase process that spans over multiple years. To maximize the bottom line, a pharmaceutical company involved with the development wants to maximize the time covered by the exclusivity protection of the patent in the final phase. For this, the computations need to be sped up as much as possible. To better understand the context of the computations that this dissertation focuses on, this chapter starts by introducing hierarchical models in Section 2.1 and Bayesian inference in Section 2.2. Next, Markov Chain Monte Carlo (MCMC) samplers and some general diagnostics are discussed in Section 2.3 to illustrate how Bayesian inference can be tackled in practice. After introducing terminology in Section 2.4, simple parallelizations for both hierarchical models and samplers are presented in Section 2.5 and Section 2.6 respectively and motivations are given why these are far from optimal. Next, the more general framework separating the front-end from the back-end is given in Section 2.7. Finally, the chapter concludes with Section 2.8.

2.1 Hierarchical Models

As data for PK and PD models consists of measurements taken from human subjects, e.g. the concentration of a compound in the body at a specific time requiring blood to be taken for each data point, only a limited number of measurements are available. By pooling data from multiple subjects or even multiple trials, the predictive power of the model increases. Here, PK and PD

models for each subject are combined into a larger hierarchical model. The top level describes the population as a whole, while lower layers add increasingly more details. In more classical models where all data is "independent and identically distributed", a single type of noise is modeled to account for the deviation of the fit to the data. In contrast, in hierarchical models, noise is separated out into different types by introducing more noise terms, each capturing noise at different places in the hierarchy. To accomplish this, the data needs to include an identifier that specifies to which patient each measurement belongs.

An example model from Pinheiro et al. [PB00] demonstrates why separating the data into experiment units is important. The example studies the time it takes for ultra-sonic waves to travel across rails. The data consists of measurements of the travel time across $M = 5$ different rails. It can be grouped into M sets, since for each measurement, the data also specifies from which rail the measurement was taken. The number of measurements for each rail n_i can vary across rails.

Consider the model that ignores the grouping given by Equation (2.1), where $\mathcal{N}(\mu, \sigma)$ is a normal distribution with mean μ and variance σ^2, β expresses the mean travel time and for each measurement the normally distributed error term $\epsilon_{i,j}$ captures the deviation from this mean. The goal for such a model would be to estimate the parameters β and σ. The concatenation of the parameters is referred to as θ throughout this dissertation. In this example, $\theta = [\beta, \sigma]$.

$$\begin{aligned} y_{i,j} &= \beta + \epsilon_{i,j}, \quad i = 1, \ldots, M, \quad j = 1, \ldots, n_i \\ \epsilon_{i,j} &\sim \mathcal{N}(0, \sigma) \end{aligned} \tag{2.1}$$

The model from Equation (2.1) is flawed since it assumes that all rails are identical. The difference between rails will be partly captured by β and partly by the error term. To better account for this difference, the model can be refined using the labels in the data by introducing another term b_i, as shown in Equation (2.2).

$$\begin{aligned} y_{i,j} &= \beta + b_i + \epsilon_{i,j}, \quad i = 1, \ldots, M, \quad j = 1, \ldots, n_i \\ b_i &\sim \mathcal{N}(0, \sigma_b) \\ \epsilon_{i,j} &\sim \mathcal{N}(0, \sigma) \end{aligned} \tag{2.2}$$

The term b_i expresses how rail i deviates from some mean rail. Note that the distribution of b_i also needs to be specified. The goal is now to also find σ_b. The remaining nuisance variables b_i can also be optimized. Note that the second

model could be rewritten into a classical linear model since the convolution of normally distributed random variables again gives a normally distributed random variable, but for more complex models like those in PMX, this is not always possible.

2.2 Model Parameter Distribution Estimation

A common method to find the parameters given some data is to maximize the likelihood $p(\mathcal{D}|\theta)$, where $\mathcal{D}$ refers to the data, i.e. $y_{i,j} \in \mathcal{D}$. The likelihood for the model from Equation (2.1) is given by Equation (2.3).

$$p(\mathcal{D}|\theta) = \prod_{i,j} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_{i,j}-\beta}{\sigma}\right)^2} \tag{2.3}$$

With Maximum Likelihood Estimation (MLE), the goal is to find the θ that maximizes the likelihood, denoted by θ_{MLE}. To improve numerical stability, the log of the likelihood is often maximized instead. This is especially important when $|\mathcal{D}|$ is large.

One key issue with this approach is that θ_{MLE} does not encode the shape of the likelihood, which is important to assess the quality of the result. For example, there might be other choices for θ that have the same likelihood value. One way around this is to compute the Hessian at θ_{MLE}, but this is only a crude approximation of the overall shape of the likelihood. It is especially important to note this for non-linear problems since such an approximation ignores the presence of many local maxima.

Another way to see this is to note that θ_{MLE} is a single vector that does not account for the uncertainty in the problem. In reality, there should be a way to differentiate between fitting the same model to more data even if the same θ_{MLE} maximizes the likelihood since the certainty of the estimates increases with more evidence. The Bayesian framework allows to account for the uncertainty in the problem. In addition, prior beliefs, i.e. knowledge from an expert or a previous experiment, can be encoded as well. The central idea in the Bayesian framework rests on Bayes' theorem applied in the context of modeling and data. Equation (2.4) provides the Bayesian framework. It shows how the posterior $p(\theta|\mathcal{D})$, the likelihood $p(\mathcal{D}|\theta)$ and the prior $p(\theta)$ relate. Note that the posterior is determined by the likelihood and the prior only up to a normalizing constant.

$$p(\theta|\mathcal{D}) \propto p(\mathcal{D}|\theta) \times p(\theta) \tag{2.4}$$

Starting from a prior belief of possible values for θ, the likelihood expresses how this prior belief should be altered when taking into account the data. This leads to the posterior which expresses how well each choice of θ explains the data.

The difficulty of Bayesian inference stems from the normalizing constant in Equation (2.4). Since $p(\theta|\mathcal{D})$ is a probability density in θ, the normalizing constant is given by Equation (2.5). Here, Ω denotes the part of the space where θ has support, i.e. $p(\theta|\mathcal{D}) \neq 0$. Note that the dimensionality of this integral depends on θ and the larger the model, the more demanding computation becomes.

$$\int_{\Omega} p(\mathcal{D}|\theta) \times p(\theta) d\theta \tag{2.5}$$

Nevertheless, due to the non-linear nature of the models used in PMX, modelers often resort to the Bayesian framework. Besides taking prior beliefs into account to quantify the uncertainty in the estimates, this approach is also more robust when models exhibit practical or structural identifiability [Rau$^+$09], an issue that commonly arises when data is either limited, noisy or both.

2.3 Markov Chain Monte Carlo Samplers

It might seem that the complete posterior could be obtained through discretization of the space and evaluation of the likelihood and the prior at each discretization point, but this quickly becomes prohibitively expensive due to the curse of dimensionality. While it is a feasible approach for the model from Equation (2.1) since θ is a two-dimensional vector, for higher dimensional problems like those of hierarchical models where the dimensionality of θ depends on the number of experimental units, this is not practical. For example, the dimensionality of $\theta = [\beta, \sigma_b, \sigma, b_{1:5}]$ in Equation (2.2) is eight. It is not uncommon to have on the order of thousands of parameters in PMX when thousands participate in the drug trial.

Many real world problems in the Bayesian setting can only be approximated with MCMC Samplers [Sol$^+$12], popularized by the fact that they suffer much less

procedure MH($\tilde{\pi}, q, \theta^1, m, \rho$) ▷ $\tilde{\pi}$ proportional to π
  for $t = 1, \ldots, m$ **do**
    $\tilde{\theta} \sim q(.|\theta^t)$
    if $u \sim \mathcal{U}(0,1) < \frac{\tilde{\pi}(\tilde{\theta})}{\tilde{\pi}(\theta^t)} \frac{q(\theta^t|\tilde{\theta})}{q(\tilde{\theta}|\theta^t)}$ **then**
      $\theta^{t+1} = \tilde{\theta}$
    else
      $\theta^{t+1} = \theta^t$
    end if
  end for
  return $\theta^{\rho m+1}, \ldots, \theta^m$ ▷ Discard $\theta^1, \ldots, \theta^{\rho m}$ samples as burn in
end procedure

Figure 2.1: The classical Metropolis Hastings algorithm. Starting at a position θ^1, this sampler takes m samples from a target density $\tilde{\pi} \propto \pi$ using a proposal distribution $q(.|\theta^t)$ centered around the current position θ^t. To avoid dependency on the starting position θ^1, a fraction ρ of the samples is discarded.

from the curse of dimensionality as only the part of the space that has as support is explored. For this reason, recent research has focused on developing a wide variety of samplers [Mur10; Mur$^+$16; Fen$^+$07; And$^+$03; For$^+$13]. Unfortunately, there is currently no panacea that will provide satisfactory results for all use-cases as each sampler has different statistical properties. It is therefore useful to have a complete toolbox of samplers [Hab$^+$18] available to choose from when faced with a given problem.

In their essence, all MCMC samplers perform a random walk, mathematically referred to as a Markov Chain. The transition kernel of the chain is constructed in a way that ensures that the stationary distribution of the Markov Chain obeys an arbitrary target π. As an introduction to MCMC samplers, consider the classic Metropolis-Hastings (MH) sampler [RC99; CC00] given in Figure 2.1. Like all samplers, an arbitrary target density π determined up to a constant, denoted by $\tilde{\pi}$, is given as input along with a desired number of samples. For the MH sampler, a proposal density q, a starting position θ^1 and a fraction ρ is also provided.

The sampler explores the target space by performing a random walk. For example, if a normal distribution with unit variance and mean θ^t is used as a proposal distribution $q(.|\theta^t)$, drawing from this normal will create a candidate position $\tilde{\theta}$ around the current position θ^t. This candidate is either accepted or

rejected with a probability given by the MH accept-reject ratio. This step is accomplished by drawing a uniform distributed random number between 0 and 1, denoted by $u \sim \mathcal{U}(0, 1)$, and comparing it with the accept-reject ratio. If the proposal is symmetric, the ratio simplifies to Metropolis ratio $\tilde{\pi}(\tilde{\theta})/\tilde{\pi}(\theta^t)$. Note that since the target density only appears in the ratio, it need not be normalized. For the remainder of this thesis $\tilde{\pi}$ and π are used interchangeably, but it is important to note that the normalization constant is typically not known.

While the samplers exhibit the Markov property, an important property of the MCMC samplers is that within the produced samples, the choice of the starting θ^1 still plays a limited, although important, role. The effect of the choice of the initial position diminishes gradually. Therefore, enough samples need to be taken and a fraction ρ needs to be discarded as *burn-in*.

The convergence rate of Monte Carlo methods is $\sqrt{n}$ where n is the number of independent samples. However, with MCMC, the convergence rate is slower due to dependency between the samples. One way to make the samples independent is by keeping every k^{th} sample, a process referred to as thinning. The higher the thinning factor k, the more independent the samples. This also provides intuition for approximating the real convergence rate of MCMC. Instead of expressing it in terms of n, the convergence rate depends on the effective number of samples n_{eff}, referred to as Effective Sample Size (ESS). In general, it is impossible to compute n_{eff} exactly for arbitrary problems, but approximations exist.

For Bayesian inference, $\pi(\theta)$ is set to $p(\theta|\mathcal{D})$ and samplers are collected to characterize the posterior. Note that $p(\theta|\mathcal{D})$ needs to be evaluated for each sample, a computationally demanding task in PMX as it requires simulation of PK and PD models of each subject.

For illustrative purposes, suppose that the MH sampler from Figure 2.1 is run on the target given in Equation (2.6) with $\theta = [x, y]$, based on Rosenbrock, a test function commonly used to assess the performance of optimization methods [JY13]. The resulting two dimensional samples with a tuned proposal distribution q are shown by Figure 2.2 with $\theta^1 = (0, 8)$ and $k = 100$. Assuming that enough samples are taken, the shape of $\pi(\theta_1, \theta_2)$ is characterized by the samples. Note how the first samples depend on the initial position θ^1. These samples are not invalid, but are an artifact caused by taking only a finite number of samples. A typical practical approach around this is to set $\rho = 50\%$.

$$\log \pi(x, y) = (1 - x)^2 + 100(y - x^2)^2 \tag{2.6}$$

While theoretically, with an infinite number of samples, the whole distribution is explored, note that in practice, a sampler can "get stuck" in a region of

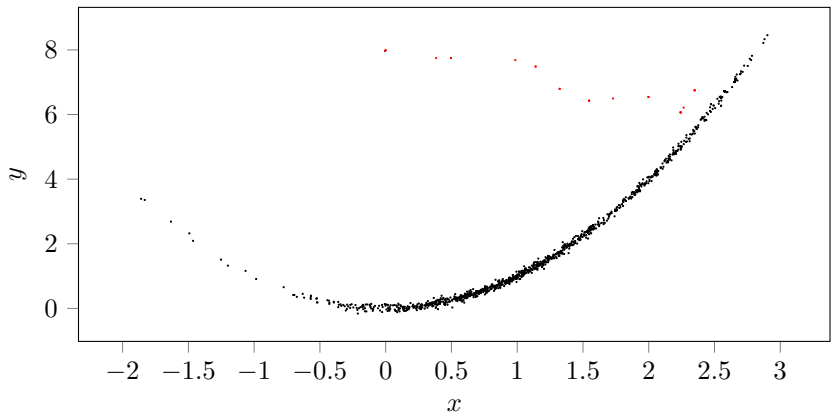


Figure 2.2: Samples from the Rosenbrock target density started at $\theta^1 = (0, 8)$. Note that the first samples, shown in red, depend on θ^1 and do not approximate the target density well.

high density. Consequently, the resulting samples do not describe the target distribution well and inference based on these samples will be inaccurate. This is an important issue to consider when the target density has multiple modes.

For that reason, it is advisable to run the sampler multiple times and to ensure that samples overlap. Another issue is that of a low ESS. It typically means that the chain did not take enough steps to properly "mix" and the resulting samples are not reliable. A diagnostic to assess proper mixing is a trace-plot constructed by plotting one dimension of the samples based on their index. As long as there is not enough mixing, another proposal distribution needs to be used, more samples need to be taken with a higher thinning factor, or other MCMC samplers with different convergence properties are required. The back-end parallelization part of this thesis focuses on parallelism in some of the more robust and prominent samplers in use at the time of writing.

The trace-plot for five different runs with progressively larger k is shown in Figure 2.3 for the first component of θ. With a d dimensional target π, a similar plot should be created for each of the d components. In the example shown, the total number of samples n increases with k to ensure that 500 samples remained after thinning, i.e. $n = 500k/\rho$. Note how the sampler "gets stuck" around the 250^{th} sample with k set to 50 or 100, but with k set to 500 or more, this problem is resolved. However, while increasing k results in more reliable samples,

it also directly increases computation time, hence motivating the need for more advanced samplers with different mixing properties.

2.4 Parallel Computing

In parallel computing, many independent processors are simultaneously employed to solve a single problem with the goal to reduce the time required to produce a result. The flip-side of this is that it puts more of the burden on the software developers, or even the users, as they need to determine how to best utilize resources. It is far from straightforward to leverage all the compute power available in parallel systems, and it requires rethinking algorithms with explicit parallelism in mind [EGC16; Gra+03].

Many models and implementations of parallel computing exists [Ski91], but this dissertation focuses mainly on the message passing model of parallel computing due to its generality, relative ease of reasoning, and practical applicability to large parallel systems. In this model, sequential processes communicate through first-in first-out queues. The ease of reasoning stems from the deterministic execution of each process given the reception order of its messages. Each sequential process is referred to as a *processor* throughout this dissertation. It is mostly encountered through implementations of the Message Passing Interface (MPI) standard [Gra+06]. From the perspective of the programmer, a set of processes are running on one or more multi-core systems and each is treated identically. These processes communicate through send/receive primitives or through collective operations built on top of these primitives. By default, hardware details like CPU topology [Bro+10] and interconnect technology are hidden from the programmer. If MPI is used in practice, some overhead is introduced for each message exchanged between processes due to (de)serialization.

Contemporary parallel systems are typically organized in a hierarchical fashion; some processors are "closer" to each other than to other processors. The larger system is comprised of smaller physical systems, that in turn contain multiple multi-core CPUs. Co-located processes typically have a link with lower delay and higher bandwidth, compared to links between processes that are further apart. Note that overhead is introduced even when running MPI processes on the same physical system where communication passes through shared memory, since at least one copy of the message contents is made.

In Linux, there is a subtle distinction between threads and processes [Ker10; BC05] while with MPI, the communicating entities are referred to as processes. Even though most implementations rely on MPI to evaluate the ideas proposed

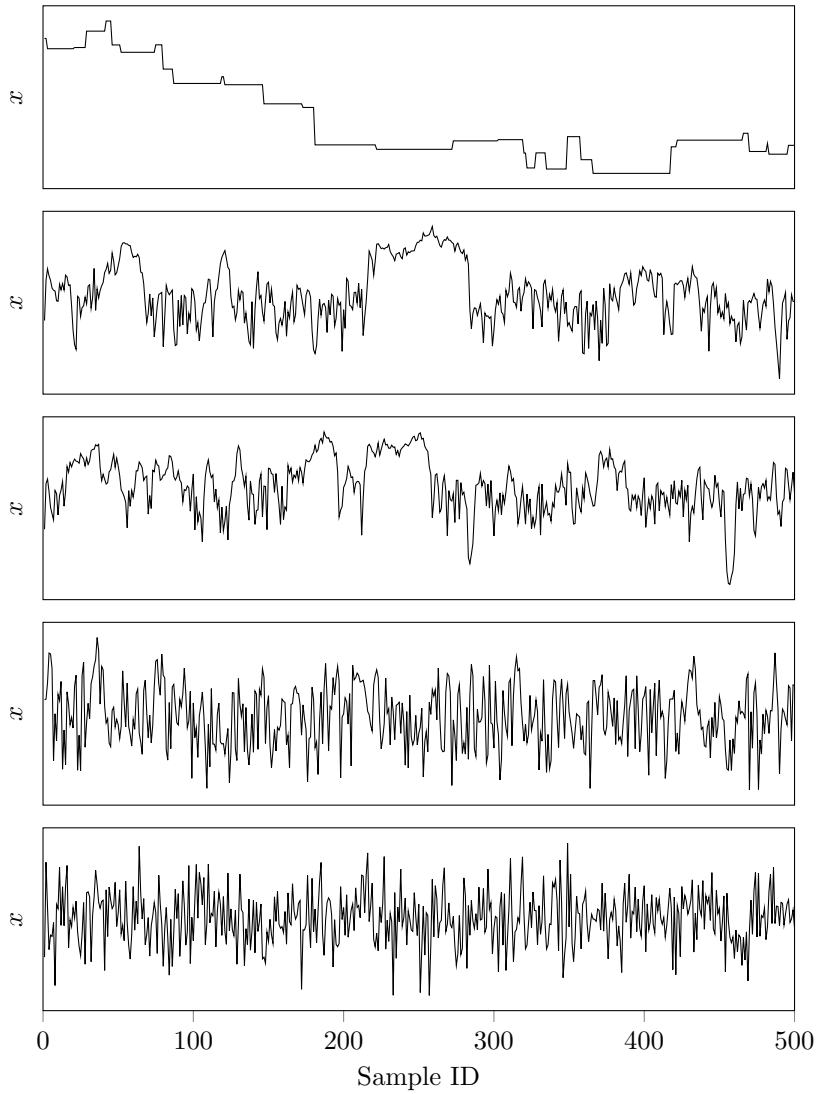


Figure 2.3: The first component x of the samples from the Rosenbrock target density with k set to $1, 50, 100, 500$ and 1000 from top to bottom, illustrating the difference between poor and good mixing.

in this dissertation, the term processor is used for the sake of generality and to signify that the same ideas are also applicable to threads, processes and other future technologies or a combination of these.

Productivity, not only in general but also in parallel computing, can be defined as the ratio of work to time. Increasing productivity is either accomplished by reducing time or increasing work. To study the quality of parallel approaches to problems, the amount of work needs to be quantified first. For this, the sequential execution time is taken as a baseline, denoted by T_s. The speedup with p processors on the same problem, denoted by S_p, is then given by the ratio of T_s/T_p. Intuitively, the best possible speedup is limited by p, although in some cases, speedup exceeds p referred to as super-linear speedup. As an example of this, suppose that computation is decomposed into a set of tasks and each task can cancel and abort overall computation. When these tasks are executed in parallel, the task that aborts computation could be executed earlier by one of the processors than when these tasks are executed sequentially. If such a task aborts computation, less work is performed and speedup will be above p. Nevertheless, it is common practice to consider the parallel efficiency E_p, given by S_p/p, to put performance into perspective with respect to this limit. While all the presented parallelizations in this thesis have an efficiency below 100%, it is important to note that the results should be considered in the context they are reported. For example, a parallel algorithm that reaches 60% efficiency with a hundred processors is still highly desirable as it will yield results 60 times faster than a sequential algorithm. In any case, it is important to understand what limits efficiency to avoid needlessly wasting computing power.

Note that S_p is given by the ratio of the sequential version of the algorithm to the parallel version with p processors. One common way to parallelize an algorithm is to insert constructs that assign work to processors and constructs to collect and combine the results from each processor. This typically adds overhead even if the parallel version is run with one processor. More complex parallelizations have a similar effect. This distinction between the sequential and parallel version results in a more optimistic speedup when the baseline is the parallel version of the algorithm. Regardless, whenever parallel speedup is reported in this thesis, the baseline is the running time of the parallel version with a single processor since there was little to no difference when the other baseline was used.

Amdahl's famous law [Amd67] gives an upper bound on the achievable speedup if the amount of work is fixed. More formally, if ρ denotes the fraction of the work that can only be executed on a single processor, with any amount of processors, execution time will at least be ρT_s.

Note that due to the inherent overhead introduced by communicating processors, reaching acceptable performance becomes harder. A message exchanged between processors needs to be prepared by the sending processor, placed on the communication link and read by the receiving processor. Each communication step adds additional computational requirements. In general, as more processors are employed, more messages are required to keep these processors busy. The overhead $T_{o,p}$, given by $pT_p - T_s$, needs to be considered with respect to the total amount of work required to solve the problem at hand, i.e. $T_{o,p}/T_s$. If there a relatively large amount of work, $T_{o,p}$ will be small and performance will not degrade much due to the overhead.

For these reasons, the common approach in High-Performance Computing (HPC) to circumvent limits imposed by overhead and the sequential fraction is to scale up work essentially reducing ρ and forcing $T_{o,p}/T_s$ to be as small as possible. However, in PMX this is only possible to a small degree. For example, the number of patients cannot be scaled up arbitrarily. Approaches, like those explored in this thesis, that have a smaller ρ or where $T_{o,p}/T_s$ is minimized by other means are required instead.

2.5 Parallelization with Multiple Chains

An embarrassingly parallel approach, applicable to any MCMC sampler, is to run multiple chains with differing PRNG seeds. This sampler is referred to as the Multiple Chain (MC) sampler. Suppose that n samples are to be taken without thinning. It turns out that by running p independent instantiations on p processors, only a limited speedup can be achieved.

To see this, consider the computational DAG of this approach shown in Figure 2.4. Edges depict information flow between tasks, shown by nodes. At the top, the t^{th} iteration of the first chains is shown in detail. Each iteration starts with the last sample $\theta^{t,1}$ and the target density evaluated at that sample $\pi(\theta^{t,1})$. Each iteration produces a new sample $\theta^{t+1,1}$ with the accompanying density $\pi(\theta^{t+1,1})$.

Figure 2.5 illustrates the problem of this approach with eight independent chains sampled in parallel on eight processors. The trace-plots show that with $\rho = 0.5$, a common setting in practice, already around 89% of the samples from each chain are discarded as burn-in, limiting speedup to approximately 1.7x. When more parallel chains are employed, speedup converges to 2x and efficiency drops towards 0%. In reality, speedup will be even lower since additional overhead will be introduced by setting up sampling and collecting samples at the end.

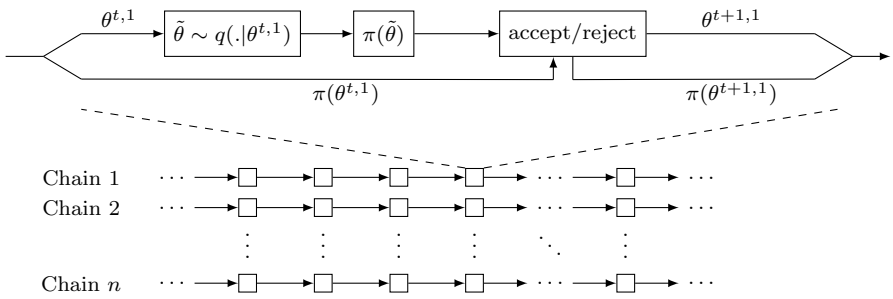


Figure 2.4: The dependency structure of the multiple chain parallelization. One iteration in the first chain has been detailed above. To avoid overloading the illustration, the inputs $\theta^{t,1}$ and $\tilde{\theta}$ to the accept/reject step have been omitted. Each chain can be evaluated in parallel without exchanging information.

In general, the total number of samples that needs to be taken per chain is $n\rho + n/p$. In the extreme, $p = n$ and one sample is collected from each chain. From this observation, it becomes clear that the burn-in can be seen as the sequential fraction in Amdahl's law. Subsequently, the speedup with p processors S_p wont exceed $1/\rho$.

To avoid this, samplers where information between chains is exchanged during a run are required to improve mixing speed under a given computational budget with parallel computing. Three such samplers, with information exchanged between chains during a run, are considered in the first part of the dissertation. Nevertheless, due to the stochastic nature of samplers, multiple runs are still advisable. It is outlined here for that reason and since it is still common in practice due to its simplicity [Car+17].

2.6 Parallel Hierarchical Models

PMX deals with models where the amount of available data is limited [PB00]. In contrast to more classical models where all data is "independent and identically distributed", it is known from which patient each measurement is taken. Hierarchical models exploit this information to improve predictive accuracy. The

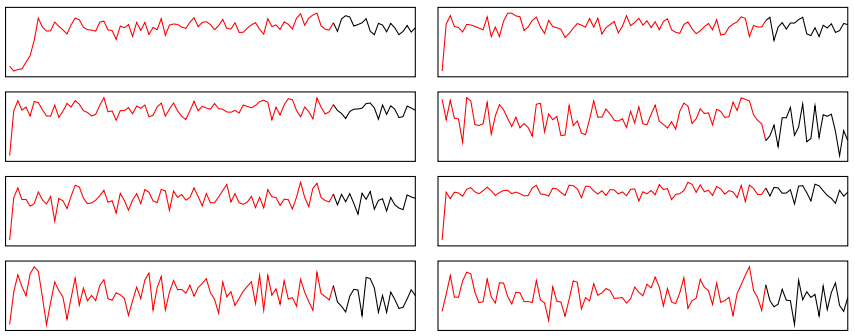


Figure 2.5: Samples taken from eight independent chains, run in parallel. Note that samples from each chain need to be discarded as burn-in. These are shown in red. Computational effort spent in the burn-in phase can be seen as the sequential fraction of execution time in Amdahl's law.

typical structure of these model is:

$$\begin{aligned} \phi_i &\sim \mathcal{N}(\mu, \Omega) \\ y_{ij} &\sim \mathcal{N}(s(x_{ij}, \phi_i), \sigma), \quad i = 1, \ldots, M, \quad j = 1, \ldots, n_i \end{aligned} \tag{2.7}$$

Here, $\theta = [\mu, \Omega, \sigma, \phi_{1:M}]$ and $(x_{ij}, y_{ij}) \in \mathcal{D}$. On top of the structural component s, describing the individual observations y_{ij}, the two statistical layers characterize the within-subject variability (within each individual profile) and the between-subject variability (on the individual parameters ϕ_i). The structural models, often expressed with Ordinary Differential Equations (ODEs), describe the biological process in terms of simple input-output equations. These systems can generally not be expressed in a closed-forms solution and computationally intensive time-stepping solvers need to be employed.

In hierarchical models, each level typically rely on the same models. For example, the PK and PD characteristics of subjects are modeled with the same equations and all subjects are placed in a single level. In Equation (2.7), these are captured by the structural component s. The difference is in the data to which each model needs to be fit.

As the model parameter θ is a concatenation of all the parameters from all the levels, it is possible to separate out only the required parameters for each subject and simulate the dynamics in parallel by assigning calls to s to different processors. This is essentially a parallelization over i. The downside is that this requires

altering the model to expose this parallelism either explicitly or by rewriting it with some predefined and rigid parallel building blocks. However, depending on the target parallel system, the models can be too small for parallelization. This is not only due to idiosyncrasies like a short duration of participation in drug trail, but also due to the simplicity of simulating some dynamics. Note also this approach might not exploit all the available parallelism. It might be beneficial to consider parallelizing over j or other computationally intensive transforms of random variables that are specific to a model. For that reason, more automated alternatives, like those explored in part two of this dissertation, are required.

2.7 A Computational Framework

Iterative methods and sampling algorithms, collectively referred to as optimization methods in this thesis, require repeated evaluation of a model to assess the quality of candidate parameters. As the complexity of scientific models increases, the time to evaluate a candidate solution for these models also becomes prohibitively long, and more evaluations are required to achieve acceptable solution accuracy. Therefore, it is no surprise that model evaluations encompass the majority of the computational time in optimization methods. Parallel computing can reduce the time to find a satisfactory solution both in the front-end and the back-end.

In the back-end, parallel versions of optimization routines [Pre+07] or sampling methods [Cal14; For+13; MDJ06; Ang+14] evaluate multiple candidate parameters concurrently and perform iterative updates based on their combined objective values. This approach follows the Bulk Synchronous Parallel (BSP) model of parallel computing [KK07]. The benefit is that the Application Programming Interface (API) of the optimization method parallelized by an expert need not change. Irrespective of what happens within the black box software library, the scientist need only provide a function to evaluate the model f at a set of parameters θ. The downside is that as systems with increasingly parallel resources become available, maintaining high resource utilization becomes difficult as the method itself constrains the amount of parallelization.

Figure 2.6 depicts the typical task graph of these optimization methods and samplers. For example, particle swarm optimization [Zeu+11] concurrently evaluates the model at multiple parameter choices, referred to as particles. The number of particles chosen limits the amount of parallelism, and using more particles does not necessarily improve convergence time.

Even if multiple evaluations of f can run in parallel, since each iteration

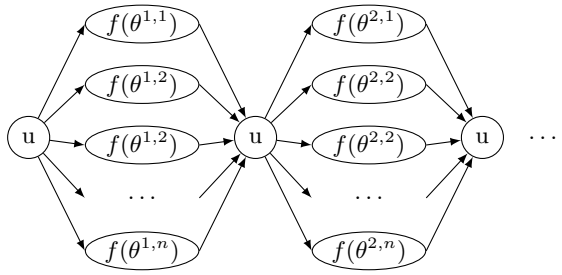


Figure 2.6: The general dependency structure of the first two iterations of parallel optimization methods depicted as a task graph. Nodes depict tasks, and edges depict inter-task information flow. Iteratively, n tasks evaluate a different candidate solution each on up to n processors in parallel. The label $f(\theta^{i,j})$ depicts the j^{th} candidate in the i^{th} step.

ends by collecting the results of these evaluations, depicted by the node with label u in Figure 2.6, some time will be lost unless all evaluations take the same amount of time. However, an important property of PMX models, not expressed by Amdahl's law, is that the computational time required to evaluate the model depends on the choice of θ. This is referred to as load imbalance and needs to be handled explicitly to avoid degraded parallel performance.

2.8 Conclusion

This chapter introduced the concepts required for the remainder of this dissertation. It has also demonstrated why naive parallelization will only yield limited results in samplers and why more advanced samplers are required. Discussion has also touched upon why explicitly parallelizing compute intensive PMX models is discouraged. A framework for computations has been introduced to show how the computational aspects of samplers and model evaluations interact. The notion of load imbalance, common in PMX models, was described in this framework. The next chapter will look at the PMX models in more detail.

Chapter 3

Approximate Repeated Administration Models

3.1 Introduction

One of the key questions of drug development, which PMX is concerned with, is what dosage regimen is safe and effective for individuals within a population. In this field, models from PK and PD characterize the interactions between a drug and an organism. Here, PK describes how a drug is affected by the organism, and PD describes the effect of the compound on the organism. The use of tools in this field requires both theoretical knowledge of biological systems and statistical expertise [OF14]. Therefore, methods that are easy to use, like the one described here, are of great interest.

Due to the complexity of these models, sufficient data is required to derive meaningful conclusions, but clinical data is typically sparse. Therefore, the common approach is to pool data from multiple drug trails and subjects within those trials. In this context, it is imprecise to merely consider the data as an unstructured collection of observations. Rather, with each observation, additional valuable information is available. This includes from which subject an observation is taken, his or her weight and height.

Mixed effect models incorporate this information in a statistically sound manner. Since PK and PD models typically rely on ODEs, simulation requires computationally intensive numerical methods. An integrator is configured to ensure some level of accuracy in the result. Depending on the ODEs, the size

of the steps that are taken is limited. More importantly, models with repeated administration hamper performance further. In these models, the simulation of dosing events causes the integrator to invalidate any gathered knowledge about the ODEs and take small steps. In addition, after a dosing event, computational time is spent on determining what step size to use.

Estimating parameters for these models in a reasonable amount of time requires not only the right mathematical tools, but also techniques from computer science. For example, within a drug trial, a compound is tested on multiple subjects and to determine the model parameter quality, each subject can be simulated in parallel. After parallelization, the most computationally intensive part is the numerical integration. Although parallel numerical integration has been studied [LMT01], only limited improvements are possible [Min10].

Instead, the method outlined in this chapter exploits the periodic behavior of models in PMX by reusing previous computations and employing the method of averaging to form an approximation of the model. It is applicable on top of any numerical integrator and besides a single parameter, no additional input from the user is required. To de-emphasize the existence of the parameter, it is important to note that it can be tuned automatically in a use-case dependent manner. Two examples are discussed to demonstrate this.

The remainder of this chapter is structured as follows. Section 3.2 lists related work. Two examples of repeated administration models are discussed in Section 3.3. The approximation method is presented in Section 3.4. Next, experimental results are shown in Section 3.5, and Section 3.6 concludes and provides directions for future work.

3.2 Related Work

Dunne et al. [Dun$^+$15] studied the application of the method of averaging in PMX, but their approach consisted of transforming the model by hand followed by solving it symbolically. The automated method presented in Section 3.4 partially relies on the same observations but differs in two ways. First, it does not require the user to manually alter the model. Second, for models that combine both PK and PD, all portions of the model are handled while the approach outlined by Dunne et al. focuses mainly on dealing with the PD portion where no periodicity is observed.

Conrad et al. [Con$^+$18] tackle computationally expensive models by constructing and gradually refining approximations of the posterior for Bayesian inference during MCMC sampling. Their approximation method uses previous

evaluations in a shrinking region to interpolate the posterior function. Similarly, Gong et al. [GD17] propose an adaptive refinement strategy that builds a surrogate model to explore a target distribution. Compared to these approaches where no knowledge of the underlying model is used, the approximation outlined below works at the level of the model itself. As such, the two approaches are complementary.

Rasmussen [Ras03] considers Hybrid Monte Carlo (HMC) on Bayesian integrals. In his work, gradients of the posterior are approximated using a Gaussian Process. He notes that to guarantee that the samples generated by HMC are unbiased, accurate posterior evaluations are only required at the end of a set of leapfrog iterations. Similarly, in Section 3.5, gradients are computed from the approximation and the final accept-reject step relies on the real model.

3.3 Repeated Administration Models

The two models in Sections 3.3.1 and 3.3.2 exemplify what is seen in drug development when patients are administered a compound periodically. While the details of the models are less important, they are listed here to describe their structure. Each model, denoted by f, consists of a set of ODEs parametrized by a vector ϕ. The set of q equations in f is denoted by $S = \{S_i(t)\}_1^q$.

Data to which these models are fit consists of a dosage regimen D and a sequence of observations (y_j, x_j). Each dosing event (a, c, t) in D adds some amount a of a compound to any state identified by c in model f at time t. Without loss of generality, the first dose is administered at $t = 0$, and all observations and dosing events are sorted by increasing time t. To fit ϕ, prediction $\hat{y}_j$ need only be made at x_j and Figure 3.1 outlines how to obtain predictions. It relies on a subroutine that implements an integrator of which the state is stored in $\mathcal{I}$.

The execution time of the integrator is mainly determined by the range spanned by x_j and the number of dosing events falling in that range since. Repeatedly stopping the integrator to simulate dosing events is the main cause for slowdown; as noted in Section 3.1, the integrator cannot take large steps when the internal state is changed. The method presented in Section 3.4 avoids this.

3.3.1 Nimotuzumab Model

The first model characterizes PK behavior of Nimotuzumab, a humanized monoclonal antibody mAb, in patients with advanced breast cancer [Rod$^+$15]. The

```
procedure IntegrateTo(x_1, ..., x_n, D, S)
    k = 1
    I = InitializeIntegrator(S)
    (a, c, t) = GetDose(D, k)
    for j = 1, ..., n do
        while t ≤ x_j do
            IntegrateTo(I, t)
            AddToState(I, c, a)
            k = k + 1;
            (a, c, t) = GetDose(D, k)
        end while
        IntegrateTo(I, x_j)
        ŷ_j = GetState(I)
    end for
    return ŷ_1, ..., ŷ_n
end procedure
```

Figure 3.1: Collecting predictions $\hat{y}_j$ using an integrator at time points x_j with a dosing regimen D. The ODE equations of the model are stored in S.

system of coupled differential equations in Equation (3.1) describes the dynamics of this model.

$$\begin{cases} \frac{\mathrm{d}C_{\mathrm{tot}}(t)}{\mathrm{d}t} &= -(k_{\mathrm{e}} + k_{\mathrm{pt}}) \cdot C(t) + k_{\mathrm{tp}} \cdot A_{\mathrm{t}}(t) - \left(\frac{k_{\mathrm{int}} \cdot R_{\mathrm{tot}} \cdot C(t)}{k_{\mathrm{ss}} + C(t)} \right) \\ \frac{\mathrm{d}A_{\mathrm{t}}(t)}{\mathrm{d}t} &= k_{\mathrm{pt}} \cdot C(t) \cdot v_1 - k_{\mathrm{tp}} \cdot A_{\mathrm{t}}(t) \\ \frac{\mathrm{d}R_{\mathrm{tot}}(t)}{\mathrm{d}t} &= k_{\mathrm{syn}} - k_{\mathrm{deg}} \cdot R_{\mathrm{tot}}(t) - \left(\frac{(k_{\mathrm{int}} - k_{\mathrm{deg}}) \cdot C(t) \cdot R_{\mathrm{tot}}(t)}{k_{\mathrm{ss}} + C(t)} \right) \\ C(t) &= 0.5 \cdot \Big[C_{\mathrm{tot}}(t) - R_{\mathrm{tot}}(t) - k_{\mathrm{ss}} \\ & \quad + \sqrt{(C_{\mathrm{tot}}(t) - R_{\mathrm{tot}}(t) - k_{\mathrm{ss}})^2 + 4 \cdot k_{\mathrm{ss}} \cdot C_{\mathrm{tot}}}(t) \Big] \end{cases} \tag{3.1}$$

Observations to which this model is fit consist of measured free concentrations of the mAb compound $C(t)$, at a particular time t, determined by the total mAb concentrations $C_{\mathrm{tot}}(t)$, the total target concentration $R_{\mathrm{tot}}(t)$ and the steady state rate constant k_{ss}. The change in the amount of free mAb in tissue compartments $A(t)$ depends on $C(t)$ and k_{pt} and k_{tp} which denote tissue-serum and serum-tissue rate constants respectively. The other constants that need to be estimated are

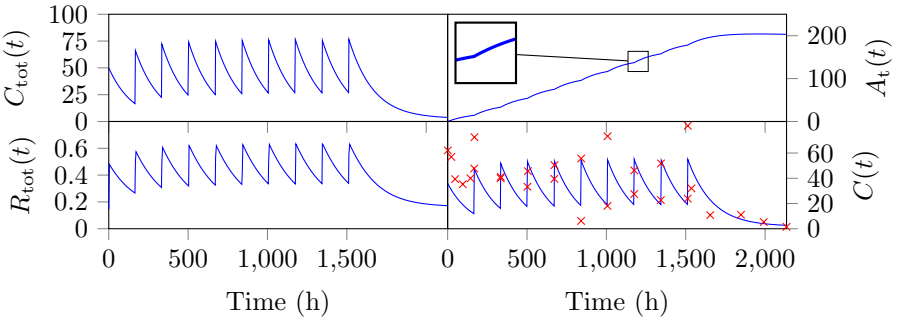


Figure 3.2: The ODE states from the Nimotuzumab three-compartment model with ten dosing events. The state for $C_{\text{tot}}(t)$, $A_{\text{t}}(t)$ and $R_{\text{tot}}(t)$ in function of time is shown on the left and top right, and the projected value $C(t)$ with observations shown as red crosses on the bottom right. After the first few dosing events, $C_{\text{tot}}(t)$ and $R_{\text{tot}}(t)$ exhibit close to periodic behavior. The plots were created by supplying a dense sequence of time points for x_j to Figure 3.1. The inset on $A_{\text{t}}(t)$ is discussed in Section 3.4.

the elimination rate k_{el}, the degradation rate k_{deg}, zero-order kinetic synthesis k_{syn} and irreversible internalization rate k_{int}. Note that there is a bidirectional influence between the compartments and $C(t)$ since it also appears on the right hand side. The model parameter vector ϕ is $[cl, v_1, Q, v_2, k_{\text{ss}}, k_{\text{int}}, k_{\text{syn}}, k_{\text{deg}}]$, where $k_{\text{e}} = cl/v_1$, $k_{\text{pt}} = Q/v_1$ and $k_{\text{tp}} = Q/v_2$.

Figure 3.2 shows an example of the evolution of ODE states in time for the Nimotuzumab model from Equation (3.1) with parameters $cl = 9.93 \times 10^{-4}$, $v_1 = 1.38$, $Q = 4.00 \times 10^{-3}$, $v_2 = 44$, $k_{\text{ss}} = 12.71$, $k_{\text{int}} = 3$, $k_{\text{syn}} = 1$ and $k_{\text{deg}} = 7$. There are ten dosing events, each adding 50 milliliters intravenously. Programmatically, this is done by adding the same amount to $C_{\text{tot}}(t)$ at each dosing event. During the first few dosing intervals, the concentration of the compound increases until the rate at which it is eliminated balances the rate at which the compound is added to the system. While $A_{\text{t}}(t)$ increases perpetually due to the bidirectional interplay between it and the compartments, *nearly* periodic behavior is observed in $C_{\text{tot}}(t)$ and $R_{\text{tot}}(t)$. Note that measurements are also taken after the final dosing event as $C(t)$ drops.

3.3.2 Canagliflozin Model

Canagliflozin is a drug for type-2 diabetes treatment. The model in Equation (3.2) for this drug consists of both a PK and a PD portion. The former is modeled by a two-compartment model [Hoe+15] denoted by the gut compartment $A_G(t)$, the central compartment $A_C(t)$ and the peripheral $A_P(t)$. Following Dunne et al. [Win+17], the latter is captured by glycated haemoglobin (HbA1c) denoted by $H(t)$.

$$\begin{cases} \frac{\mathrm{d}A_{\mathrm{G}}(t)}{\mathrm{d}t} & = -k_{\mathrm{a}} \cdot A_{\mathrm{G}}(t) \\ \frac{\mathrm{d}A_{\mathrm{C}}(t)}{\mathrm{d}t} & = k_{\mathrm{a}} \cdot A_{\mathrm{G}}(t) - k_{23} \cdot A_{\mathrm{C}}(t) + k_{32} \cdot A_{\mathrm{P}}(t) - k_{\mathrm{e}} \cdot A_{\mathrm{C}}(t) \\ \frac{\mathrm{d}A_{\mathrm{P}}(t)}{\mathrm{d}t} & = k_{23} \cdot A_{\mathrm{C}}(t) - k_{32} \cdot A_{\mathrm{P}}(t) \\ \frac{\mathrm{d}H(t)}{\mathrm{d}t} & = k_{\mathrm{in}} + Ef - k_{\mathrm{out}} \cdot H(t) \\ C(t) & = A_{\mathrm{C}}(t)/v \\ Ef & = (Ef_{\mathrm{c}} + Ef_{\mathrm{p}}) \frac{H(0)-5}{8-5} \\ Ef_{\mathrm{c}}(t) & = E_{\max} \frac{C(t)}{EC_{50}+C(t)} \end{cases} \tag{3.2}$$

For this model, $\phi = [k_{\mathrm{out}}, H(0), Ef_{\mathrm{p}}, EC_{50}, E_{\max}]$, where Ef_{p} represents the placebo effect, $k_{\mathrm{in}} = H(0) \cdot k_{\mathrm{out}}$, EC_{50} is the exposure that gives half-maximal effect and $E_{\max}$ is the maximal effect of the drug. The remaining parameters are fixed. A simulation with $k_{\mathrm{out}} = 10.24 \times 10^{-4}$, $H(0) = 7.72$, $Ef_{\mathrm{p}} = -0.482$, $EC_{50} = 60.34$ and $E_{\max} = -0.736$ is shown in Figure 3.3. The remaining parameters are $k_{\mathrm{a}} = 3.86$, $k_{23} = 0.101$, $k_{32} = 0.0928$, $k_{\mathrm{e}} = 0.174$ and $v = 92.2260$. Similarly to Nimotuzumab, periodic behavior is observed for the PK portion.

3.4 Approximating Models

In a model, states are classified either as periodic or non-periodic. Typically, the PK portion is periodic and the PD portion is non-periodic, but this need not be the case. In the integrated states, three phases are distinguished. The first phase spans over all dosing events for which the system has not yet entered periodicity. The second phase is the periodic phase typically taking up the majority of time in repeated dosing models as noted in Section 3.3. The start of this phase is detected based on a threshold τ that defines when a state is classified as periodic. The final phase starts at the last dosing event and ends at the last observation. In Figure 3.2, depending on τ, the second phase could start at 500 hours.

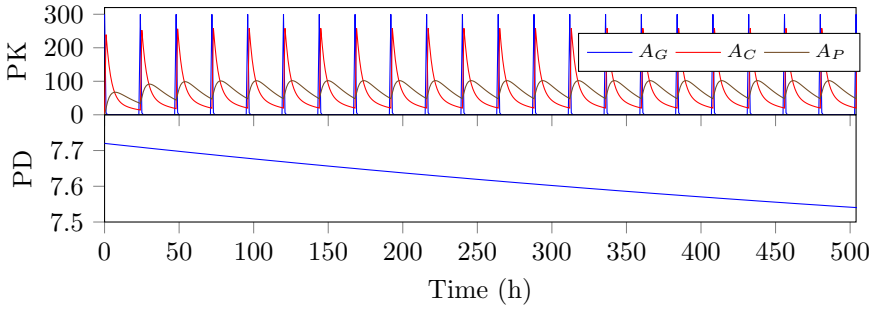


Figure 3.3: Canagliflozin PK/PD model for the first 21 dosing events. Periodic behavior is observed after a few dosing events for the PK portion of the model shown at the top. The PD portion, shown at the bottom, does not stabilize.

The goal is to avoid stopping and altering the state of the integrator to simulate dosing events since this increases execution time substantially. During the first interval of the second phase, all periodic states for the remaining observations are collected. The value of all non-periodic states is collected during the full length of the second phase by applying the method of averaging numerically.

In clinical trials, it is common to have dosage regimens where all dosing events add the same amount of a compound in the same way, i.e. $a_i = a_j$ and $c_i = c_j$ for any pair of dosing events i and j in Figure 3.1. However, it is possible to generalize the presented method where multiple runs of periodic behavior are observed. Since the models considered here only use dosage regimens with a fixed dosing amount, such extensions are left as future work. As will be shown in Section 3.5, the efficacy of the presented method depends on the time spent in periodic phases.

In reality, doses will never be spaced *exactly* uniformly throughout time. For example, one of the individuals in the Nimotuzumab data set with 10 dosing events, has the last dose administered at 1512.2 hours after the start of the trial. The average dosing interval is thus approximately 168.02, but the dosing intervals for this individual are between 167.33 and 170.07. In case varying intervals are captured by the model, noise is added complicating periodicity detection. Therefore, a preprocessing step ensures that the events are spaced equally at the cost of potentially introducing some error in the final approximation.

If the mean time between doses is $\Delta t = t_{|D|}/(|D| - 1)$, then the time for

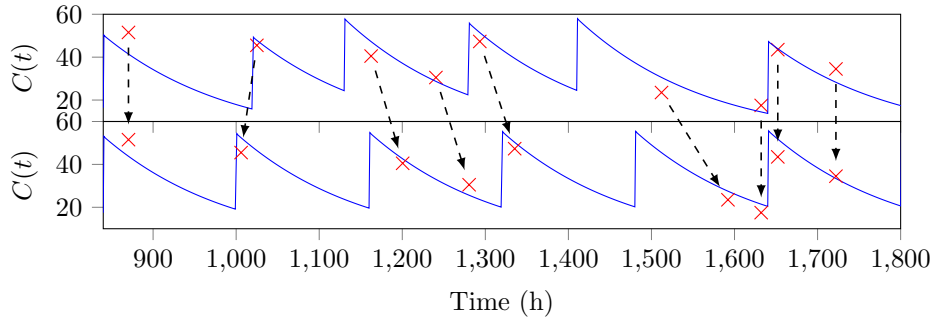


Figure 3.4: Dosing events are shifted to ensure that each dosing interval is the same. All observations, shown as red crosses, associated with each dosing interval are shifted accordingly. The 7^{th} observation is an example of an observation that, without capping, would be shifted to the next interval.

dosing event k is set to $t'_k = (k-1) \cdot \Delta t$. Next, each observation j is shifted according to the offset to the dosing event before it. Concisely, x_j is shifted to $x'_j = t'_k + z_j$, where z_j is computed as follows. If t_k denotes the time of the dosing event before it, then $z_j = \min(x_j - t_k, \Delta t - \epsilon)$. Here, capping the offset at $\Delta t - \epsilon$ ensures that the observation is not shifted to the next interval when it is close to the end since doing so introduces a large error due to the rapid rise in compound concentration after a dose. Figure 3.4 illustrates this process for an exaggerated example; as for the Nimotuzumab example shown above, the variance in dosing intervals for real use cases is typically much smaller. For models in which the dosing intervals are fixed, like for the Canagliflozin model, data need not be preprocessed.

After preprocessing, integration can start. For any model f, three different sets of equations S, S' and $\tilde{S}$ are used. Here, S is the original unaltered set of equations used during the first and third phase. During the first interval of the second phase, S' is used and the method of averaging is applied numerically during the remaining intervals in the second phase using $\tilde{S}$. The details of these sets of equations will be introduced next.

Integration commences on the set of equations $S = \{S_i(t)\}_1^q$ in f. At each dosing event k, all states in S are partitioned into r periodic states $P = \{P_i(t)\}_1^r$ and $q - r$ non-periodic states $N = \{N_i(t)\}_{r+1}^q$ by using some threshold τ and the criteria $|(S_i(t'_k) - S_i(t'_{k-1}))/S_i(t'_k)| < \tau$. If $|P| > 0$, the state of the integrator

$\mathcal{I}_{\text{real}}$ is copied to $\mathcal{I}_{\text{approx}}$. At this time, denoted by t_α below, the second phase is entered and integration continues using $\mathcal{I}_{\text{approx}}$.

During the first interval of the second phase, integration continues with S', a set of equations constructed by adding the equations $\mathrm{d}P_i'(t)/\mathrm{d}t = P_i(t)$ to those in S for a total of $2|P| + |N|$ equations. The value of $P_i'(t_\alpha)$ is set to 0. These additional equations will be used to compute the average of the periodic states for use in the remaining intervals of the second phase. After one dosing interval, integration continues using $\tilde{S}$, constructed by taking the equations $\tilde{P} = \{\mathrm{d}\tilde{P}_i(t)/\mathrm{d}t = 0\}_1^r$ together with the states in N. The initial value for the states in $\tilde{P}$ is $P_i'(t_\alpha + \Delta t)/\Delta t$. In other words, the states in P are replaced by a constant equal to the mean value during a dosing interval. This is how the method of averaging is applied numerically. The values of the states in N are then collected during the second phase at each x_j'. Finally, at the last dose, integration continues using S restoring the states in P to those saved in $\mathcal{I}_{\text{real}}$. The top left of Figure 3.5 demonstrates when each of these sets is used.

The states of P during the second phase are collected at times $t_\alpha + z_j$ for all observations j for which $x_j' > t_\alpha$. Note that if integration can only continue forward in time, all z_j need to be sorted. This can be seen as moving observations to the first interval of the second phase. Figure 3.5 shows the output for the Nimotuzumab model from Figure 3.2. Note that except for a different value of the integrated states, preprocessing and shifting of observations and events is not reflected in the output.

Let $c(t_0, t_1, S)$ denote the computational cost of using an integrator between time t_0 and t_1 on a set of equations S. The total cost of integration can be broken down into $c(0, t_\alpha, S)$, $c(t_\alpha, t_{\alpha+\Delta t}, S')$, $c(t_{\alpha+\Delta t}, t_{|D|}, \tilde{S})$ and $c(t_{|D|}, t_{n_i}, S)$. Since doses need not be simulated in $\tilde{S}$, $c(t_{\alpha+\Delta t}, t_{|D|}, \tilde{S}) \ll c(t_{\alpha+\Delta t}, t_{|D|}, S)$. Some overhead is introduced by preprocessing the data and using S' for one interval, but this is typically much smaller than the reduction in execution time obtained by avoiding simulation of doses between $t_{\alpha+\Delta t}$ and $t_{|D|}$.

Note that states in P are distinguished from those in N by τ. If τ is set too low, all states remain non-periodic and there is no second and third phase. In this case, no cost reduction will be made while some error will still be introduced by the preprocessing step. On the other hand, if all states are marked as periodic, then $c(t_\alpha + \Delta t, t_{|D|}, \tilde{S}) = 0$ since it can be skipped completely and larger cost reductions are expected. Note also that if all measurements after the last dose fall within a span of Δt, integration does not need to switch back to S from $\tilde{S}$.

A useful aspect of the outlined approach is that S' and $\tilde{S}$ can be constructed from S without symbolic manipulation. Integrator implementations require

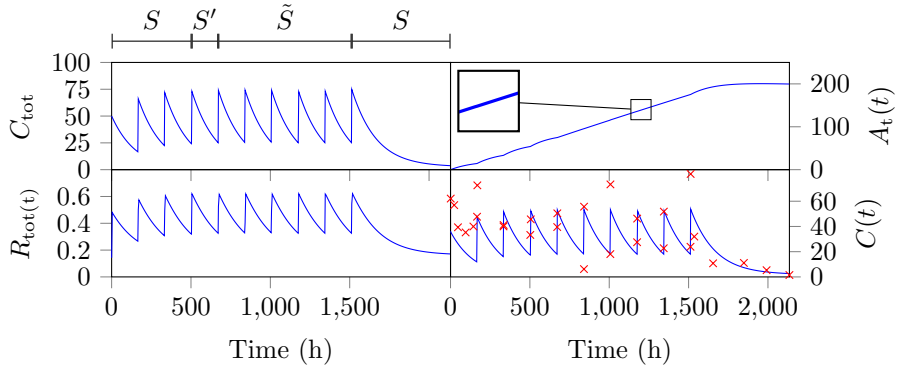


Figure 3.5: Approximation of the Nimotuzumab three-compartment model with ten dosing events. Different sets of equations are used at different times. The sets are $S = P \cup N$, $S' = P' \cup N$ and $\tilde{S} = \tilde{P} \cup N$. These are only shown in the top left, but the change in equations effects all states. The choice for τ defines the phases. Here, the first phase spans $[0, 504]$, the second phase spans $[504, 1512.2]$ of which the first interval is $[504, 672]$ and the third phase starts at 1512.2. Compare all results with Figure 3.2 and note how $A_t(t)$ is smoothed out due to applying the method of averaging numerically. However, preprocessing and event shifting happens transparently. The effect of approximation on the other states is barely visible.

the user to provide a function that, given $S_i(t)$, returns a vector of which the i^{th} component represents $\mathrm{d}S_i(t)/\mathrm{d}t$. Multiplying this vector with the bit vector where all the components corresponding to states in N are set to 1 is a straightforward way to transform S into $\tilde{S}$.

3.5 Performance Evaluation

Test data is taken from an online resource [Tra18] for the Nimotuzumab model and is generated synthetically for the Canagliflozin model using the parameter estimates from Dunne et al.[Win+17]. The Stochastic Approximation Expectation Maximization (SAEM) algorithm from Kuhn et al. [KL04] is used to fit a complete hierarchical model, described in Section 7.3. It is difficult to obtain a clear understanding of how well the presented approximation performs by comparing SAEM directly. Instead, the SAEM algorithm is run on the real model and the parameters at which the likelihood is evaluated are logged. The CVODE solver from the SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers (SUNDIALS) software package [Hin+05] is used as the integrator implementation.

The evaluation time together with the log-likelihood value of the classical approach from Figure 3.1 is measured for the collected parameters. The same is measured for the approximate model with different choices for τ. Figure 3.6 illustrate the influence of τ on both the relative error of the log-likelihood and the speedup between the real and the approximate model. For $\tau = 0$, no speedup is expected since no states will be classified as periodic. Since doses are shifted for the Nimotuzumab model, some error is still introduced. This is not the case for the Canagliflozin model as it does not take into account varying dosing intervals. In both models, the slowdown with $\tau = 0$ is due to computing and sorting z_j, and the additional bookkeeping that is needed to compare the value of each state with τ. Note the difference in speedup between the two models. The Canagliflozin data contains individuals with a much larger number of dosing events than those in the data for the Nimotuzumab model.

Next, data is generated synthetically with an increasing number of doses to show that the total time spent by the integrator in the second phase determines the improvements that can be obtained by using the approximate model. In Figure 3.7, τ increases from 0 to 0.008, showing that with more dosing events, and hence more periodic behavior, a larger increase in performance is observed.

Recall from Section 7.3 that $\phi_i = \mu + \eta_i$. In algorithms like SAEM, one of the steps involves integrating out random effects η_i for a given individual. Due to

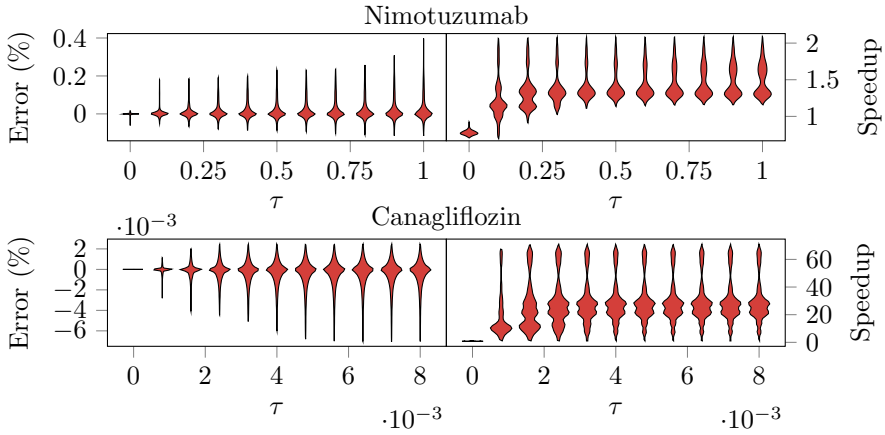


Figure 3.6: Violin plots showing relative error and speedup as the threshold τ increases for the Nimotuzumab model at the top and for the Canagliflozin model at the bottom. A larger τ increases the probability of introducing a larger error. At the same time, a higher speedup factor is obtained. While both models show the same behavior as τ increases, there is a difference in scale of the error and τ due to a different number of dosing events in the data and structural differences between the models.

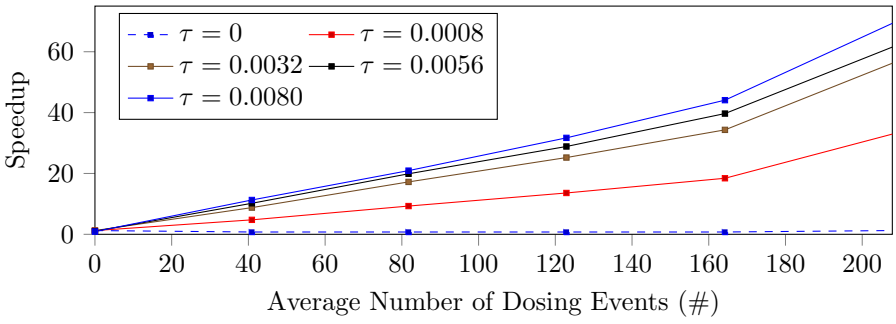


Figure 3.7: Speedup for varying τ and varying number of observations for the Canagliflozin model. With more observations, the second phase makes up a larger fraction of the total execution time. Hence, there is a more opportunity to reduce execution time. Although not clearly visible, with $\tau = 0$, a slowdown of up to 25% is seen.

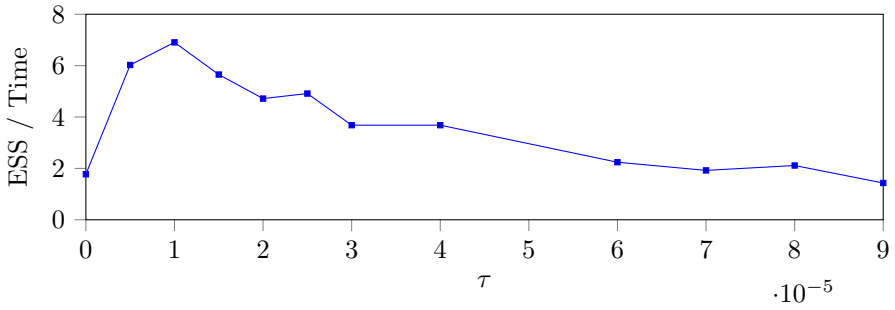


Figure 3.8: ESS per unit time while varying τ for the Canagliflozin model. This metric can be used to tune τ automatically.

the complexity of the models, MCMC samplers are used. Using the approximate model directly in this step results in biased estimates as the introduced errors change the distribution of random effects. As shown above, through the choice of τ, accuracy is sacrificed for performance. Two ways are discussed to use the approximation without introducing bias. A function that weights both the accuracy and the performance aspects is given for each. The same function can then be used to tune τ automatically. While tuning brings with it some computational costs, estimating parameters of hierarchical models takes orders of magnitude longer so it is worth spending some time on the tuning process. The objective is to find a sufficiently good value for τ and not necessarily the optimum. Therefore, tuning can be done on a subset of individuals.

One way to use the approximation is with HMC. Here, new positions are proposed by following the gradient L times and performing an accept-reject step at the final position. If gradients are computed from the approximate model and the accept-reject relies on the real model, the samples obtained remain unbiased [Ras03]. Note that in scenarios where L is large, larger reductions in execution time are possible. Since the gradients are only approximate, proposals will be of lower quality. For example, if the real and the approximate gradients differ too much, the proposed positions will have low mass and many points will be rejected. In turn, this lowers the ESS metric used to evaluate the information content of dependent samples. Tuning τ is accomplished by maximizing ESS per unit time. Figure 3.8 shows this metric for Canagliflozin using $L = 4$ while varying τ. Clearly, the optimal value for τ depends on the choice of L.

As noted above, generating samples directly with any MCMC sampler from

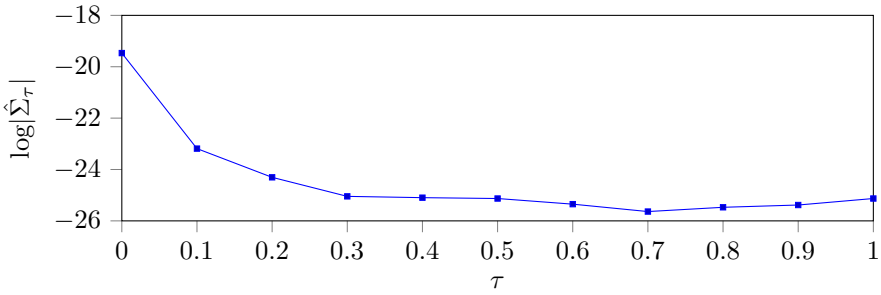


Figure 3.9: The value of $\log|\hat{\Sigma}_\tau|$ in function of τ for the importance sampling estimator. By setting τ to 0.7, an appropriate trade-off between approximation accuracy and computational cost is made.

the random effects distribution built with the approximate model will introduce bias due to the errors. Another way to use the approximation is through importance sampling, where bias is corrected by weighting each sample [Mac03]. These weights, obtained by taking the ratio between the density of the real and the approximate model, can be computed in parallel. If there is too much difference between the importance distribution and the target distribution, expectations computed from samples will exhibit more variance, denoted by σ_τ. An estimator $\hat{\sigma}_\tau$ is built by repeated sampling. A value for τ that trades off between computational efficiency and quality is chosen by minimizing $\hat{\sigma}_\tau$ while keeping time fixed. With multiple random effects, the covariance estimator $\hat{\Sigma}_\tau$ is used instead. Figure 3.9 shows this for the Nimotuzumab example. In this case, τ is tuned by minimizing $|\hat{\Sigma}_\tau|$, the determinant of the covariance matrix.

3.6 Conclusion and Future Work

This chapter introduces an approximation of repeated administration models that exploits past computation efforts and employs the method of averaging numerically. In case of models with varying dosing intervals, a preprocessing step allows for detection of periodic behavior at the cost of adding some error to the approximation. The actual improvements vary depending on the model and the parameters of the model. On one of the test models, up to 70-fold reductions in run-time were measured while introducing only on the order of 10^{-3} relative error. Since fitting a hierarchical model can take up to hours or

even days depending on the configuration parameters of algorithms like SAEM, these improvements have a tremendous impact on the end-users.

The approximation relies on setting the threshold τ to detect repetitive behavior in ODE states. It determines both the error and speedup of using the approximation instead of the real model. Incorporating a self-adjusting mechanism to automatically set τ for an MCMC sampler was discussed. Different objective functions can be devised depending on the use-case to tune τ, some of which will be studied in future work.

Speculative parallelism is a method to parallelize sequentially dependent tasks [Gra+03]. It has previously been applied to the classic Metropolis-Hastings MCMC sampler [Ang+14] where the sequence of accept-reject choices are guessed to predict the chain positions. Verification of these predictions then proceeds in parallel. A benefit of the speculative approach is that the collected samples are unaffected. Similarly, the approximation method presented here can be applied to predict the chain, after which verification can occur in parallel. As in Section 3.5, it is again possible to tune τ. Here, τ trades off between the prediction accuracy and the time spent creating the prediction.

The choice of τ does not bound the error in the approximation. Tolerance bounds are typically already provided as parameters for numerical integration methods. Therefore, a promising direction of future work is to consider the change in integration results by entering the second phase one interval later.

Part I

Back-end parallelization

Chapter 4

Improving Operational Intensity

4.1 Introduction

It has been a long established fact that the memory subsystem in contemporary systems plays a crucial role in determining performance. It was predicted that performance of all algorithms would be dominated by the memory subsystem [WM95], but this is only so for algorithms that neglect these system specifics. The memory hierarchy of modern processors, for which bandwidth at different levels is shown by the left of Figure 4.1, places particular importance on data locality.

At the same time, the number of cores per processor has increased circumventing physical limits of driving up frequency and keeping costs low [EGC16]. This has led to development of parallel algorithms to leverage available compute capacity.

While modern systems employ multi-channel memory architectures to increase memory bandwidth available to the processor [Dre07], it still determines performance for computations with low operational intensity. The situation is worsened in multi-core systems as each core in the processor competes for the same memory controller. Per core memory bandwidth diminishes, and computation stalls more frequently. In addition, multiple cores are needed to saturate memory bandwidth. Figure 4.2 illustrates this with data collected using a tool based on lmbench [MS96]. Note that when two cores run the benchmark

Memory type	Size	Bandwidth
CPU registers	few	1 TB/s
CPU caches	KBs	100 GB/s
Main memory	GBs	10 GB/s
Disk storage	TBs	100 MB/s

Figure 4.1: Approximate performance at each memory hierarchy level in a modern system.

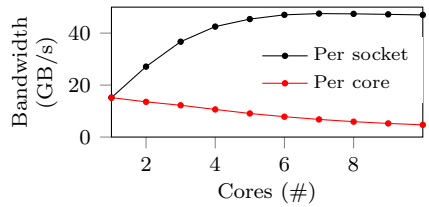


Figure 4.2: Memory bandwidth for increasing number of cores on a single processor.

in parallel, the per core memory bandwidth drops only slightly while if one core could saturate the available bandwidth, a larger drop would be expected.

Operational intensity [WWP09] is a feature of a computation kernel that measures the number of operations per unit of data. If this ratio is low, only a few cycles are spent processing each data element placing more pressure on the memory subsystem. If this subsystem becomes saturated, processing stalls. This is especially relevant for many machine learning applications such as regression, support vector machines and neural networks, as the training data set needs to stream through the memory hierarchy multiple times to compute the cost associated with the a candidate parameter θ. With Big Data, the hierarchy depth increases since data is loaded from disk or networked storage.

Modern super-scalar processors implement various techniques including prefetching, Simultaneous Multithreading (SMT) and ILP to avoid stalls caused by memory latency. When the number of operations per unit of data is high, the processor is less likely to stall from data starvation. Based on the operational intensity perspective, this chapter proposes to perform useful computation during otherwise stalled cycles shifting the computation kernel in the roofline model [WWP09] towards maximum compute performance and away from memory bandwidth limits. The two parallel MCMC samplers considered here are the parallelization over multiple chains from Chapter 2 and the recently introduced generalization, referred to as the Multiple Proposal (MP) sampler, from Calderhead [Cal14].

The design of the MP sampler, that allows parallelization within a chain, stems from the observation that a finite-state Markov chain constructed over a set of proposals maintains the target density. Figure 4.3 illustrates the DAG of

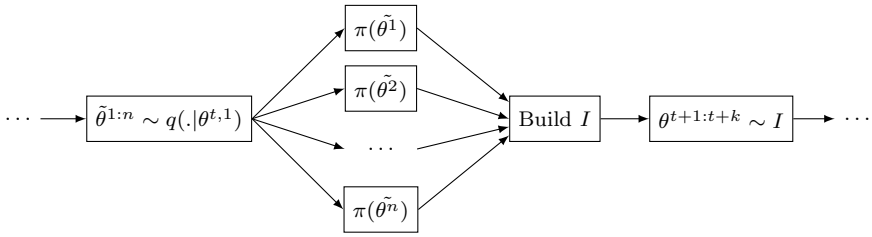


Figure 4.3: An iteration of the MP sampler with proposals $\tilde{\theta}^{1:n}$, $\pi(\tilde{\theta}^i)$ calculated in parallel forming a stationary distribution I from which k samples are drawn.

an iteration emphasizing that the target π can be evaluated at the proposals $\tilde{\theta}^{1:n}$ in parallel.

The goal of this chapter is not to introduce a new type of parallel sampler, but rather to introduce a methodology to restructure existing parallel samplers improving their operational intensity. In other words, rather than focusing on *what* should be executed, a central aspect tackled by other parallel samplers, the emphases is on *how* to execute tasks. Therefore, it can also be applied to other parallel samples like the parallel version of the Independent Metropolis-Hastings (IMH) sampler presented by Jacob et al. [JRS11].

The remainder of this chapter is structured as follows. Section 4.2 provides background on Bayesian Logistic Regression and the target density to be sampled with MCMC. Section 4.3 references related work. Section 4.4 describes the methodology applied to those samplers. Section 4.5 provides results and Section 4.6 concludes the chapter.

4.2 Bayesian Logistic Regression

One benefit of using machine learning algorithms on increasingly larger quantities of data is that more complex structures can be revealed. In addition, it is not uncommon to deal with high-dimensional feature data sets in today's Big Data era. Machine learning on this scale has been a subject of research resulting in methodologies to solve problems under certain constraints [Qiu+16].

As discussed in Equation (2.4), in the Bayesian framework, knowledge about model parameters θ is combined with evidence $\mathcal{D}$ to form the posterior: $\pi(\theta) = p(\theta|\mathcal{D}) \propto p(\mathcal{D}|\theta) \times p(\theta)$. Bayesian logistic regression [P M12] is typically used to model categorical data. Assume independent data entries (x_i, y_i), where x_i

are covariates and $y_i \in \{-1, 1\}$ are binary labels, the log likelihood is given by $\sum_i -\log\left(1 + e^{y_i x_i \cdot \theta}\right)$. All entries stream through the cache at least once to evaluate this posterior.

The methodology outlined below is demonstrated in the context of Bayesian inference using logistic regression on a data set larger than the last level cache since it is both generally applicable, and has a low operational intensity by default; execution becomes dominated by posterior evaluations as the amount of data grows. However, the idea is generally applicable to any parallel algorithm. The ratio of the bandwidth to the latency of successive caching layers will mostly determine the performance improvement.

4.3 Related Work

Scott et al. [Sco+16] explores consensus Monte Carlo algorithms in the Big Data context by proposing to shard data and to break up prior information among workers and combining partial posterior distributions. However, performance is not evaluated from the operational intensity perspective.

Chopin et al. [CR17] discuss to which extent it is sound to showcase a Bayesian computational approach on a binary regression model. Based on their results, instead of relying on sampling from the posterior directly, they advocate to use Expectation Propagation to approximate the posterior although they note dealing with data that has a high dimensionality could require other approximation methods. The reason to use binary regression in this chapter is two-fold. First, a fairly large dataset with 512 features has been selected as it demonstrates the main idea, keeping processors busy during otherwise stalled cycles, well. Second, binary regression generalizes to other levels of the memory hierarchy where the same idea is applicable, an interesting aspect when considering datasets with more features.

Balan et al. [BCW13] reformulate the accept/reject ratio as a statistical hypothesis test. More biased samples can be gathered per unit time reducing the variance of the estimator. The idea of improving results given the same computation time is analogous, but since likelihood evaluations are factorized, the outlined methodology is orthogonal since all data are still used.

Delayed Acceptance introduced by Banterle et al. [Ban+19] is similar in that rejection is possible after partial computation at the expense of increased variance.

Zhao et al. [ZQR16] propose caching middleware with a heuristic and a design allowing explicitly caching control for distributed storage. Since the

methodology below focuses on performance at the CPU level, the opposite approach of restructuring the samplers is needed to leverage caching hardware.

4.4 Hiding latency with Useful Computation

As long as the target density evaluation dominates execution, it seems reasonable to assume that the MC sampler with $\rho = 0$, and the MP sampler should scale with the number of chains or proposals n set to some integer multiple of p. Recall from Section 2.3 that a fraction ρ of the samples is discarded. For this reason, the MP sampler is preferred since $\rho > 0$ in practice. However, assuming a sufficiently large working set, cores compete for memory bandwidth in both parallel samplers. Data fetched into the CPU cache by one of the cores will *potentially* be reused by others. This happens by chance since there are no synchronization points, and as the span of data entries accessed at a given time exceeds cache capacity, performance degrades.

The target distribution for Bayesian logistic regression can be factorized, denoted by $\pi(\theta) = \prod_i \pi_i(\theta)$. This leads to restructuring the samplers from Figures 2.4 and 4.3 as shown by Figures 4.4 and 4.5 respectively. The restructuring assures data entries are loaded exactly once from memory since each core operates on a data partition. The main advantage of this is that n can be set more freely. By increasing n while keeping p constant, useful computations can be introduced during otherwise stalled cycles. Note that increased ILP further reduces execution time.

An assumption made here is $\pi(\theta)$ can be factorized into many more terms than p. This assumption is satisfied by many contemporary datasets. If this is not the case, each core will need to perform a full likelihood evaluation. The main idea of this chapter is still applicable in that each core could be assigned multiple complete likelihood evaluations.

4.5 Results

To compare scalability, performance, expressed as the average number of samples per second, in function of the number of cores was measured. The test system had an Intel E5-2690v2 processor with 10 cores and 32 GB of memory. For reproducibility and reduced variance across runs, operating frequency of the cores was fixed, Operating System (OS) features to migrate memory were disabled, and software threads were affinitized one-to-one with hardware threads.

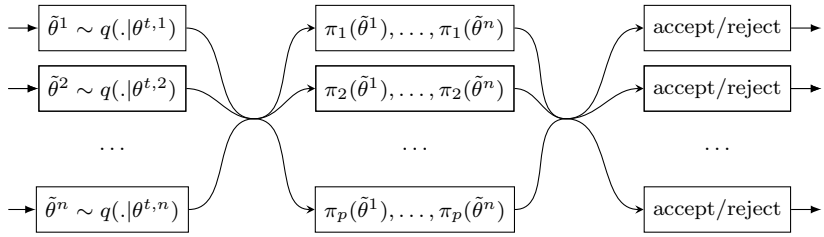


Figure 4.4: An iteration of the restructured MC sampler with n chains. The target distribution π is factorized into $\pi_1, \ldots, \pi_p$ each of which is assigned to one of p cores. Core i computes factor π_i for all n chains. The resulting factors are collected during a reduction phase followed by accepting or rejection to produce n new samples.

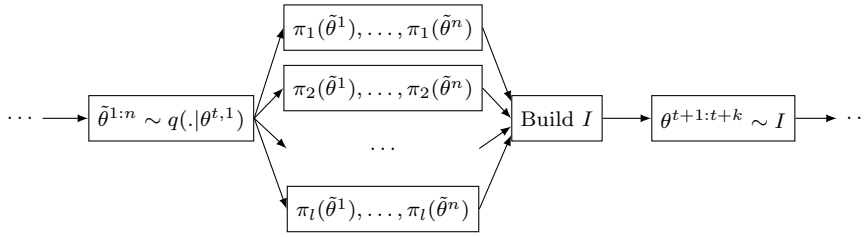


Figure 4.5: An iteration of the restructured MP sampler with n proposals and p cores.

The results highlight improvements for each sampler separately, but, although not the goal in chapter, a comparison shows that the MP sampler outperforms the MC sampler since less samples are discarded.

On the left, Figure 4.6 shows performance of the basic and restructured MC sampler in function of the number of cores. To distinguish from the definition of S_p in Section 2.4, $S_{n,p}$ denotes the speedup with n chains or proposals and p cores. Since ρ is set to 0.5, $S_{n,p} \leq 2$ for the same reason that $S_p < 1/\rho$ as discussed in Section 2.5. The basic sampler scales far less due to reduced per core memory bandwidth while the restructured sampler performs better. First, if $p = n$ then $S_{n,p}$ reaches 2. Second, the best configuration scales far beyond this since n is reduced. The right of Figure 4.6 relates the performance of the restructured sampler with both configurations. Results are expressed in terms of improvement with respect to the basic sampler that suffers from low operational

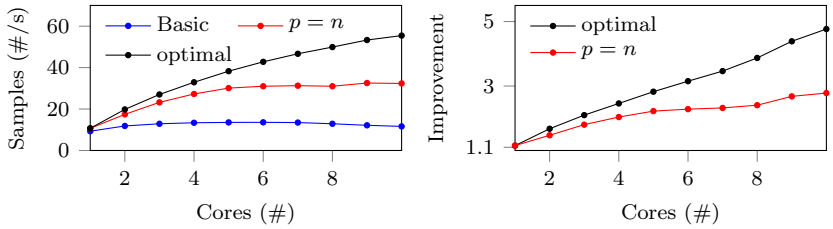


Figure 4.6: Performance of the MC samplers is shown on the left. The basic sampler shows no scaling, while the restructured sampler with $p = n$ is limited by a speedup of 2 and the best configuration scales beyond that. The right shows the improvement by restructuring the MC sampler with respect to the basic sampler. A larger improvement is observed with more cores as the basic sampler becomes more memory bound

intensity. Each data point therefore reflects performance improvements due to restructuring while keeping p fixed and deriving the optimal value for n from Figure 4.8.

Figure 4.7 compares the performance for the basic and restructured MP sampler with and without SMT. The data reflects that the basic sampler is more memory bound since, in relative terms, it benefits more from SMT than the restructured sampler. The number of proposals calculated during each iteration for the basic sampler was determined by the number of hardware threads. Again, performance does not scale linearly with an increase in the number of cores. With the restructured sampler, the number of proposals can be chosen arbitrarily and the optimal configuration is shown in Figure 4.7. The bottom of Figure 4.8 shows performance in function of the number of proposals from which the optimal configuration is determined. The optimal number of proposals is *higher* than the number of hardware threads, as this hides latency caused by data being loaded.

4.6 Conclusion and Future Work

It is well known that the memory subsystem in contemporary systems plays a decisive role in performance. The presented method introduces useful computations to hide stalls resulting in significant improvements. This has been demonstrated in the case of Bayesian logistic regression with two MCMC samplers. The optimal for the MC sampler and the MP sampler was a reduction

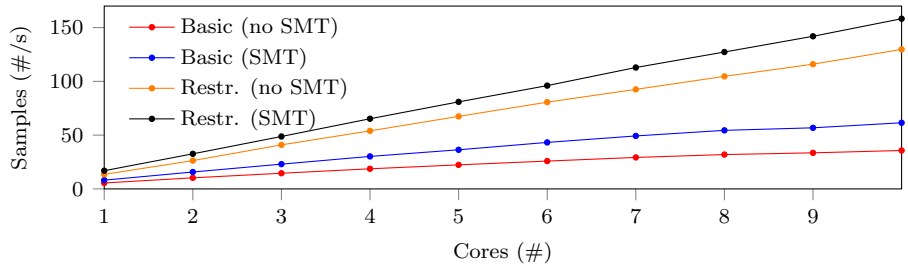


Figure 4.7: Performance of the MP samplers. The restructured sampler scales better than the basic sampler. SMT is relatively less beneficial for the restructured sampler due to an increase in operational intensity.

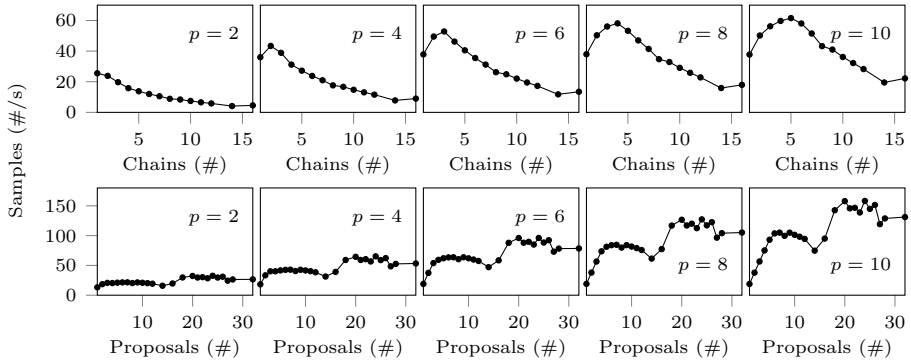


Figure 4.8: Performance of the restructured MC (top) and MP (bottom) sampler in function of n. The optimal n for the MC and MP sampler is between 2 and 5 and around 24 respectively.

in the number of chains and using more than twice as many proposals as cores respectively.

The effectiveness of restructuring with likelihoods that can not be factored is left for future work. For this use-case, the amount of work assigned to each core needs to be at least doubled.

Automatically finding the optimal configuration have been left out since the focus has been on demonstrating the effectiveness of restructuring. Such automatic parameter tuning with stochastic optimization during the initialization phase is desirable for a general solution and is part of future work.

Since the computational kernel of Bayesian logistic regression can be rewritten in terms of a matrix multiplication, another direction for future work is to leverage highly optimized implementations, like those available in the Basic Linear Algebra Subprograms (BLAS) package. The general trend with these implementations is that a higher number of operations can be performed as the input size increases.

In addition to the example of parallel prefetching noted earlier, other samplers where computation is interleaved with stalls while subsequent data elements are fetched, include model selection and ensemble models. Exploring how restructuring can be of benefit here is also part of future work.

Chapter 5

Distributed Affine-Invariant Sampling

5.1 Introduction

MCMC methods, like the Metropolis-Hastings sampler [And+03] and the Gibbs sampler [CG92], can be parallelized by evolving one or more independent chains. As noted in Section 2.5, the downside of this straight-forward approach is that speedup is limited since samples from each chain are discarded as burn-in [Mur10]. Another important issue is that the need for parameter tuning complicates their use.

Goodman & Weare [GW10] introduced a sampler with an affine invariant property. Their sampler performs equally well in statistical terms, measured for example by autocorrelation time [Tho10], on any affine transformation of the target density. Consequently, little or no tuning is required from the user. The sampler has been parallelized by Foreman-Mackey et al. [For+13], and has become popular since it provides *some* speedup from parallelism. However, it performs poorly in a distributed environment as the widened gap between communication and computation cost is neglected.

This chapter improves the sampler of Foreman-Mackey et al. by removing dependencies between computational steps. While these changes do not affect the generated samples, consequently leaving the statistical properties of the parallel sampler unchanged, the decrease in synchronicity makes the sampler perform well in a distributed environment. The observation that allows this is

that pseudorandom choices can be predicted.

The remainder of this chapter is structured as follows. Section 5.2 references related work. Section 5.3 describes the parallel affine invariant algorithm currently in use. Section 5.4 presents the fully decentralized version of the algorithm. Section 5.5 provides and discusses results, Section 5.6 concludes the chapter and proposes future work.

5.2 Related Work

Angelino et al. [Ang+14] speculatively evaluate future positions of a sampler in parallel by relying on determinism in PRNG streams. This chapter builds on the same determinism.

Murray [Mur10] also considers a distributed system by mixing proposals from multiple distributed chains, but this requires global communication. In the sampler outlined by this chapter, communication overhead is minimized by exchanging messages between pairs of chains during each step.

Partitioning and analyzing data simultaneously across multiple machines is a viable solution when data sets are large [NWX14; Sco+16]. These algorithms rely on an expensive reduction step, during which machines are idle. This chapter focuses mostly on compute bound problems. Note also that the presented algorithm is fully decentralized and no reduction is performed at the end.

5.3 Parallel Stretch Move

The stretch move algorithm from Goodman & Weare evolves an ensemble of walkers $\{X_i\}$. The position of each walker X_i is combined with the most recent position of another randomly selected walker, X_j. Foreman-Mackey et al. adjusted the stretch move to allow for parallelism. The version listed in Figure 5.1 is a rewrite of the algorithm to highlight the dependency structure. The parameter a_w is set to 1.5 [CF10] and $\mathcal{U}\{a, b\}$ denotes the discrete uniform distribution over all integers in $[a, b]$. The position of walker i in step s is denoted by X_i^s. Walkers are initialized at random positions where the target has support.

This algorithm follows the fork-join model. The main loop is executed by one thread. Walkers are split into two groups G_1 and G_2 and updates in each group are executed in parallel. Hereafter, each inner loop will be referred to as a partial step. An update involves pairing each walker in the current group with a random walker X_j in the other group. The position of X_i from the previous

step is used during the update. All threads are synchronized after each partial step to ensure all walkers have been updated.

The MPI based implementation provided by Foreman-Mackey et al. translates this into a master-slave model. Slaves listen for positions at which the target density needs to be evaluated. After returning a result to the master, the master replies with more work if not all walkers in the current group have been updated. Execution progresses to the next partial step after *all* walkers in the current partial step have been updated.

Three aspects of this master-slave approach limit performance. First, with many workers, the master is too busy processing requests and workers starve. Second, the time between assigning an update to a worker and receiving a response includes the network latency. As it increases, overall performance degrades. Third, it is unreasonable to assume that evaluating the target at any position takes a constant amount of time. Not only is this caused by characteristics of the target distribution itself, operating system level factors also play a role [Tsa+05]. While load balancing partially alleviates this, synchronization after each partial step is bound to leave workers idle.

5.4 Distributed Sampler

Stochastic algorithms are almost always implemented using PRNGs for practical reasons. Usually, the seed is fixed to make debugging easier and for reproducibility. Building on this idea, if multiple processors execute the algorithm listed in Figure 5.1 with the same seed, each processor will make the same "random" choices. Put differently, any "random" choice can be predicted.

The two uniformly distributed random variables used during the update are independent from the random variable used for pairing walkers. This makes it possible to not only draw $n/2$ random numbers first and pair each walker, but also to generate all the dependencies between all walker positions. The direction of pairing walkers can then be reversed; for each walker, the walkers in the next partial step that will select it are known. If random number sequences do not overlap, each walker can use an independent PRNG state. This is the approach taken here, but alternatives, including reserving blocks from a single random stream or leap-frogging [BM07], are viable as well.

All dependencies can be summarized in a graph as shown by the top of Figure 5.2 for $n = 8$. Nodes represent walker positions. Edges represent dependencies between positions.

Each walker depends on exactly one walker in the previous partial step, while

procedure ParallelStretchMove(π, n)
  $G_1 = \{1, \ldots, n/2\}, G_2 = \{n/2+1, \ldots, n\}$
  Initialize($\{X_1^0, \ldots, X_n^0\}$)
  for $s = 1, \ldots$ **do**
    for $i \in G_1$ **do**
      $j \sim \mathcal{U}\{n/2+1, n\}$
      X_i^s = Update(π, X_i^{s-1}, X_j^{s-1})
    end for
    for $i \in G_2$ **do**
      $j \sim \mathcal{U}\{1, n/2\}$
      X_i^s = Update(π, X_i^{s-1}, X_j^s)
    end for
  end for
end procedure

procedure Update(π, X_i, X_j)
  $z = ((a_w - 1) \times \mathcal{U}(0,1) + 1)^2 / a_w$
  $\tilde{X}_i = X_j + z(X_i - X_j)$
  if $\mathcal{U}(0,1) < z^{n-1} \times \exp\Big(\pi(\tilde{X}_i) - \pi(X_i)\Big)$ **then**
    return $\tilde{X}_i$
  else
    return X_i
  end if
end procedure

Figure 5.1: The parallel stretch move algorithm [For+13] rewritten to highlight the dependency structure. A target distribution π and the number of walkers n are provided as input. Walkers in groups G_1 and G_2 are updated alternatingly. Stop conditions are omitted.

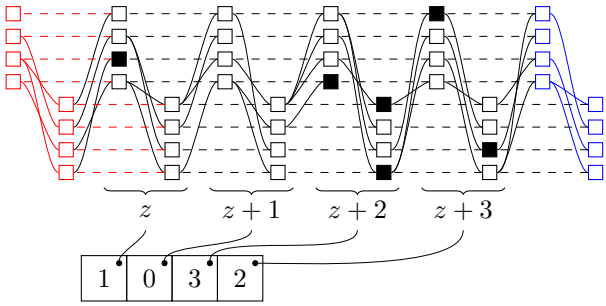


Figure 5.2: Walker positions are tracked and part of the dependency graph is kept in memory. The active walker positions are filled in black. Red depicts dependencies that have been removed from memory. The dependencies shown in blue will be generated when a walker enters that step.

the walker itself can be used by up to $n/2$ walkers from the next partial step. A walker update always involves the previous position of the same walker, shown as a dashed edge. Every dependency graph will follow this structure, although the solid edges might differ if another seed is chosen.

This dependency structure is loosely coupled. In theory, although unlikely, it is even possible that the graph is disconnected. Since $|G_1| = |G_2|$, it is almost certain that there will be walker positions that will not be used by other walkers. The probability that a given walker is not selected is $(1 - 2/n)^{n/2}$ which approaches $1/e$ quickly as $n \to \infty$. With $n = 20$, the probability is already 0.35. Hence removing global synchronization after each partial step is beneficial. If dependency graph management overhead is kept to a minimum, wait time can be reduced.

Increasing n makes the dependency structure more loosely coupled allowing more communication delay to be hidden. At one extreme, if one sample is collected from each walker, the problem becomes embarrassingly parallel. However, to get independent samples, each walker needs to take enough steps. Statistical effects of the choice for n is out of the scope of this work as the effects are the same in both a shared and a distributed-memory system.

The graph can be used in a master-slave model to reduce wait time in comparison to the fully synchronized version from Foreman-Mackey et al. More importantly, in a fully decentralized system, assuming a homogeneous cluster, an equal share of walkers can be assigned to each processor. Such an assignment

can be computed directly and ensures that no communication is required for the dashed lines from Figure 5.2. Other dependencies are not taken into account.

Even if n is too low to hide all communication, using the graph in a distributed system will outperform the master-slave samplers since only one-way communication delay is paid. A two-way communication delay would need to be paid if each position would be requested explicitly. The PRNG seed used to create the dependency graph is a single number that encodes all these requests.

Figure 5.3 outlines the distributed sampler. Each processor runs a loop that multiplexes two operations: receiving and buffering results, and executing tasks. A task stores all the data required to update a walker. The scheduler matches walkers with incoming dependencies using a hash table.

Each task is placed in the scheduler, which internally places the task in a ready-queue if all dependencies are available. After a task is executed, its result needs to be sent to each processor that runs tasks that depend on it. Note that if a processor runs multiple tasks that depend on it, the result is sent only once to that processor. This optimization is implemented with a lookup table to detect to which processor the result has already been sent. It reduces communication overhead if multiple walkers assigned to a processor depend on the same result. At reception, results are stored in memory. A result is removed from memory after all tasks that depend on it have moved to the ready-queue.

It is memory inefficient to store the whole graph in memory in each process. Even if available memory is not a limiting factor, random access in the graph is expensive since the data will likely be out of the CPU cache [Dre07]. By tracking all current walker positions, it is enough to store only a part of the graph in memory as shown by Figure 5.2. A double-ended queue that allows direct access to its elements is maintained together with an offset, z. A slice of the graph is generated when the queue is extended.

Since each processor collects samples from its assigned walkers, and walker positions are only shared on a need-to-know basis, samples are spread across processes. Samples can be aggregated at one process either during execution, or by having each process store its samples on a distributed file system. The latter approach has been taken in the implementation that is evaluated in Section 5.5.

The goal of any scheduler is to choose an order of execution to optimize some metric without influencing the results. For the distributed sampler, the scheduler chooses which of the executable tasks should be prioritized among those for which the dependent position has been received. A First-Come-First-Serve (FCFS) scheduler executes tasks in the order that they become executable. The benefit of such a scheduler is that it requires minimal resources. In contrast to the FCFS scheduler, the heuristic described in this section makes choices to reduce wait

```
procedure DistibutedStretchMove()
    W = {X_i | i mod p is the id of this processor}
    T = {t | t initializes a walker in W}
    add each t ∈ T to scheduler
    loop
        while receives pending do
            receive position v
            T = {d | task d depends on v}
            mark each d ∈ T as executable
        end while
        if scheduler has executable tasks then
            take a task t from scheduler and execute
            T = {d | task d depends on t}
            N = {n | process n runs task t ∈ T}
            send result of t to each n ∈ N
            add t to scheduler
        end if
    end loop
end procedure
```

Figure 5.3: The distributed stretch move on a processor. Each processor is assigned an equal share of walkers. A scheduler takes care of matching received results with tasks that require those results. Although not shown, execution is halted after enough samples have been collected.

time. While the samples remain the same, the order in which they are taken changes depending on the choices of the scheduler. If required, the order can be restored by keeping track of the index of each sample.

The ideal scenario is one where no time is lost waiting for new executable tasks since it causes efficiency to drop as measured by the ratio between actual speedup and theoretical linear speedup. As noted earlier, one parameter that influences how much network latency can be hidden is n. If network latency is at most l units of time, and a task takes u units of time to execute, each process should be assigned at least $\lceil l/u + 1 \rceil$ walkers in each partial step to hide all communication.

In practice, task execution time is not known precisely, so avoiding idle times is hard or even impossible even if the whole dependency graph would be considered upfront. However, executing some tasks before others can help the system to progress faster. First, walkers that are in earlier steps should be updated first. As noted in Section 5.4, the span of steps is limited. Hence, if walkers in lagging steps are not updated, other walkers are expected to block. Second, within a partial step, walkers that have not been selected should not be executed first. By delaying execution of these walkers, the execution itself hides communication at the end of the partial step. During this period, the process would otherwise potentially wait for results for the next partial step.

A more intelligent scheduler that respects these aspects can be implemented using a priority queue on each processor. Tasks within an earlier partial step take priority over tasks in later partial steps and tasks within the same partial step are prioritized by having dependent tasks or not. Since prioritization only makes sense with many tasks, polling is required to receive all available results in the first part of the loop in Figure 5.3 while a FCFS scheduler need only receive positions until a tasks becomes executable. Therefore, prioritization is not always beneficial as polling and maintaining the queue adds overhead. The overhead is only justified if the penalty of the random choices made by the FCFS scheduler is high enough.

5.5 Results

Two target distributions with differing communication to computation ratios are considered for evaluation. The master-slave sampler with automatic load balancing is compared to the graph based master-slave sampler and the distributed sampler. Care has been taken to minimize measurement noise by collecting enough samples.

The first target is a Rosenbrock density from Equation (2.6). Each evaluation takes the same number of operations. On contemporary systems, this takes approximately 250 nanoseconds. To put this into perspective, network latency is on the order of microseconds on the cluster used for evaluation. This places evaluations of the Rosenbrock function on the high end of the communication to computation ratio spectrum. In the master-slave model, the time elapsed between transmitting the position to a worker, computing the result, and receiving the result will be dominated by the network latency.

The second target is the likelihood of a Fitzhugh model given by the two coupled ordinary differential equations from Equation (5.1). It is used to model relaxation oscillators.

$$\begin{aligned} \dot{v} &= v - \frac{v^3}{3} - w \\ \dot{w} &= \phi(v + a - bw) \end{aligned} \tag{5.1}$$

The target density is given by Equation (5.2) where Σ is a covariance matrix $\psi(t, a, b, \phi) = [v(t, a, b, \phi)\ w(t, a, b, \phi)]^T$, $p(a, b, \phi, \Sigma)$ is a prior distribution, and $\{(y_i, t_i)\}$ are independent data elements. An important aspect of these problems is that the execution time of numerical integration is variable depending on the choice of $\theta = [a, b, \phi, \Sigma]$. A small change in θ can not only drastically change the output, but also the time it takes to produce it.

$$\pi(a, b, \phi, \Sigma | y_i, t_i) = \prod_i \mathcal{N}(y_i | \psi(t_i, a, b, \phi), \Sigma) \times p(a, b, \phi, \Sigma) \tag{5.2}$$

5.5.1 Performance Evaluation

Figure 5.4 compares the distributed sampler with the master-slave sampler with and without the dependency graph. In each of the plots, the baseline for speedup is the currently available master-slave version running on one node.

The per-node process count is 12 as each node has 12 cores. The parallel system is arranged in a star topology and uses InfiniBand. Communication is offloaded to the hardware through the Intel MPI library.

First, note that the master-slave setup is particularly unsuited for the Rosenbrock target. Performance *degrades* with more nodes. As discussed earlier, network overhead dominates the time between sending a position and receiving the result. With more nodes, the average latency between processes increases since they are separated by a longer distance. The probability that a walker selects a position residing on the same processor drops. In addition, the master

is unable to keep all workers busy. The same is true whether each partial step is followed by global synchronization or when the dependency graph is used. Consequently, performance *drops* with more nodes.

Next, consider the distributed sampler on the Rosenbrock target. Performance scales not only because computation happens in parallel, but messages are also exchanged in parallel. As expected, more computation can be used to hide communication as n increases. The difference in performance for the master-slave versions as n increases is negligible due to relatively high network latency.

Finally, compare sampler scalability with the Fitzhugh model. With a small number of walkers, the master in the master-slave versions quickly becomes the bottleneck due to communication and synchronization overhead; performance is constant starting from 4 nodes. As the results show, a dependency graph is beneficial even in this case. Again, the distributed version does not suffer from the same bottleneck. Increasing the number of walkers improves performance of all samplers due to more loosely coupled dependencies. This time, however, the master-slave versions also show improved scaling since communication delay is relatively small compared to the time required to integrate the system of ordinary differential equations from Equation (5.1). With $n = 1024$ on the Fitzhugh target, the master-slave approach with a graph slightly outperforms the distributed sampler up to 10 nodes. The fixed assignment of walkers to processes for these configurations is inferior to dynamically assigning walker positions from a master to idle workers. Still, the difference in performance is minimal and, in general, the distributed sampler is preferred.

5.5.2 Scheduling Heuristic Performance

To evaluate the benefit of the scheduling heuristic, running time with three nodes is compared with the FCFS scheduler. The results in this section should be treated separately. The parallel system here consists of three nodes each with 20 cores and the one-way network delay on this network was approximately 150 microseconds. Performance is measured while increasing per-node process count from 1 to 20. The number of walkers n is kept constant. The baseline is the FCFS scheduler performance with three nodes and one process per node.

As noted earlier, the Rosenbrock target has a high communication to computation ratio. Speedup with this target is shown in Figure 5.5. To hide communication delay, n is set to 2048. The per-process walker count drops from $2048/3 \approx 682$ walkers to $2048/60 \approx 34$ walkers. The overhead of maintaining the priority queue, when compared with the evaluation of the target, is significant. With many walkers per node, and relatively high network delays, the FCFS

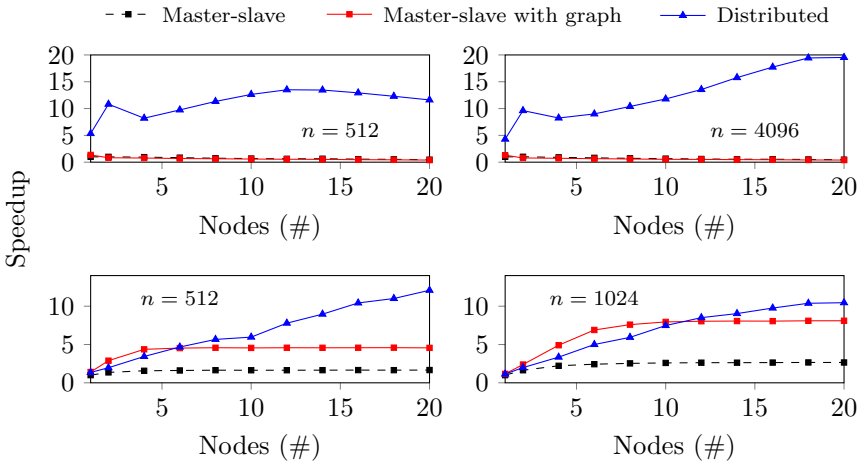


Figure 5.4: Comparison of sampler scalability for the Rosenbrock target (top) and Fitzhugh target (bottom). The baseline for speedup in each plot is the master-slave version with one node.

scheduler performs better. In the worst case, performance drops by 55%. With a few walkers per process, there is no difference between the two schedulers.

Since the Fitzhugh target takes tens of milliseconds to evaluate on average, a suboptimal scheduler choice causes nodes to wait. This is shown by Figure 5.6 with $n = 512$. Initially, the per-process walker count is high, in which case there is little to no difference in execution time between the FCFS scheduler and the heuristic. With only a few walkers per node, choosing the correct one to execute first becomes more important. The net improvements of using the heuristic instead of the FCFS scheduler are between 0% and 15%.

As expected, depending on the configuration, a heuristic further improves performance. With targets that take even longer to execute than the Fitzhugh target, the penalty of bad choices rises further and the improvements are expected to be even higher.

5.6 Conclusion and Future Work

This chapter contributes a novel distributed affine invariant sampler. Two target distributions are used to evaluate scalability. The results show that, in practically

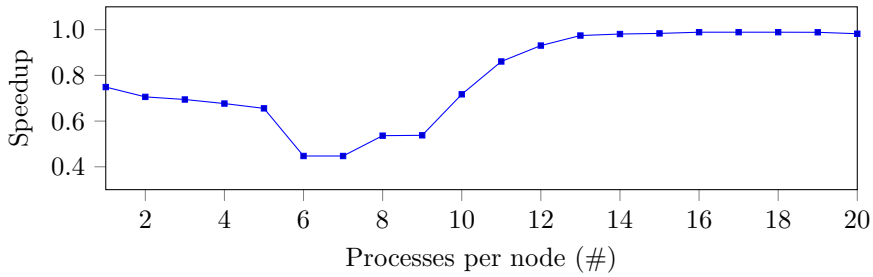


Figure 5.5: Speedup of using the scheduling heuristic with the Rosenbrock target. The baseline for speedup is the FCFS scheduler. The benefits of the scheduling heuristic do not outweigh the penalty of the FCFS scheduler for this target.

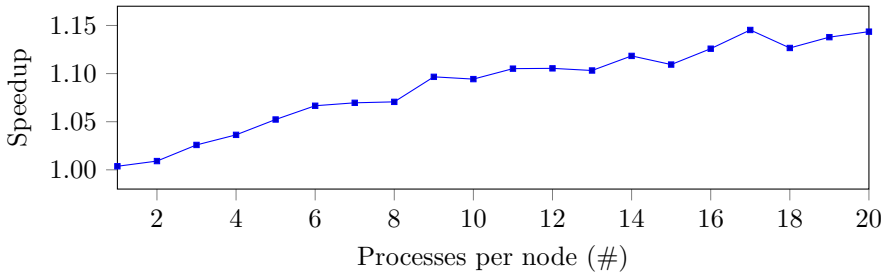


Figure 5.6: Speedup of using the scheduling heuristic with the Fitzhugh target. The baseline for speedup is the FCFS scheduler. Since n is fixed, the per-process walker count is lower with more processes. With less walkers per process, the penalty of suboptimal scheduling decisions is higher and the overhead of the heuristic is lower.

all configurations, the distributed sampler outperforms the master-slave sampler. The speedup ranges from 10 to 20 depending on the target.

By observing that PRNGs are deterministic, "random" choices can be predicted. The dependency structure, conceptualized as a graph, is shown to be loosely coupled providing motivation that global synchronization is excessive.

Through instrumentation, it was observed that a significant portion of time was spent in MPI calls. Future work includes studying how tuning of MPI parameters could be beneficial for the distributed sampler.

To avoid the overhead of tracking the task graph, threads within a system could share a thread-safe task graph. This architecture would avoid some of the MPI calls and load could be balancing dynamically within each node.

By prioritizing some tasks, wait-time is further reduced when the target density requires a significant amount of computational effort. Alternative policies could still improve performance.

In the presented sampler, processes communicate in a full-mesh topology, although no broadcasts are used. In reality, some processes are closer to others either because they reside on the same node or in the same rack. To further reduce wait time, mapping the dependency graph to the structure of the underlying system will be explored.

Chapter 6

Relaxing Scalability Limits with Speculative Parallelism

6.1 Introduction

Sampling for Bayesian inference with non-linear models requires tools that can cope well with hard to navigate distributions. SMC Samplers are among the currently best known methods as they tend to cope well with such distributions. They inherently provide parallelism, but speedup is limited for models that exhibit a high degree of imbalance across posterior distribution evaluations.

Dependencies between computational steps form the biggest obstruction for parallelization. For example, a computational task consisting of two parts might seem impossible to parallelize if the output of the first part forms the input to the second part. Speculative execution is known to be a useful technique to leverage more parallel compute resources in such scenarios [Gra+03]. By predicting the output of the first part, the second part can *tentatively* be executed in parallel. If it turns out that the output was predicted correctly, the tentative result will be available sooner, resulting in a reduction of execution time. On the other hand, when an erroneous prediction was made, the result is discarded.

It is important to note that almost certainly some compute resources will be wasted when speculation is used, but the amount of resources wasted by speculation is outweighed by the benefits for SMC samplers. It turns out that it is easy to predict the result of the part of the algorithm that would cause computation to stall. Furthermore, predictions can be improved by a simple

renumbering scheme.

The remainder of this chapter is structured as follows. Section 6.2 references related work. Section 6.3 discusses Sequential Monte Carlo Sampler algorithms. Section 6.4 describes the effects of load imbalance. Section 6.5 details and motivates implementation choices. Section 6.6 lists and discusses results. Section 6.7 concludes the chapter and Section 6.8 gives directions for future work.

6.2 Related Work

Angelino et al. [Ang+14] explore speculative execution in the context of the basic Metropolis-Hastings sampler. This sampler is inherently sequential since each step in the Markov Chain depends on the previous step. A proposal is either accepted or rejected, with a bias towards being rejected. The authors note that all possible decisions form a binary tree. They leverage parallelism by speculatively evaluating potential future steps in this tree, prioritizing the nodes that will most likely be used. The prioritization is based on an inexpensive approximate distribution. Similarly to their work, the sampler presented here speculatively evaluates potentially useful future steps. However, the goal is different. Since SMC already provides parallelism, the goal here is to use speculation in a sequential fashion to relax the limits imposed by load imbalance. The speculative tasks introduced by Angelino et al. are orthogonal as they *add* parallelism. Their approach could be added to the speculative SMC sampler to further improve scalability.

Marendić et al. [Mar+12] also note how reductions degrade efficiency under load imbalance. They consider use-cases where work and associated data is partitioned across processors and reductions require communicating with the appropriate neighbors. To alleviate the impact on efficiency, they dynamically build a rebalanced reduction tree by finding neighbors that have already terminated. This allows reductions to proceed between processors that would otherwise wait on processors with more compute intensive tasks. The reduction in SMC samplers is a simple summation over weights. This is computed in a negligible amount of time when compared to the overall amount of work. Instead, the focus is a synchronization barrier between steps that causes processors to wait due to load imbalance.

Another approach taken by Proficz [Pro18] to tackle load imbalance with reductions is to exchange information about the progress on the tasks. It is then possible to predict when the tasks will finish, and which processors should work together to complete the reduction. This technique is not applicable to the SMC

sampler for the same reasons listed above.

6.3 Sequential Monte Carlo

Sequential Monte Carlo Samplers are a class of algorithms that iteratively move a set of n weighted particles through a sequence of distributions [MDJ06; DFG01]. These algorithms are useful in the Bayesian setting since they can be set up to smooth out modes in the target distribution during the initial iterations. Consequently, particles do not get stuck quickly before the target distribution is explored.

6.3.1 Moving Through a Sequence of Distributions

In SMC samplers, a sequence of distributions $\pi_1, \ldots, \pi_s$ is used sequentially. The sequence starts with a distribution π_1 from which samples can be generated directly during the initialization step. All weights are initialized to 1. Next, particles are moved and weights are updated while the underlying distribution is changed to π_2. The central idea is that as long as π_1 and π_2 are not too different, particles will remain in regions of high density. The process is repeated while the underlying distributions gradually approach some target distribution $\pi_s = \pi$.

There are many possible choices for the sequence of distributions. For Bayesian inference with non-linear models, a tempered sequence, given by Equation (6.1), is used.

$$\pi_i(\theta|X) \propto \pi(X|\theta)^{\gamma(i)} \cdot \pi(\theta) \tag{6.1}$$

Here, X denotes evidence and θ denotes the model parameters. The posterior $\pi_i(\theta|X)$ is given by the product of the likelihood $\pi(X|\theta)$ with the prior $\pi(\theta)$ up to a normalizing constant. The rate of tempering is controlled by a tempering schedule γ. The initial distribution, obtained with temperature $\gamma(1) = 0$, is the prior. For the final distribution, $\gamma(s)$ is set to 1, resulting in the true Bayesian posterior. Intermediary values for γ allow for further control.

The assumption made here is that likelihood evaluations dominate run-time. This is a realistic assumption to make since for many problems, computation intensive models are simulated as part of likelihood evaluations. Note that the findings are still applicable if significant time is also spent on evaluating the prior. In that case, evaluations can be scheduled as two separate concurrent tasks.

```
procedure SMC(π(X|θ), π(θ), γ, n, s, t)
    for i = 1, ..., n do
        INITIALIZE(θ_i)                          ▷ Draw θ_i ∼ π(θ)
    end for
    for i = 2, ..., s do
        π̃_i(θ|X) = π(X|θ)^γ(i) · π(θ)
        for j = 1, ..., n do
            MOVE(θ_j, π̃_i(θ|X))                  ▷ Also update weight w_j
        end for
        if (Σ w_j)^2 / Σ w_j^2 < tn then          ▷ ESS
            RESAMPLE(θ_1, ..., θ_n; w_1, ..., w_n)
        end if
    end for
end procedure
```

Figure 6.1: Algorithmic structure of Sequential Monte Carlo Samplers with tempering to generate n samples from a Bayesian posterior using s steps. One particle is used for each sample. A likelihood $\pi(X|\theta)$ and a prior $\pi(\theta)$ describe the target distribution. In addition, a function γ sets the rate of tempering. A threshold $t \in [0, 1]$ specifies when particles are deemed degenerate. To counteract degeneracy, particles are resampled.

6.3.2 Weights and Resampling

A weight w_i is associated with each particle. The procedure to update weights depends on the choice of kernel used during particle updates. For the purpose of the discussion here, a Metropolis-Hastings MCMC kernel is chosen for the particle moves [MDJ06]. After all particles have moved once, the ESS is approximated to determine if the particles exhibit degeneracy. This approximation is given by $(\sum w_i)^2 / \sum w_i^2$. If the ESS falls below a threshold, particles are resampled. Typically, this happens rarely, a property exploited throughout the speculative SMC sampler. This process is similar to fitness proportionate selection in genetic algorithms. The resampling process consists of drawing n samples where the normalized weight of each particle determines the probability that the particle will be selected. Next, the weights are reset and the whole process is repeated iteratively s times [MDJ06]. The structure of this algorithm is outlined in Figure 6.1. Within each iteration, particle moves can occur in parallel, but calculating the ESS requires all moves to be finished.

6.4 Load Imbalance

For the sake of brevity and clarity, all terms relating to overhead have been omitted in this section. Only the time required to evaluate the likelihood for a particle i in step s, denoted by $\delta_{i,s}$, is considered. It is therefore important to keep in mind that the limits are an estimate. Some time is spent on bookkeeping tasks and an interconnect always introduces some delays. Since likelihood evaluations dominate execution time in realistic use-cases, the derived limits should still be representative.

6.4.1 Scalability Limit Due to Load Imbalance

Consider the ratio between the sequential execution time and the parallel execution time [Gra+03]. For a specific step s the speedup is limited by Equation (6.2). Here, $\delta_{\max,s} = \max\{\delta_{i,s}\}_i$ is the execution time with an infinite number of processors or the minimum amount of time required for the step regardless of how many processors are employed and Δ_s is the sequential execution time. More interestingly, the same equation can also serve to quantify load balance as S_s is close to s only when $\delta_{\max,s} \approx \delta_{i,s}$ for all i.

$$S_s = \frac{\Delta_s}{\delta_{\max,s}}, \quad \text{where } \Delta_s = \sum_{i=1}^{n} \delta_{i,s} \tag{6.2}$$

The maximum overall speedup S for a run is a weighted average over all S_s, where the weight is given by $\delta_{\max,s} / \sum_{i=1}^{s} \delta_{\max,i}$. As expected, this can be written as the ratio between the total sequential execution time $\sum_{j=1}^{s} \Delta_j$, and the shortest possible parallel execution time given by the sum over the minimum time required to execute each step $\sum_{k=1}^{s} \delta_{\max,k}$.

Consider two consecutive steps s and $s+1$. The speedup is limited by the left-hand side of Equation (6.3). The limit can be relaxed if the synchronization barrier is removed. The relaxed limit is given by the right hand side of Equation (6.3), where $\delta_{\max,[s,s+1]} = \max\{\delta_{i,s} + \delta_{i,s+1}\}_i$.

$$\frac{\Delta_s + \Delta_{s+1}}{\delta_{\max,s} + \delta_{\max,s+1}} \leq \frac{\Delta_s + \Delta_{s+1}}{\delta_{\max,[s,s+1]}} \tag{6.3}$$

This inequality clearly holds; the worst case occurs when both $\delta_{\max,s}$ and $\delta_{\max,s+1}$ are associated with the same particle. For all other cases, $\delta_{\max,[s,s+1]} > \delta_{\max,s} + \delta_{\max,s+1}$. Figure 6.2 illustrates a more favorable scenario where $\delta_{\max}$ is associated with a different particle in each step.

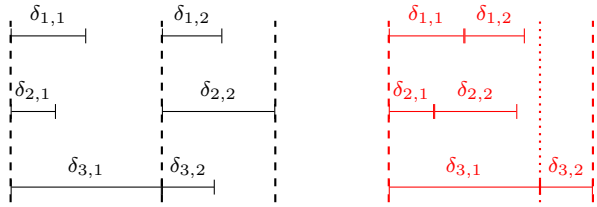


Figure 6.2: Effect of removing the synchronization barrier between two consecutive steps. In the example shown, the minimum duration to update all particles is determined by $\delta_{max,1} = \delta_{3,1}$ and $\delta_{max,2} = \delta_{2,2}$ for the first and the second step respectively. Dashed lines represent barriers between steps. The dotted line on the right represents the synchronization barrier that has been removed.

Note also that speedup is limited by the number of particles. Each particle can be updated in parallel, but subsequent updates to the same particle need to happen sequentially. As will be discussed in Section 6.5.5, it is possible to skip some updates. This further introduces imbalance and limits speedup with respect to sequential execution. However, both sequential and parallel runs will take less time since there is less work to be executed.

6.4.2 Scalability Limit After Removing Barriers

It turns out that by continuing execution speculatively, most barriers can be removed. To find the limit on scalability after these barriers are removed, consider the total running time required to update a particle i from its initial position to step s, denoted by $D(i, s)$. This is given by Equation (6.4), where $P(i, s)$ denotes the particle from which particle i is moved in step s. Speculative updates are performed under the assumption that $P(i, s) = i$. For the initialization step, $D(i, 0) = \delta_{i,0}$.

$$D(i,s) = \begin{cases} D(i,s-1) + \delta_{i,s} & \text{if } P(i,s) = i \\ \max\{D(k,s-1)\}_k + \delta_{i,s} & \text{otherwise} \end{cases} \tag{6.4}$$

For every update, the duration $\delta_{i,s}$ needs to be taken into account. When a resampling step occurred, a particle is moved from a different particle only when $P(i, s) \neq i$. At this point, it is known that the assumption was incorrect and subsequent updates can only take place after all particles in the previous step have been updated.

This relaxes the scalability limit from the one given in Section 6.4.1 to $\sum_{j=1}^{s} \Delta_j / \max\{D(i,s)\}_i$. Note that this ratio is always larger than the one given in Section 6.4.1 and that the renumbering step described in Section 6.5.4 further raises the limit. This coincides with the intuition that removing barriers never degrades performance. In practice, there is some overhead, associated with speculation. This is especially true in a master-slave architecture such as the one studied in this chapter. Here, the master spends cycles processing useless speculative results or when speculative results are occupying network bandwidth, slowing down transfer of non-speculative results. However, as will be shown in Section 6.6, even when particles are resampled frequently, speculation accuracy is high enough to outweigh these costs.

6.5 Speculative Sampler

The speculative sampler presented here employs a master-slave architecture where one processor runs the algorithms and takes care of all the bookkeeping, and other processors request work from the master. The ingredients for building the speculative sampler are detailed in this section.

The use-cases considered for evaluation in this chapter exhibit a high degree of imbalance. A master-slave architecture already tries to cope with this as it provides dynamic load balancing; when a slave becomes idle by returning a result to the master, the master sends the next piece of work.

6.5.1 Treating Steps With and Without Resampling Uniformly

To simplify implementation, no distinction is made between steps with and without resampling. Each particle update consists of moving a particle *from* the position of some other particle in the previous step to a new position. A table, referred to as the "from table", is maintained to keep track of the origin particle to move from. If no resampling has occurred, this table maps each element to itself. When resampling did occur, the selected samples are stored in the table. Speculative execution consists of assuming that the index of the origin particle is the same as the particle from which the move is to be executed. It should be clear that even when particles have been resampled, the move can still correctly be speculated if the i^{th} selection happens to fall on the i^{th} particle. The selection procedure for multiple steps is shown in Figure 6.3.

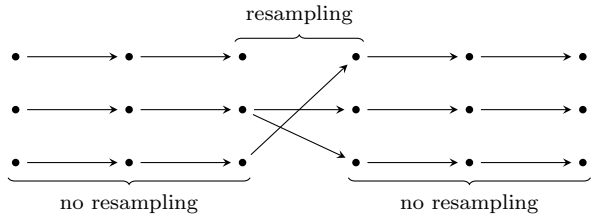


Figure 6.3: Three particles moving through six steps. After the second update, particles are resampled. One particle is selected twice. All steps are treated uniformly, whether resampling occurred or not. The position of a particle in each step depends on the position of a particle in the previous step.

6.5.2 Speculative Execution

An important ingredient for speculation is the ability to rollback execution. Updates involve consuming random numbers from a PRNG stream. New positions are proposed and accepted probabilistically. While not required, it is advisable to use a PRNG that can quickly jump in its sequence [Sal+11] to minimize the overhead of rolling back states. Alternatively, a standard PRNG can be used if the sequence is pregenerated and stored in random access memory.

The goal is to continue evaluating particles in the next steps even when not all particles in the current step have moved. To gain insight into how this works, consider n particles of which one particle is moved to a position where the likelihood takes an inordinate amount of time to evaluate. Updates on all other particles will finish relatively quickly. Speculative evaluation will start for the next steps of these particles. It might even be possible that *all* remaining steps for these particles will be evaluated before the evaluation on the lagging particle finishes.

If it turns out that resampling was required, results for the particles that are not selected will be discarded. To allow rolling back to the correct state, the sequence of positions, one for each step, needs to be kept for each particle. Reverting to a previous state is then a matter of selecting the right index in the sequence. For this part of the algorithm, space requirements increase by a factor of at most s. This happens when, during the first step, all future steps are speculatively executed. A more advanced memory management mechanism would be required when this becomes the limiting factor, but such a mechanism is beyond the scope of this chapter.

6.5.3 Parallelism and Speculation

Let p denote the processor count. In comparison to the particle count n there are three cases to consider: $p \ll n$, $p \approx n$ and $p > n$. Note that in general, each processor will evaluate approximately n/p likelihoods per step.

In case $p \ll n$, each processor evaluates many particles in each step and speculation only provides marginal benefits. This can be seen in two ways. First, consider when the first speculative task is launched. Within a step, this occurs when, out of n particles, $n - p + 1$ updates have finished. At this time, $p - 1$ processors are still working on the last non-speculative evaluations. In other words, speculation occurs only at the end of a step. Second, since $n/p \gg 1$ likelihoods are evaluated at each processor, imbalance is automatically reduced due to the Central Limit Theorem [Fil10]; all processors will finish at approximately the same time and idle time will be minimal. Note that at one extreme, where $p = 1$, load is balanced by definition. Another point worth mentioning is that due to speculation, execution on the slaves can continue while the master computes the ESS and potentially performs the resampling step. In comparison, without speculation, all slaves are idle during this time.

The scenario in which speculation is expected to yield the most benefits is where $p \approx n$. Approximately all likelihood evaluations will start at the same time. Without speculation, processors that are assigned with tasks that complete quickly will need to wait on other processors due to the synchronization barrier. With speculation, these processors can continue with likelihood evaluations for particles in the next step.

To avoid having to modify code inside the model, a likelihood evaluation that has commenced cannot be canceled; processors commit until completion. It might seem that using $p > n$ processors is superfluous, but with additional processors bad speculation will have less negative consequences; a committed processor will not affect performance as another will take its place. To put it differently, as some bad speculation is to be expected, the scalability limit with barriers removed will typically be reached with a slightly higher processor count than when all speculative results are be useful.

6.5.4 Renumbering Particles

As noted above, speculation after a step in which particles have been resampled is not always incorrect. As long as the i^{th} selection falls on the i^{th} particle, the speculative result can be used. To increase the probability that these events occur, the "from table" containing selections is reordered to ensure that indexes

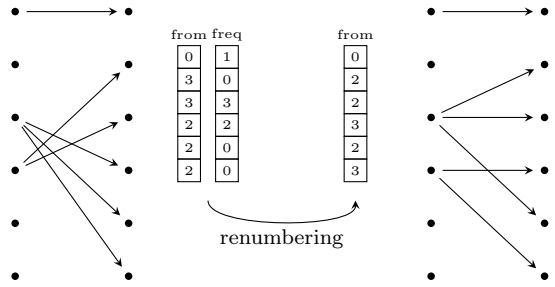


Figure 6.4: The renumbering process with six particles. The original selections, stored in the "from table", are depicted on the left. The frequency of each selection is easily derived from this table. Renumbering ensures that the i^{th} selection falls on the i^{th} particle as often as possible, improving the accuracy of speculative execution.

coincide with selections as often as possible.

First, a frequency table, where entries denote how often each particle has been selected, is created. Next, for each index i, the frequency table is consulted to see if at least one selection has fallen on particle i. If this is the case, the i^{th} selection is set to the i^{th} particle. Finally, all missing entries are filled using the remaining counts in the frequency table. An example is shown in Figure 6.4 with six particles.

A PRNG is associated with each particle. Renumbering can be seen as exchanging PRNGs between particles to maximize speculative usefulness. Since each PRNG stream outputs another sequence, renumbering influences the output of the sampler. Note however that the random numbers in each stream are still distributed identically, which in turn ensures that the distribution of the output samples is left unchanged. Accordingly, the statistical properties of the sampler are left unaltered.

All other alterations to the classical SMC sampler do not influence the output of the sampler in any way. Renumbering is therefore enabled in the original algorithm for a fair comparison study with the speculative sampler; the same samples are generated in either case.

6.5.5 Skipping Updates

Interestingly, when MCMC kernels are used during particle moves, together with the backward kernel proposed by Del Moral et. al [MDJ06], it turns out that

this allows weights to be updated *before* particles are updated themselves since the weight depends only on the value of the previous likelihood. In addition, since weights determine if resampling is required, it is possible to perform the selection portion of the resampling procedure *before* the update as well.

Consequently, updates for particles that are not selected can be skipped. These particles are detected using the same frequency table that tracks how frequently each particle will be selected during the subsequent resampling procedure. Since this information is already available, finding which updates can be skipped is a relatively inexpensive operation and should therefore always be included.

6.5.6 Implementation Details

Figures 6.5 to 6.7 summarize the speculative sampler implementation. All steps are detailed below. To ease implementation, placeholders for long-running tasks, called futures [BH77; Hal85], are used extensively.

Each iteration starts by updating weights and determining which particles will be selected in the *next* step. Note that this is different from the listing in Figure 6.1 where particles are updated *before* weights are updated and resampling occurs. Next, tasks to move the particles are scheduled. A future for each update is returned. Futures are also composable to form a Directed Acyclic Graph (DAG) of computation [Hel+17]. In the speculative sampler, futures representing completion of particle updates in a step are composed into a single future F_s which is fulfilled when all particles have been updated. This composition allows to run the scheduler until all particles in a step have been updated.

As non-speculative execution progresses, and a new iteration is started, a new particle update is only scheduled if no valid computation has been scheduled yet or if there is a version mismatch with the previously scheduled computation. However, if a valid future is stored for that update, it is returned immediately. Either the future has already been fulfilled, or it will be when the slave executing the speculative task will return its result and the post-processing step has finished executing. In either case, the result will be used.

Finally, control is transferred to the scheduler which will return when the future F_s is fulfilled. The scheduler will prioritize dispatching scheduled tasks first. When all slaves are busy, the scheduler waits until a slave becomes idle. Speculative tasks will only be started when all non-speculatively scheduled tasks have been dispatched, and there are idle slaves. At this point, the final updates in the current step are already being computed at the slaves.

A version is kept together with each particle update. This version is incremented when the particle does not select itself, something that *potentially* occurs during a resampling step. The version number determines if a speculative result needs to be discarded or if it valid for use.

When a particle update has finished, a post-processing procedure is run through a continuation on the future. This is not shown explicitly in the listings in Figures 6.5 to 6.7. If there is a version mismatch, the post-processing procedure is aborted. A speculative task to update the same particle in the next step is pushed onto the speculative task queue if there is no version mismatch. To disable speculation, it suffices to disable adding speculative tasks to this queue; the remainder of the algorithm does not change. Since updates can cause particles to run ahead, a version mismatch can occur by the time a speculative task, scheduled at some time in the past, would be launched. Therefore, an additional check is needed just before the speculative task is dispatched.

Speculative tasks are prioritized first by step and then, for updates in the same step, through a first-in, first-out order. Note that for models where execution time is shorter for relatively well chosen parameters, particles that finish updating first will typically be associated with a higher weight. Accordingly, the first-in, first-out order happens to start speculative computation on particles for which the result will likely be useful. Another approach is discussed in Section 6.8.

6.5.7 Fully Distributed Approach

The downside of the master-slave approach is that some portions of the algorithm execute sequentially. This, by itself, limits scalability in terms of Amdahl's law [Amd67; Pad11]. More importantly, the post-processing step of a *speculative* task that executed remotely can slow down the system. Therefore, it might be tempting to consider a fully distributed sampler where particles are assigned to processors, but it turns out that it has more performance limiting drawbacks than the simpler master-slave approach.

When particles are partitioned across processors, information still needs to be shared *globally* to calculate the ESS after each step. With speculative parallelism enabled, each processor will start evaluating speculatively immediately if the ESS is not available after the previous likelihood has been evaluated.

The processors that are executing speculatively will not contribute to calculating the ESS while they are busy with speculative execution. These processors will be committed to speculation while it is already known that the result will be useless. Compared to the master-slave approach, a processor will also be committed, but progress can be made nevertheless. One possible work-around is

```
procedure SpeculativeSMC(n, s)
    nextFrom = IdentityTable(n)
    for step = 0, . . . ,s do
        from = nextFrom
        UpdateWeights()
        if ResamplingNeeded(step) then
            nextFrom = CalculateResampling()
            ResetWeights()
        else
            nextFrom = IdentityTable(n)
        end if
        F_s = MoveParticles(step, from, nextFrom)        ▷ See Figure 6.6
        SetMinimumSpeculativeStep(step + 1)
        RunSchedulerUntil(F_s)                           ▷ See Figure 6.7
    end for
end procedure
```

Figure 6.5: Sequential Monte Carlo with speculative execution. To avoid distracting from the essentials of the algorithm, arguments like t and γ have been omitted. Instead of waiting for all updates in each step to finish, execution continues speculatively if possible as shown by Figure 6.7.

```
procedure MoveParticles(step, from, nextFrom)
    if step == 0 then
        return InitParticles()
    end if
    F = {}
    for i = 0, ..., n − 1 do
        if Contains(nextFrom, i) then
            if ValidTaskStarted(step, i) then
                F = F ∪ GetFuture(step, i)
            else
                F = F ∪ ScheduleUpdate(step, i)
            end if
        end if
    end for
    UpdateVersions()
    return WhenAll(F)
end procedure
```

Figure 6.6: Moving particle potentially reuses speculative results from subsequent steps. New computational tasks are only started for those particles on which computation has not yet been started.

```
procedure RunSchedulerUntil(F_s)
    while not Fulfilled(F_s) do
        while IdleSlaves() > 0 and HaveTasks() do
            DispatchScheduledTask()
        end while
        while IdleSlaves() > 0 and
                HaveSpeculativeTasks() do
            DispatchSpeculativeTask()
        ReceiveResult()
    end while
end procedure
```

Figure 6.7: Execution continues until all particles in the current step have been moved, as signified by fulfillment of F_s. Idle slaves are kept busy with speculative updates of particles in the subsequent steps.

to periodically check if the next ESS is available while evaluating speculatively. This by itself has two drawbacks. First, adding periodic checks within the model is intrusive. Second, as there is overhead involved with these checks, determining the optimal frequency at which these checks are to be executed requires tuning, making the sampler more difficult to use.

A sampler that allows ownership of particles to change over time might yield better results, but such an implementation is more complex and out of the scope of this chapter. Note that there is no fixed assignment between particles and processors in the master-slave architecture.

6.6 Results

This section discusses performance of the listings in Figures 6.5 to 6.7 with and without speculation for three use-cases with varying load balance. The prominent technology used to program multiple interconnected systems is MPI. Note that other message-based systems can also be used for the speculative sampler. Execution time of an implementation using Intel MPI with InfiniBand is measured on two different clusters.

The first cluster consists of homogeneous systems, each with two Intel Xeon CPU X5660 CPUs with 6 cores. Even though these CPUs are relatively old, they suffice for a study of scalability with and without speculation. Processes are affinitized to cores and slaves are allocated in a compact manner. For example, with 14 slaves, the master and 11 slaves run on the first system and the remaining 3 slaves run on the second system. The results from Sections 6.6.1 to 6.6.3 were obtained running on this cluster.

Each system in the second cluster has two Intel Xeon Skylake CPUs. These computational resources, provided by the Flemish Supercomputer Center, were used to collect the results in Section 6.6.4. With 18 physical cores per CPU, and 16 systems for a total of 576 cores, performance of the speculative sampler was evaluated at a much larger scale. Again, cores were allocated in a compact manner.

6.6.1 Accuracy Improvement From Renumbering

It is clear from Equation (6.4) that by renumbering particles, more speculative results can be used. Figure 6.8 shows the fraction of updates in which speculative results were useful for the second use-case described below. Since it is difficult to control the frequency of resampling by changing the threshold t a biased coin

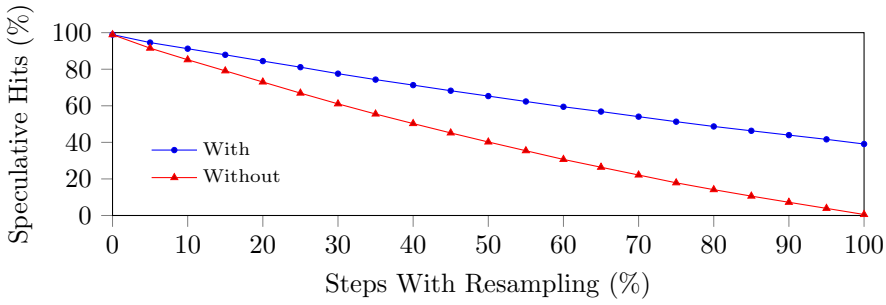


Figure 6.8: Effect of renumbering on the fraction of particles for which speculative execution is useful. The horizontal axis shows the fraction of steps in which resampling took place. A bigger fraction of speculative results are used when the renumbering scheme described in Section 6.5.4 is employed.

is flipped instead at the end of each step to determine if particles should be resampled. Without resampling, renumbering does not provide any benefits. As particles are resampled more often, renumbering becomes more important. Interestingly, when resampling occurs in *each* step, speculation without renumbering is almost useless while 40% of the updates still benefit from speculation if renumbering is enabled.

6.6.2 Distribution of Likelihood Evaluation Time

As discussed in Section 6.4, speculation becomes more important when execution time is more imbalanced. Three use-cases are considered below. Other configurations that have been omitted were tested as well. In *all* cases, the speculative sampler was showing improved performance.

The target distribution $\pi(\theta|X)$ of the first use-case is a simple three-dimensional multivariate normal distribution, with an added fixed-duration sleep of 50 milliseconds to increase the evaluation time. Little to no improvement is to be expected from speculative execution for this use-case. Some part of the sampler running on the master will execute in parallel with evaluations at the slaves, but this is minimal. Nevertheless, this use-case is included to show that speculation does not negativity impact performance even with a likelihood that has no imbalance in terms of the definitions from Section 6.4. More specifically, $\delta_i = \delta_{\max,s}$ for all s. Even in this perfectly balanced scenario, some variation is

to be expected at run-time due to system noise [Tsa+05] and varying network delays. The distribution for this use-case is shown on the right in Figures 6.9 to 6.11. Although not visible, there is some variation in execution time.

The second use-case is the Fitzhugh model given by Equation (5.1). The target density is given by Equation (6.5), where $\theta = [a, b, \phi]$. Equation (6.5) differs from Equation (5.2) in that Σ has been fixed.

$$\pi(\theta|y_i, t_i, \Sigma) = \prod_i \mathcal{N}(y_i|\psi(t_i, \theta), \Sigma) \times p(\theta) \tag{6.5}$$

The system of equations from Equation (5.1) is integrated using the CVODE solver from the SUNDIALS software package [Hin+05]. The initial values, $\psi(0, a, b, \phi)$, are fixed, and the remaining three parameters are estimated. As a prior, a three-dimensional normal distribution is placed around the origin. The distribution of evaluation times for this use-case is shown by the histogram on the right in Figure 6.10. Note that evaluations take on the order of microseconds.

In MCMC kernels, an accept-reject ratio is used to determine if a proposal is accepted or rejected [Mac03]. A number $u \sim \mathcal{U}(0, 1)$ is drawn and compared to this accept-reject ratio. If u falls below the accept-reject ratio, the proposal is accepted. The maximum threshold at which the proposal will be rejected can be calculated *before* evaluating the likelihood by drawing u first. For some models like the one described above, it can be shown that the likelihood is a decreasing function as more data is considered [Sol+12]. Therefore, evaluation can terminate if the threshold is reached. Not only does this reduce the total computation time, it also affects the distribution of evaluation durations. For the Fitzhugh model, it turns out that load becomes more imbalanced. Due to the added ability to terminating early on, around 20% of evaluations take less than 50 microseconds for the second use-case. There are two other distinct modes visible in the histogram, one at 90 microseconds, and another around 420 microseconds.

The third use-case uses more compute intensive non-linear tasks. It consists of simulating a Susceptible-Infected-Recovered (SIR) model used to study the spread of diseases [Hen+12]. The model consists of three compartments $S(t, a)$, $I(t, a)$, and $R(t, a)$ for each time step t and age group a for a population of size N. These are modeled as a set of Partial Differential Equations (PDEs). The likelihood is given by Equation (6.6), where $S(a)$, $I(a)$, and $R(a)$ denote steady state solutions and $\mathcal{B}$ denotes a Bernoulli distribution. Again, a multivariate normal prior is placed around the origin. The distribution for this use-case is

shown at the right in Figures 6.9 to 6.11.

$$p(y, N|S, I, R) = \prod_a \mathcal{B}\left(y_a \middle| N_a, \frac{S(a)}{S(a) + I(a) + R(a)}\right) \tag{6.6}$$

6.6.3 Relaxed Scalability Limit

The most important performance aspect is the execution time of the speculative sampler from the listings in Figures 6.5 to 6.7. For this, strong scaling is studied by comparing speedup with and without speculation on the left in Figures 6.9 to 6.11. The baseline for speedup in each comparison is the sequential case. A separate non-speculative run is used to measure the duration for each particle update. This information is then used to calculate the theoretical limit for each use-case as described in Section 6.4. These limits are shown in dotted lines together with the results in Figures 6.9 to 6.11.

In the first use-case, the perfectly balanced case, speculation provides no noticeable benefit when the particle count n is divisible by the processor count p. In this case, each slave finishes work at approximately the same time, and the wait time at the end of each step is minimal. However, when n is not divisible by p, the speculative sampler provides some improvement. For example, with $p = n - 1$, there will be one processor running two evaluations doubling execution time of each step, and consequently the whole run. This results in jumps in performance as shown in Figure 6.9. With speculation, similar delays are introduced, but only *once* at the end as the master is balancing load across the barriers.

In the second use-case, speculation raises the scalability limit from 15.11 to 50.54; this is a 3.34-fold increase shown in Figure 6.10. Initially, with $p = 1$, speculation provides no benefit, nor does it negatively affect performance. As p increases, speculation becomes more beneficial. The best performance with speculation is reached around $p = 71$ which corresponds to a speedup of 73% of the limit.

Another property of the speculative sampler can be observed with this use-case. As model evaluations take on the order of microseconds, the master does become the bottleneck here. With $p > 71$, performance starts to degrade slightly. As more processors are added, incorrect speculation becomes more common. The master has to process each task, be it normal or speculative. As the master is already the bottleneck, non-speculative tasks are delayed. In any case, speculation is still beneficial to use even if the optimal choice for p is not known.

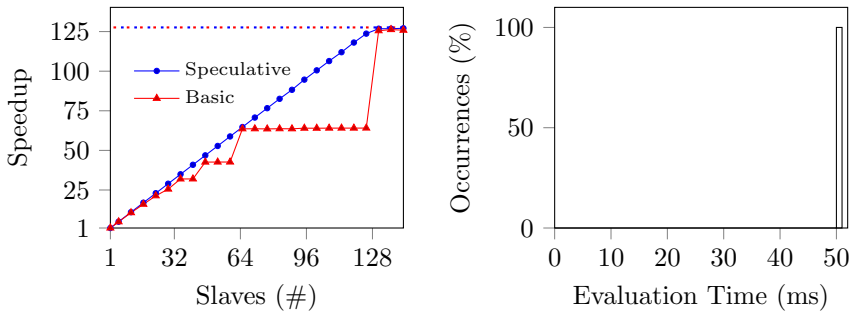


Figure 6.9: Scalability of the perfectly balanced use-case. The dotted line shows the scalability limit with and without speculation. The histogram of evaluation times is plotted below. For the first use-case, when the particle count n is divisible by the processor count p speculation provides no benefit.

The scalability limit is increased from 10.19 to 23.17 using speculation in the third use-case, shown by Figure 6.11. Compared to the limit, 99% of what is estimated to be possible is reached. This is shown on the right of Figures 6.9 to 6.11. Similar to the second use-case, the biggest improvements are made with many processors. Here, when the limit is actually reached, performance converges and no degradation is observed as evaluations take on the order of seconds.

6.6.4 Weak Scaling

Due to the limited amount of computation time available on the cluster provided by the Flemish Supercomputer Center, only weak scaling results with the third use-case were collected. Hence, particle count was dictated by the number of processors, i.e. $n = p$. The baseline for performance was the execution time of a single system running with 36 cores either with or without speculation. Execution time was averaged over multiple runs, each with a different random seed, until the results were stable. Performance is expressed in terms of parallel efficiency [Gra+03]. Note that parallel efficiency with and without scalability with 36 processors is 100% due to the choice of a different baseline. However, speculation provides a 2.15-fold reduction in execution time when comparing the two baselines. The results, shown in Figure 6.12, indicate that efficiency is better preserved as more processors are added when speculation is used.

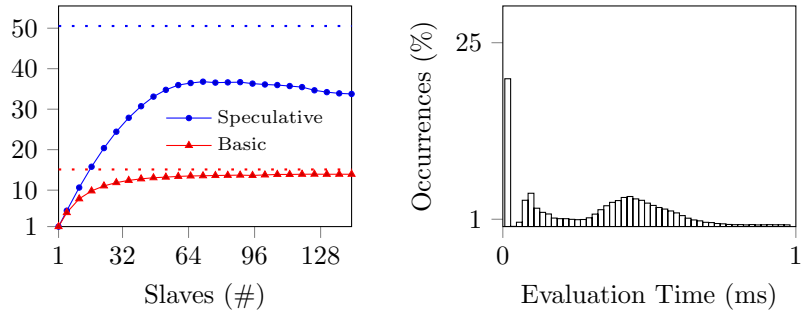


Figure 6.10: Scalability with significant amount of load imbalance caused by potentially terminating evaluation early. The dotted line shows the scalability limit with and without speculation. The histogram shows the distribution of evaluation times, but 4% of the data falling in $]1, 95.17]$ has been omitted. Nevertheless, these evaluations are important in the context of speculative execution. This use-case shows a decrease in running time, for any choice of p, by a factor of up to 2.72.

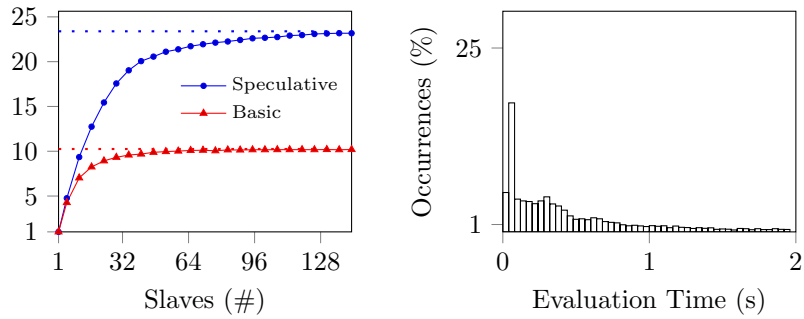


Figure 6.11: Scalability with a compute intensive use-case. The dotted line shows the scalability limit with and without speculation. For clarity, 10% of the data falling in $]2, 43.25]$ has been omitted from the histogram that shows the distribution of evaluation times. Note that these evaluations contribute to the efficacy of speculative execution. A decrease in running time, for any choice of p, by a factor of up to 2.28 is observed.

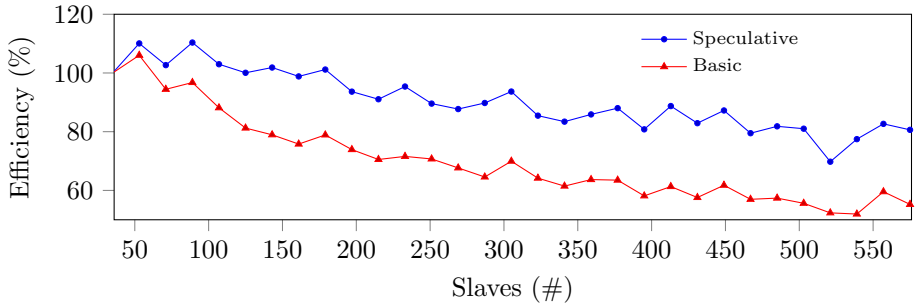


Figure 6.12: Weak scaling on a large scale cluster. The baseline is the sampler running with or without speculation on a single system with 36 cores. As more slaves are added, speculation allows the sampler to maintain higher parallel efficiency.

6.7 Conclusion

SMC samplers track the ESS of particles to determine when resampling is required at the end of each step. The ESS is derived from global information which requires *all* particles to be updated. Processors that finish updating particles quickly wait on the processor assigned with the most compute intensive updates. The fact that resampling does not occur at the end of each step is exploited. By assuming that particles survive through multiple steps, and by evaluating subsequent steps speculatively, scalability limits are relaxed. This is shown both theoretically and practically using three use-cases representing a variety of real-world scenarios including perfectly balanced use-cases and use-cases with a high degree of imbalance. To further improve the accuracy of speculative execution, a renumbering scheme is introduced that does not affect the statistical properties of the sampler.

At one extreme, with a single processor, no speculation is performed. As more processors are added, speculative execution becomes increasingly beneficial. The biggest performance improvements are for parallel systems where processor count and particle count are of the same order of magnitude. To evaluate the effectiveness of speculative execution, speedup is compared to a maximum derived from execution characteristics for each use-case.

There were no configurations where speculative sampling was slower. The

biggest improvement is a 2.72 fold reduction in execution time, but, at least theoretically, the improvements can be much higher depending on how imbalanced particle updates are. Given the execution time of all updates, it is possible to calculate the relaxed scalability limit. The results show that as long as the master processor is not the bottleneck, with speculative execution, performance approaches this limit.

6.8 Future Work

Currently, aside from prioritizing earlier steps, the order in which tasks complete also determines the order in which speculative tasks will be started. While this order is quite arbitrary, it happens to be beneficial for some models. A more robust approach is to use particle weights in the prioritization process. These express how likely the particle will survive the next step. Using this information as well should further improve the probability that a result will be used. Note that as long as there are more processors than particles available, no choice needs to be made since all particles are updated in parallel; when an update to a particle finishes, a task to update it again speculatively will be scheduled and dispatched next.

While in the listings from Figures 6.1 and 6.5 to 6.7 the tempering schedule γ is given as input to the algorithm, it can also be chosen dynamically. This way, the user does not need to tune this parameter. The tempering schedule is chosen in a manner that causes the ESS to be reduced by an amount falling within some bounds, i.e. between 99% and 90% of the previous ESS value. The tempering schedule itself is also constrained to fall within predefined bounds.

With such a dynamic tempering scheme, it is still possible to use speculation. This is accomplished by assuming that the previous jump in temperature will be used in the next step. Speculative results are correct if the previous jump delta causes the ESS to drop by a fraction that falls in the predefined bounds again. Since the ESS decreases slowly, another benefit is that resampling will be required less frequently.

As mentioned in Section 6.2, the strategy of speculation within a chain presented by Angelino et al. [Ang+14] can augment the speculation presented below. Since this would introduce a second type of speculative execution, it will be important to prioritize the speculative task that will provide the biggest improvement.

The second use-case considered in Section 6.6 shows some degraded performance due to processing incorrectly speculated tasks at the master. A direction

for future work would be to investigate how messages could be prioritized with the goal to advance non-speculative execution as fast as possible. In MPI, it is not possible to prioritize messages. Therefore, it might be worthwhile to explore exchanging messages between slaves to determine when to send messages to the master or a way to automatically find the optimal number of processors based on how congested the master processor is.

Although the load on the master strongly depends on the use-case, typically it still forms a single synchronization which can be a recipe for disaster in terms of parallelism. For long-running computations, there is usually no issue with the master-slave architecture unless many particles are used. This can be beneficial since it yields more accurate estimates. Future work will therefore consider a distributed sampler that would not suffer from such a bottleneck. As discussed in Section 6.5.7, a fixed assignment of particles to processors will not work well for speculation. Instead, allowing particles to migrate between processors could yield satisfactory results. The technique of work stealing [Din+07; Li+13] might be useful in this context. The added complexity of such an approach needs to be weighed with the benefits.

The implementation studied in Section 6.6 was written in C++ as the focus was on a detailed study of performance. It is the intent to release a Python implementation of the speculative sampler presented here since it is commonly used by scientists.

Part II

Front-end parallelization

Chapter 7

From Conditional Independence to Parallel Execution

7.1 Introduction

Now, attention turns to front-end parallelization, where the scientific models themselves are targeted. Due to their complexity, fitting on a single processor takes too long in practice. The focus here is on the parallelization of hierarchical models composed of multiple interconnected levels. Computational tasks required for each model evaluation are typically spread across relatively few layers. Consequently, this brings with it the opportunity to execute each level in parallel. While it might not be the optimal parallelization, it turns out that it works well in practice. It can even be used in conjunction with other methods that search for more fine-grained parallelism like the one presented in Chapter 8.

When the number of tasks exceeds the number of processors in a layer, some processors will inevitably execute more than one task. Depending on the variability of execution times between these tasks and the ratio between the number of tasks and processors, neglecting the scheduling problem can result in inefficient use of the underlying hardware. The parallelization approach is augmented with the well-known Longest Processing Time (LPT) static scheduling heuristic [CS18], where independent jobs with varying execution time are

scheduled on p identical processors.

The reachable efficiency is model-dependent; in general, the more compute-intensive tasks are available at each level of the hierarchy, the better performance will scale. Therefore, two different models are considered for evaluation: one containing only a few tasks and another with many more compute-intensive tasks. While parallelization adds overhead introduced by inter-processor communication, overall run time decreases in both cases.

The remainder of this chapter is structured as follows. Section 7.2 references related work. Section 7.3 discusses hierarchical models, their structure in the dataflow graph representation and the relationship with conditional independence. Section 7.4 describes the parallelization approach. Section 7.5 discusses performance results. Section 7.6 provides future work directions and concludes the chapter.

7.2 Related Work

The input to the optimization routines or sampling algorithms is a function that evaluates a model and returns a score that reflects the quality of the parameters. In this chapter, the input is a model description specified similarly to the probabilistic languages used in Turing [GXG18], Stan [Car+17] and WinBUGS [Lun+00].

The Turing system [GXG18] relies on explicit vectorization syntax to gain performance. The presented approach relies on the message passing model [KK07] for parallelism and vectorization is an extension that is left as future work.

Stan [Car+17] is a platform for statistical modeling and high-performance statistical computation. Recently, an extension to its modeling language has been proposed for parallelization [Web18], but use requires changing the model description. In contrast, the parallelization outlined below does not require the user to specify additional input signifying how computation should be scheduled on the hardware, but the downside is that it can be too aggressive causing performance to degrade in some cases.

Gibbs sampling [CG92] draws samples from the marginal target distribution by combining samples taken from conditional distributions. The concept of a graphical model is fundamental for Bayesian inference Using Gibbs Sampling (BUGS), implemented in WinBUGS [Lun+00]. MultiBUGS [Gou+17] has added parallel execution to WinBUGS by working directly on the graphical model from which conditionally independent parts are identified and scheduled to parallel processors only when deemed beneficial by a heuristic. Execution of Gibbs

Sampling requires synchronization between phases more closely resembling the BSP model. The difference with the work presented below is that the graphical model is used indirectly to detect parallel parts of the dataflow graph. Since the posterior is evaluated as a whole with less synchronization instead of being separated into smaller conditional densities, the applicability is not limited to Gibbs sampling. Another difference is that MultiBUGS ignores load imbalance by explicitly assuming that tasks have the same running time.

Even if the outlined approach is applied in a Gibbs setting, the parallelization within a single phase is different. For example, given a posterior $p(\theta|\mathcal{D})$, if $p(\theta_i|\ldots)$ and $p(\theta_j|\ldots)$ are assigned to one Gibbs phase, computation shared between these two conditional distributions can be executed only once even without blocking, a technique that affects convergence properties of Gibbs sampling [Yil12].

Nemeth et al. [Nem+20a] uses an Evolutionary Algorithm (EA) to parallelize the evaluation of probabilistic models by optimizing schedules through simulation of a parallel system with communication overhead. The downside is that searching for a schedule can become prohibitively slow, even though, at least in theory, the optimal schedule could be found. In contrast, using the graphical model is a simpler strategy as only tasks assigned to phases can be executed in parallel. However, it turns out that such an approach already yields well-performing schedules. Another difference is that the EA approach yields a static schedule in which both the execution order and the assignment of tasks to processors are fixed while the tasks that have been identified from the graphical model can be re-assigned depending on load imbalance changes.

An extensive survey for the well researched task graph scheduling problem is provided by Yu-Kwong et al. [KA99]. The main difference with conventional scheduling approaches is that the target domain is rather specific. The dataflow graph of a generative model specification always obeys a specific template. From this observation, a mapping can be formulated from which the parallelism is extracted directly.

7.3 Hierarchical Models and Conditional Independence

The main goal of this chapter is to show how model descriptions can be parallelized by relying on information from the graphical model. This section introduces the notion of a model description, its dataflow graph, and its graph-

```
for i in 1, ..., N
  φ_{i,1} ~ Lognormal(μ_2, σ_1)
  φ_{i,2} ~ Lognormal(μ_5, σ_2)
  p = h1(μ_1, μ_4, μ_5, φ_{i,1}, φ_{i,2}, pk_i)
  iv = [0.0, 0.0, 0.0, h2(μ)]
  ŷ = int_ode(t_i, 0, iv, dose_i, p)
  for j in 1, ..., n_i
    sdv = h3(ŷ_j)
    y_{i,j} ~ N(sdv, σ)
  end
end
```

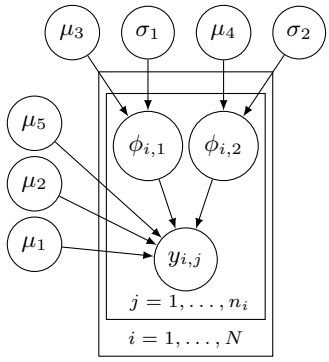


Figure 7.1: The Canagliflozin model description $\mathcal{M}$ on the left and its graphical model on the right. A PK/PD model, used to describe the compound concentration over time for N individuals in a population, is numerically integrated by `int_ode`. This model is defined elsewhere. The number of measurements for the i^{th} individual is given by n_i, and `h1`, `h2` and `h3` are side-effect free helper functions. In addition to $y_{i,j}$, the data $\mathcal{D}$ also contains the dosing regimen dose_i and PK paramters pk_i for each individual.

ical model. To distinguish between the structure of the two representations, "layers" refers to candidates for parallelism in the former and "levels" refers to the depth of variables in the latter.

From a Bayesian perspective [SS06], a model description defines a posterior $p(\theta|\mathcal{D})$. The numeric value of the posterior determines the quality of a chosen set of parameters θ while taking into account evidence $\mathcal{D}$. In what follows, θ_i denotes a component of the θ vector and $y_i \in \mathcal{D}$ denotes a data entry.

The description consists of likelihood expressions $y_i \sim p(.|\text{pa}(y_i))$ and prior expressions of the form $\theta_i \sim p(.|\text{pa}(\theta_i))$ where pa(.) is the set of random variables conditioned upon. These expressions will be generalized to $\gamma_i \sim p(.|\text{pa}(\gamma_i))$ for convenience. As an example, consider the model shown by Figure 7.1 on the left describing both PK and PD of a drug for type-2 diabetes treatment [Win+17].

To convert a model description into an executable function $f(\theta, \mathcal{D})$, prior and likelihood expressions are replaced by probability density function evaluations of the density $p(.|\ldots)$ at γ_i, denoted by a call to `pdf()` to which the distribution and the position are passed. Finally, the product of the resulting probability densities is returned while the remaining expressions are left untouched. The resulting function is then converted into a dataflow graph [Cul86; BJP91]. In contrast

to the typical controlflow style reasoning, a dataflow graph is an alternative model of computation where instead of executing operations on data, data flows through operators. This representation of computation lends itself well to parallelization [KK07]. This conversion is detailed in Chapter 8. A dataflow graph $G = (V, E)$ represents the set of computational tasks V and specifies how data flows between the tasks with edges E.

In general, the dataflow graph of a function $f(\theta, \mathcal{D})$ for a hierarchical model has the structure shown in Figure 7.2. The inputs θ and $\mathcal{D}$ are shown at the top and the product over densities is shown at the bottom. These are connected with the central portion of the graph, shown by dashed lines. Considering only the part with solid lines, the relationship with the graphical model is revealed. Each level depends on any of the previous levels through density evaluation nodes in V. In the example shown, the connections are less dense; for example, the first level is only connected to the second and fourth level and not to the third level. However, it is easy to see how the structure generalizes to any hierarchical model.

The model from Figure 7.1 is even less dense. Part of the first layer μ_3, σ_1, μ_4 and σ_2 are connected with the second layer with variables $\phi_{i,1}$ and $\phi_{i,2}$ and all variables in the second layer together with the remaining part of the first layer are connected with the third layer with variables $y_{i,j}$.

One simplification made here is that an edge in Figure 7.2 can represent a sequence of operations that transform random variables between layers or parts of layers like `h1`, `h2`, `h3` and `int_ode` in Figure 7.1. It is important to keep this in mind for the discussion in Section 7.4.

A graphical model $H = (R, F)$, is a representation of the conditional independence between variables. Figure 7.1 shows the graphical model on the right for the Canagliflozin model. For brevity, it is conventional to summarize similar variables with the plate notation by placing them into boxes with the range of iterated indices specified at the bottom [Gou+17]. For hierarchical models, H is a Directed Acyclic Graph (DAG), where the set of nodes R represents the random variables in the hierarchical model and their priors, and the edges $F \subseteq R \times R$ denote how the posterior can be factorized, i.e. $p(\theta|\mathcal{D}) \propto p(\theta, \mathcal{D}) = p(\gamma) = \prod_i p(\gamma_i|\text{pa}(\gamma_i))$. An edge from γ_j to γ_i is placed in F if $\gamma_j \in \text{pa}(\gamma_i)$.

To convert a dataflow graph G into a graphical model H, the nodes R and edges F need to be defined in terms of V and E. All nodes with input parameters in G, i.e. θ_i and y_i at the top of Figure 7.2, form R. The edges F are defined by the density evaluation nodes. By traversing the edges in E in the opposite direction starting at the node that provides the density input, the variables $\text{pa}(\gamma_i)$ can be found. Similarly, following the other input, γ_i can be

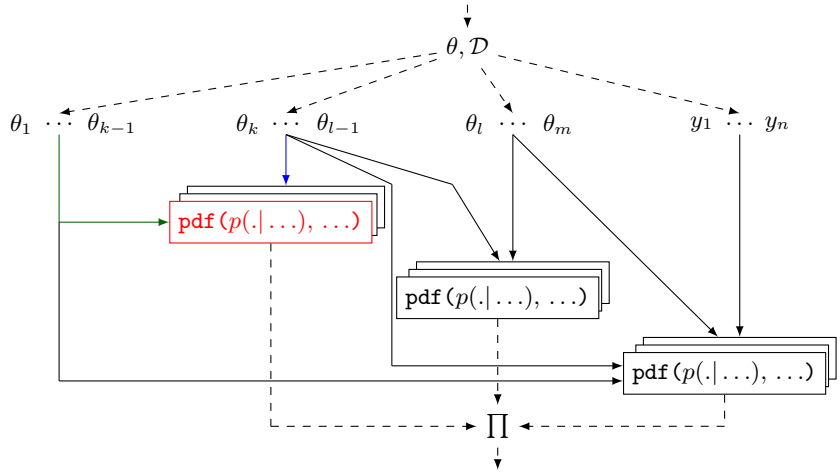


Figure 7.2: Simplification of the structure of the dataflow graph of a function $f(\theta, \mathcal{D})$ built from a hierarchical model with four layers the last of which is the data layer. The structure generalizes to any generative specification of a hierarchical models with arbitrary interconnected layers.

found. This mapping introduces a function m from R to V where $m(r)$ is either the corresponding probability evaluation node if it exists, or the input node of that variable. The set $\text{pa}(\gamma_i)$ and γ_i for the red node in Figure 7.2 can be found by following the green edges and the blue edges respectively.

7.4 Extracting Parallelism from the Graphical Model

Since the dataflow representation naturally exposes parallelism in a model it is possible to execute the dataflow graph by starting execution of each node when all its inputs become available. It is well known that scheduling such a computational DAG is hard to solve optimally, but many heuristics exist [Bła+07].

Scheduling each node separately is prohibitively expensive in practice due to the amount of overhead introduced on a contemporary system; not only is overhead introduced by starting a function compiled from the expression in a node, but also by tracking and storing its inputs and outputs. To reduce overhead, sets of nodes can be grouped into larger tasks and treated as a single unit at the cost of potentially reducing parallelism. It is possible to find satisfactory assignments of tasks to processors by considering the dataflow graph and the characteristics of the underlying parallel system directly [Nem+20a], but this search can be slow in practice, especially with graphs that have on the order of 10^4 nodes or more. Such graphs are not uncommon for typical PMX models. Assuming that the order of tasks is not fixed, there are n^p possible assignments to consider with p processors. Parallelization based on the graphical model is more tractable, although less detailed.

Since the posterior can be seen as a product of conditional densities as discussed in Section 7.3, the most basic approach is to create one task for each conditional density from the dataflow graph. This is accomplished by traversing the dataflow graph backwards from each node c that contains the expression $\texttt{pdf}(p(.|\ldots), \gamma_i)$ and selecting all reachable nodes, denoted by the set $\text{pred}(c)$. The procedure $\text{pred}(c)$ extends naturally to sets of random variables as well.

While this leads to an embarrassingly parallel solution since each task can be computed independently, the downside is that many nodes will be recomputed due to the similarities. More formally, for two density evaluation nodes c_1 and c_2, $\text{pred}(c_1) \cap \text{pred}(c_2) \neq \emptyset$. For example, suppose that one of the input parameters γ_i is first transformed to $g(\gamma_i)$ and there are multiple γ_j such that $\gamma_i \in \text{pa}(\gamma_j)$. Then, to compute each conditional density $p(\gamma_j|g(\gamma_i), \ldots)$, $g(\gamma_i)$ will need to be

recomputed for every conditional density, a situation that is undesirable if a significant portion of computation effort is spent in g.

Therefore, this chapter proposes to use the conditional independence to find similarities between conditional densities. Computationally, conditionally independence of two random variables γ_i and γ_j given γ_k means not only that $p(\gamma_i, \gamma_j|\gamma_k) = p(\gamma_i|\gamma_k) \cdot p(\gamma_j|\gamma_k)$ holds, but also that some of the tasks related to γ_i and γ_j are to be executed after some of the tasks related to γ_k. In addition, there might be some similarity between the computation related to γ_i and γ_j, but there will also be some differences. If this was not the case, then $p(\gamma_i|\gamma_k) = p(\gamma_j|\gamma_k)$.

Computational similarities can be captured by introducing deterministic variables β so that $p(\gamma_i|\beta, \gamma_k) = p(\gamma_i|\beta)$ and $p(\gamma_j|\beta, \gamma_k) = p(\gamma_j|\beta)$. Probabilistically, after marginalizing the deterministic variables β, Equation (7.1) holds.

$$p(\gamma_i, \gamma_j|\gamma_k) = p(\gamma_i|\beta) \cdot p(\gamma_j|\beta) \cdot p(\beta|\gamma_k) \tag{7.1}$$

Once $p(\beta|\gamma_k)$ has been computed, both $p(\gamma_i|\beta)$ and $p(\gamma_j|\beta)$ can be computed sharing as little information as possible. If no information is shared, they can be computed in parallel. Figure 7.3 shows how to accomplish this by processing random variables in the graphical model.

The end goal is to assign random variables to layers and to construct tasks from the variables in these layers. The assumption is that tasks constructed from a layer are independent. In the extreme, when a deterministic variable is introduced for each node in the dataflow graph, all tasks will be independent given their predecessors. Note however that Figure 7.3 introduces only a limited number of deterministic variables. Therefore, the predecessor relationship imposed by E will still need to be respected since there might still be some dependencies. The rationale behind this is that variables in the same layer tend to share computation through their connection with previous layers, on a layer by layer basis, while little or no computation is shared within a layer.

First, following the depth definition from MultiBUGS [Gou$^+$17], the level $d(r)$ is computed for $r \in R$. If $\text{pa}(r) = \emptyset$, then $d(r) = 1$. Otherwise, $d(r) = 1 + \max_{p \in \text{pa}(r)} d(p)$. In Figure 7.1 the depth is 1 for all μ and σ variables, 2 for all ϕ variables, and 3 for the data variables $y_{i,j}$. The levels of the random variables partition R into sets $R_1, \ldots, R_D$. Here, R_i contains all the random variables at level i.

It might seem that D layers can now be constructed, one for each set of variables R_i. However, this does not expose computational similarities present between layers. Instead, multiple layers will be introduced for each level i,

procedure ExtractLayers(G, H, m) ▷ G, H and m defined in Section 7.3
  Compute $d(r)$ for $r \in R$ as in MultiBUGS [Gou+17]
  for $i = 1, \ldots, D$ **do**
    $L_{i,i} = \{\{m(r)\} | r \in R_i\}$
    $P_i = \cup_{r \in R_i} \text{pa}(r)$ ▷ All direct parents of level i
    for $j = 1, \ldots, i-1$ **do**
      $P_{i,j} = P_i \cap R_j$ ▷ Direct parents in level $j < i$
      $L_{i,j} = \{\text{lcpred}(\text{ch}(p) \cap R_i) | p \in P_{i,j}\}$ ▷ Find computational similarities
    end for
  end for
  return $L_{1,1} \ldots, L_{D,D}$
end procedure

Figure 7.3: Extracting layers to construct parallel tasks.

represented by $L_{i,j}$. The elements of layer $L_{i,j}$ are sets of dataflow graph nodes.

For a level i, the deepest layer $L_{i,i}$ contains the dataflow graph node associated with the random variables $r \in R_i$ as singletons. Next, the directly reachable parents of the variables in R_i are collected in P_i. For each $j < i$, $|L_{i,j}| = |P_{i,j}|$ where $P_{i,j} \subseteq P_i$ are the direct parents on level j. Each element of $L_{i,j}$ is given by the last common predecessors of the children of $p \in P_{i,j}$ in R_j, denoted by $\text{lcpred}(\text{ch}(p) \cap R_j)$. For a set of nodes $S \subseteq R$, $\text{lcpred}(S)$ is computed by taking the nodes in $\cap_{c \in S} \text{pred}(m(c))$ for which edges lead to nodes in $\cup_{c \in S} \text{pred}(m(c)) \setminus \cap_{c \in S} \text{pred}(m(c))$. The expressions from the dataflow graph in each set in $L_{i,j}$ with $j < i$ constitute the computational similarities of random variables with depth i with respect to parents at depth j. These similarities correspond to deterministic variables like β.

For the model from Figure 7.1, $L_{1,1}$ contains singletons for the random variables at depth 1 like $\{m(\mu_1)\}$ and $\{m(\sigma_2)\}$. Analogously, $L_{2,2}$ and $L_{3,3}$ contains singletons for the random variables ϕ and the data entries y respectively. The direct parents of the variables with depth 3 are $\mu_1, \mu_2, \mu_5, \phi_{i,1}$ and $\phi_{i,2}$. Since $d(\phi_{i,1}) = d(\phi_{i,2}) = 2$ and $d(\mu_1) = d(\mu_2) = d(\mu_5) = 1$, two additional layers $L_{3,2}$ and $L_{3,1}$ will be introduced. Here, among others $\text{lcpred}(\text{ch}(\phi_{i,1}) \cap R_2)$ contains calls to `int_ode`, and $\text{lcpred}(\text{ch}(\phi_{i,1}) \cap R_2)$ contains calls to `h2`.

Finally, to turn the constructed layers $L_{i,j}$ into a partitioning of V, they are processed from shallowest to deepest while assigning all nodes in V. Each set $S' \in L_{i,j}$ is replaced by nodes in $\cup_{s \in S'} \text{pred}(s)$ except for those that have already

been assigned. The resulting sets form the final tasks.

While it might seem that annotating the for-loops in the model description like the one given in Figure 7.1 to specify that these should be parallelized is straightforward, the parallelization described here will not only automatically detect this, but it will also work for more arbitrarily interrelated models in which loops need not necessarily match the levels of the hierarchy.

The tasks within each layer can be scheduled to run in parallel. To maximize parallel efficiency [Gra+03], idle times need to be kept to a minimum. The only heuristic considered during performance evaluation is LPT [CS18] although other heuristics could be used as well. The focus is not so much on scheduling, but on presenting a mapping between two representations of a model to identify parallel parts.

Since subsequent posterior evaluations occur at similar positions in the parameter space, i.e. $\theta^t \approx \theta^{t+1}$, it turns out that the execution time for each task changes only gradually. For this reason, after running one iteration with tasks scheduled using a Round-Robin (RR) strategy, subsequent rounds can be scheduled with LPT using the execution time measured during evaluation of the previous candidate parameter θ.

7.5 Performance Evaluation

PMX models are key computational components leveraged for decision making during drug development. Here, only a limited amount of data is available [PB00]. The data includes the compound concentration in the blood of subjects, a costly measurement to make. In contrast to more classical models where all data is "independent and identically distributed", the data also specifies from which patient each measurement is taken creating a hierarchy as discussed above.

In this section, the performance of the proposed method is evaluated using two models from PMX. The first model, called the Nimotuzumab model, describes a humanized monoclonal antibody mAb, in patients with advanced breast cancer [Rod+15]. The second model is the Canagliflozin model used as the example in Section 7.3.

The structure is similar in both models; it consists of a population layer in which a set of patients that have taken part in the clinical trial are each modeled separately. However, it is important to note that the parallelization outlined in this chapter can be applied to models with more layers assuming that there are enough computationally intensive tasks in each layer.

The data for the Nimotuzumab model contains measurements of 12 patients

Table 7.1: The number of tasks per layer and the percentage of time in each layer for the two test models. Most of the time is spent in the fifth layer, where tasks that perform the numerical integration are concentrated. The final layer, with the most tasks, contains likelihood evaluations.

Model	**Metric**	$L_{1,1}$	$L_{2,1}$	$L_{2,2}$	$L_{3,1}$	$L_{3,2}$	$L_{3,3}$
Nimotuzumab	Tasks (#)	1	3	36	1	12	321
	Coverage (%)	0.00%	0.09%	1.43%	0.03%	90.35%	8.10%
Canagliflozin	Tasks (#)	1	2	2694	1	1144	5237
	Coverage (%)	0.00%	0.01%	0.03%	0.00%	99.90%	0.06%

resulting in limited amount of parallelization. On the other hand, the data for the Canagliflozin model consists of measurements of 1144 patients. For this model, it is important to note that some patients in the placebo group are not given the compound, while others are given the compound for either a shorter or longer period. Therefore, the time required to simulate PK and PD for each patient varies drastically [HR20]. For example, execution time of numerical integration varies up to 100x across patients for Canagliflozin.

If all expressions are compiled separately, respectively 6643 and 46261 tasks are created for the two models. The overhead of running these tasks separately, estimated by a run on a single system, slows down execution time by a few orders of magnitude. By applying the steps outlined in Section 7.4, the number of tasks drops to 375 and 9080 reducing task management overhead.

The distribution of tasks across layers is shown in Table 7.1. Most of the computation time, 90% and 99% respectively, is spent in the numerical integration of the PK and PD equations. The tasks that perform this integration are captured in a single layer. Both models compile to 5 layers with the most tasks in the last layer containing likelihood evaluations. Since likelihood evaluations in these models are lightweight, they also serve to demonstrate that the presented parallelization can be too aggressive as all layers are parallelized while manual parallelization would only assign more resources in the layer that captures numerical integration tasks.

The number of messages exchanged between processors depends on how tasks are scheduled, and varies at runtime for each evaluation when the scheduling step reassigns tasks. It is important to note that the LPT heuristic has a local view. Tasks in each layer are scheduled without considering the assignment of tasks in other layers.

Figure 7.4 compares performance when tasks in a phase are scheduled using

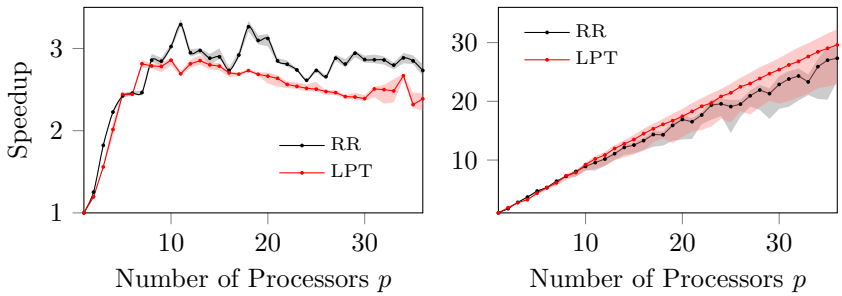


Figure 7.4: Mean scalability of the Nimotuzumab model on the left and the Canagliflozin model on the right with the shaded regions showing the uncertainty range for the 5^{th} and 95^{th} quantile of the speedup. The efficacy of the parallelization approach is model dependent, but performance improves for both models.

a RR strategy or by using the LPT heuristic on a single Haswell system with 2 Xeon E5-2699 v3 @ 2.30GHz CPUs, each with 18 physical cores for a total of 36 cores. The parallelization was implemented in the Julia programming language [Bez+17]. For the sake of stability of the results, frequency scaling was disabled. While other custom message passing implementations were also tested, the results are reported for an implementation relying on Intel MPI Version 2018 as it is widely available. Preparing and copying messages adds overhead, but note that since the results are for a single system, this could be avoided by using threads instead. Nevertheless, the mapping between the two representations with this overhead still shows promising performance scalability. It is also applicable to larger systems with a higher latency interconnect as long as the tasks are sufficiently compute intensive.

Since the outlined approach uses the message passing model for parallel execution, the more general term "processor" is used here [Gra+03]. The comparison is made in terms of the speedup achieved by running on p processors, denoted by S_p and given by the ratio between the execution time with one processor and p processors, i.e. T_1/T_p. As both T_1 and T_p are stochastic due to noise in the system [Lam09; HB15], execution time is measured 200 times for each choice of p to obtain stable results. Samples for T_1 are paired with T_p to generate samples for S_p. The 5^{th} and 95^{th} quantiles are shown to quantify the spread of S_p.

The limited number of patients in the Nimotuzumab model causes execution

time not to scale past approximately 10 processors. Note also that performance does not reach 10x with respect to baseline. Through profiling, it became apparent that this is not only due to the varying computational requirements between tasks associated with different patients, but also due to communication overhead. With the relatively small amount of available parallelism, this cannot be neglected, and it causes performance to degrade past 10 processors.

Note that the LPT heuristic results in slightly slower performance when compared to RR. This is due to the increase in the time spent communicating between some cores in some layers, an aspect not taken into account by the heuristic while in RR communication cost is spread more evenly.

Note also that initially, there is little to no difference between the two strategies. This is due to the two strategies behaving similarly when a few processors are used. As the number of processors increases, the performance of the two strategies diverges.

The Canagliflozin model scales better since there is a much larger opportunity for parallelization. Due to the amount of imbalance between patients, the LPT scheduling heuristic further improves performance by about 8%. Around 10% is lost due to overhead introduced by communication between processors and task management. This is verified by comparing to theoretically computed execution time where this overhead is ignored. Note that efficiency, computed by comparing actual scalability with linear scalability, stays above 90%. From this, it can be concluded that most of the available parallelism is exploited.

Since multiple processors are employed in each layer of the hierarchy, it only improves performance in layers with tasks that take a sufficient amount of computation to dwarf communication overhead. For layers with small tasks, the benefits of parallel execution will be outweighed by the overhead introduced by communication. In this case, overall performance will improve only when other compute-intensive layers make the overhead for layers with many small tasks negligible.

7.6 Conclusion and Future Work

This chapter introduces a novel way to parallelize evaluation of hierarchical models by observing that conditional independence in a graphical model representation can be mapped to the dataflow graph. The presented method has been shown to work for two characteristic models from PMX. Note that it is not limited to this domain. The efficacy of the model depends on the amount of parallelism inherent in the input models and the computational size of its

tasks. The results show that by using a simple well-known scheduling heuristic within each layer, performance can further improve in case execution time varies between tasks.

One drawback of the presented method is that *all* layers are parallelized. As long as there are enough layers with many compute-intensive tasks, the presented approach results in high utilization of parallel resources. However, the communication introduced in layers with small, but numerous tasks can degrade performance. Therefore, future work will explore how to disable parallelization selectively if communication overhead is high relative compared to the amount of computation.

The scheduling heuristic relies on the measured execution time of tasks during previous model evaluations. As long as the assumption holds that the execution time of tasks changes gradually while the encompassing sampling algorithm or optimization routine takes small steps in the parameter space, such an approach will suffice. There is additional overhead introduced by measuring and collecting the execution time of each task. Therefore, future work will study the trade-off of occasionally disabling these measurements while the scheduling heuristic uses less up-to-date measurements.

The current results were limited to a single system with communication between processors accomplished through memory. Another aspect that will be explored next is how to mitigate the latency of contemporary interconnects.

Finally, while the partitioning of nodes is used in this chapter to construct tasks, using the resulting assignments for initializing more complex heuristics as those used in other work [Nem+20a] to speed up convergence will be studied next.

Chapter 8

Automatic Parallelization with Varying Load Imbalance

8.1 Introduction

The parallelization based on the graphical model is dynamic. This allows a more favorable schedule to be selected and deployed at runtime with minimal overhead. Another upside is that any task that has all its inputs available can be launched. This can be seen as reordering tasks at runtime. However, the dynamic behavior is not for free as a data structure that represents tasks need to be stored in memory. Inputs and outputs need to be propagated and managed. In addition, it can be too crude and aggressive for some models and some parallel architectures where communication price is high.

Therefore, a more general and fine-grained approach like the one proposed here is desirable for some models. Again, a scientist provides the input in the form of a model description $\mathcal{M}$ lacking any notion of parallelism. Given a parallel system with p processors, the goal is to create p optimized processor specific procedures that, when run in parallel, evaluate a model. This static approach fixes the order of tasks assigned to each processor. Consequently, less overhead is introduced. The sequence of transformations outlined by Figure 8.1 accomplishes this.

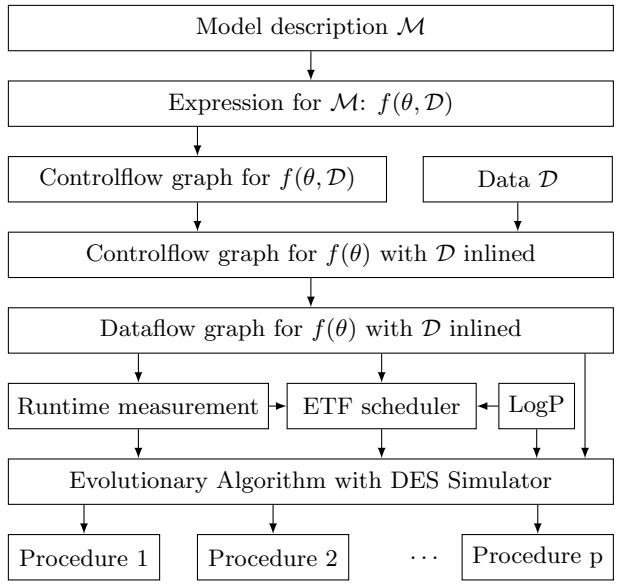


Figure 8.1: The steps required to parallelize model descriptions $\mathcal{M}$. By inlining the data $\mathcal{D}$ into the model evaluation, its parallel structure is revealed. Static schedules are combined into a more robust schedule from which processor specific procedures with embedded communication primitives, Procedure 1, ..., Procedure p, are derived.

The contribution of this chapter is not only to describe how the pieces in Figure 8.1 fit together, but to demonstrate that inlining data into the input model reveals its parallel structure and to show that an EA is effective at combining static schedules into more robust schedules that cope with the variation in the execution time of tasks.

The focus here is on PMX models used during drug development to study human-drug interaction dynamics. Typically, each participant in the drug trial is modeled by a set of ODEs [OF14]. Numerical integration of these equations can take a varying amount of time as θ changes [HR20].

While these models are structured in a particular way, it is important to note that the outlined approach generalizes to compute bound models from other domains with similar characteristics as well. In its current form, data bound models like those seen in Machine Learning (ML) cannot be handled easily due

to the large quantities of data to which these models are fit. To handle those models, parts of the model that repeat for each data element would need to be more compactly represented.

The remainder of this chapter is structured as follows. Section 8.2 references related work. Section 8.3 discusses the components of Figure 8.1 in more detail. Section 8.4 lists and discusses results. Section 8.5 concludes the chapter and provides future work directions.

8.2 Related Work

Probabilistic languages, like the one used to specify $\mathcal{M}$, are part of systems like Turing [GXG18], Stan [Car+17], nlmixr [Fid+19] and WinBUGS [Lun+00]. To leverage parallel resources, these languages either employ a parallel optimization method [Gou+17] or rely on parallel building blocks [GXG18]. Stan [Car+17] and nlmixr [Fid+19] are particularly well suited to fit nonlinear mixed-effects models, like the models studied below, through their powerful back-end algorithms. However, both packages limit parallelism to the back-end. Even with the recently proposed extension to the modeling language of Stan [Web18], parallel parts need to be explicitly specified. In contrast, as schedules are produced by an EA, $\mathcal{M}$ need not change. If parallelism on the target platform is not deemed beneficial for some parts of the model, that part will execute sequentially.

TensorFlow [Mar+15] is the most prominent framework for machine-learning today. Users specify a model as a dataflow graph directly, and its nodes are mapped across multiple systems within a computing cluster. The scheduling problem in TensorFlow using heuristics has been studied in the past [MML17], but those results are limited to ML models expressible in TensorFlow. Similarly, the parallelization of the models presented below relies on the dataflow representation, but the input models are compiled from an abstract description and static scheduling is used instead.

Explicitly inserting parallel constructs and tuning of computer programs can be extremely time-consuming and error-prone. Therefore, it is no surprise that auto-parallelizing compilers have been around since the earliest parallel supercomputers in the 1960s. These analyze programs to detect implicit parallelism and restructuring opportunities [Wol95; Pre75]. The methods described below have a similar purpose but focus solely on parallelizing probabilistic models instead of general programs. The specific structure of these models simplifies the dependency analysis to determine where the sequential ordering can be relaxed.

Chen [Che06] builds on the work of Hou et al. [HAR94] by using an EA

to optimize schedules under a LogP model, a well-known model for parallel computing discussed in more detail in Section 8.3, for deterministic task execution times. The evolutionary component in Figure 8.1 also optimizes schedules with the LogP model, but instead of augmenting the task graph with send and receive nodes to account for communication overhead, schedule execution time is estimated through Discrete Event Simulation (DES) with stochastic task execution times. The genetic operators are not those from Chen, but resemble those from Omara et al. [OA09] since these have been found to produce better schedules.

Kalinowski et al. [KKT00] considered Earliest Task First (ETF) scheduling under the LogP model where a schedule produced by the classical ETF heuristic is turned into a LogP feasible schedule by delaying tasks assigned to processors that are occupied by sends and receives. They only account for the latency, overhead and gap parameters. To more accurately assess schedules, including those produced by ETF, DES is used instead to also account for out of order receives and the cost of (de)serializing data exchanged between processes.

Zheng et al. [ZS13] discuss a Monte Carlo approach to schedule tasks with uncertain execution times. Their method consists of a generating phase to create static schedules with arbitrary heuristics based on sampled task execution times followed by a selecting phase to evaluate these schedules. The initial generation of the evolutionary approach outlined below similarly contains different static schedules, but these are evaluated by a DES simulation of the LogP model. In addition, the best schedules from subsequent generations are combinations of the schedules from the initial individuals.

Labyrinth is a method to compile control flow constructs to a single dataflow job with the goal to avoid incurring scheduling overhead in each iteration of the encompassing algorithm [Gév+18]. The compilation approach taken in Labyrinth consists of first compiling each basic block into a Static Single Assignment (SSA) and then into a dataflow graph. Each block is then linked using dynamic conditional edges. In contrast, all loops are compiled away when $\mathcal{D}$ is combined with $\mathcal{M}$ leading to static dataflow graphs, greatly simplifying parallelization. Although scheduling only takes place once, the characteristics of multiple iterations are taken into account.

Task runtime systems like those in implementations of version 3 of the OpenMP specification [LaG+11], StarPU [Thi18] and OmpSs [Fer+14] schedule tasks dynamically taking into account inter-dependencies and system load. While this dynamic approach allows to adapt to changes in load imbalance and reorder tasks at runtime, it also inherently adds overhead requiring tasks to be large enough to amortize the scheduling costs. For example, for StarPU, tasks should

take at least $100\mu s$ [Thi18] to avoid introducing too much overhead in relative terms. Since the smallest unit in the models is a single operation, it is on the order of nanoseconds. The communication overhead in the platform is still present, but concurrent operations that are too small to be scheduled on different processors will be assigned to the same processor, and no runtime overhead will be incurred. The performance of the resulting function for a system with a single processor system will be close, if not equal, to the performance of evaluating the model sequentially as all tasks are compiled together.

Many extensions to the standard LogP model have been proposed each targeting different network characteristics [HLR07]. For the parallelization presented here, the extension that separates send and receive overhead [Cul+96] is sufficient to predict execution time accurately. Depending on the target parallel system complexity and input model characteristics, other extensions of the LogP model will need to be considered.

8.3 Automatic Parallelization of Models

The goal is to transform an input model $\mathcal{M}$ into p processor specific procedures that when executed in parallel, compute the quality of a chosen set of parameters θ. An input model $\mathcal{M}$ consists of a set of statements each describing the distribution of unknowns. Statements follow the standard probabilistic syntax, e.g. $\theta_i \sim \mathcal{P}(.)$ for some distribution $\mathcal{P}$. Optionally, any number of procedures, which perform arbitrary computation in a side-effect free manner, like conditional constructs expressed as functions, numerically solving ODEs or performing other complex simulations, can be referenced in the model. Turning the code of $\mathcal{M}$ into an executable function corresponding to the log-posterior $\log p(\theta|\mathcal{D})$ that takes both a candidate solution θ and the data $\mathcal{D}$ as input, and outputs the sum of the log probability densities of all the parameters and all data entries, is accomplished by replacing all $\theta_i \sim \mathcal{P}(.)$ expressions by an evaluation of the log probability of $\mathcal{P}(.)$ at θ_i. All steps outlined below work on the level of expressions. Calls to functions, like those that evaluate the log probability or arithmetic operations, are treated as black boxes. The left of Figure 7.1 shows the code for an example model written with probabilistic syntax.

The model $\mathcal{M}$ describes a phenomenon abstractly, by specifying *how* data is distributed; it does not specify how many data elements there are and *what* the data is. However, since the data $\mathcal{D}$ for a particular problem is fixed for all evaluations of the posterior function, it can be combined with $\mathcal{M}$ creating a specialized version to reveal concurrent parts. This is accomplished by substituting

expressions with their known constants, where $\mathcal{D}$ is seen as a constant. For this step, a controlflow graph needs to be used [Aho+06].

In the dataflow model of computation, data flows through operations as opposed to the Von Neumann architecture where operations act on data. A program in this model is represented by a graph where nodes specify operations and edges specify how data flows through the operations. In general, when dataflow graph encode loops and conditional constructs, they are referred to as being dynamic and some nodes might either not be executed at all or multiple times. On the other hand, static dataflow graphs guarantee that each node is executed exactly once [Cul86]. The main benefit of this representation is that since it is clear on which data each operation depends, concurrency is exposed.

The approach to convert the controlflow graph into a static dataflow graph is based on Beck et al. [BJP91]. The central idea of their work is that each node in a controlflow graph can be seen as an operation that transforms the memory of a machine. Since memory is modeled by variables, a token is introduced for each variable that grants access to the memory associated with the variable. Load and store nodes are introduced that operate on memory, loading or storing a value of a variable. Each controlflow node is replaced by a set of nodes that will become the dataflow graph. Figure 8.2 shows the translation for an example expression $r = g(a_1, a_2, \ldots)$. Tokens for $a_1, a_2, \ldots$ are each connected to load nodes of which the loaded values are passed to a node that performs the operation g. The output of this node is fed into a store node that also takes the access token for r. Once the dataflow graph has been created, stores followed by loads are resolved, essentially parallelizing memory operations.

Loop dependency analysis [DK14] is avoided by inlining $\mathcal{D}$ into $\mathcal{M}$, typically resolving all loops. For the models considered in Section 8.4, the remaining loops and conditionals could be translated into so called dataflow graph switch nodes, but it turns out that sufficient parallelism is exposed by simply packing these into side-effect free functions. Since these are treated as black boxes, this avoids having to deal with dynamic dataflow simplifying execution.

To reduce the search space for the EA, the input graph should be as small as possible. Therefore, subsets of nodes in the dataflow graph that appear in the same order in any topological sort are merged. The rationale is that it is typically not beneficial to schedule these nodes over multiple processors due to the lack of available parallelism.

The next step is to find schedules $S = (O, A)$, where O is a permutation of nodes that respects the predecessor relationships of the dataflow graph and A is a vector that assigns nodes to processors. Processor specific procedures can be generated from a schedule as shown by the example in Figure 8.3 for a dataflow

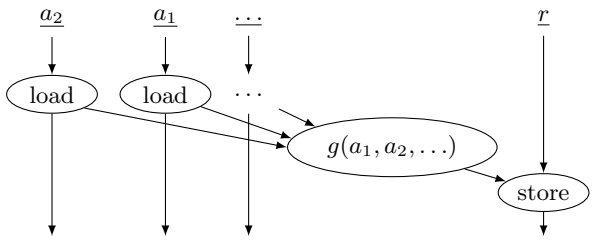
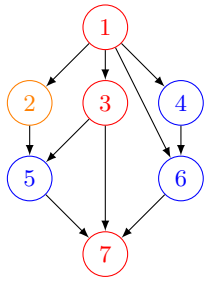


Figure 8.2: Dataflow graph nodes and edges for the expression $r = g(a_1, a_2, \ldots)$. Tokens, denoted by underlined variable names, for all variables $a_1, a_2, \ldots$ that are read in the expression, pass through load nodes first, and tokens to the variable r that holds the result of the operation passes through a store node. Tokens for all other variables, not shown here, do not connect to any of the nodes for this expression. This conversion was introduced by Beck et al. [BJP91].

graph of 7 nodes and a schedule with $p = 3$ processors. Since each receive operation blocks on its requested input, these are placed as late as possible in the procedures to avoid unnecessary blocks. If results are received out of order, they are transparently buffered by the receive operation. Note that the more classical approach of executing the task graph directly requires tracking all intermediary values adding a significant amount of overhead. This approach was explored but only a limited speedup was achieved. With processor specific procedures, if related tasks are assigned to the same processor, data flowing through edges is propagated by standard variables.

Another benefit of creating processor specific procedures is that an optimizing compiler can be leveraged. This means that even though the dataflow graph contains a node for each operation, these might be combined into a single runtime instruction. For example, $g_6(r_1, g_4(r_1))$ in Figure 8.3 might be compiled to a single instruction. The downside is that a schedule must be fixed beforehand. To maximize performance, schedules that minimize the total runtime are preferred, but the best schedule changes as task execution times change not only due to system noise [Lam09], but also by the choice of the model parameter θ. The latter is referred to as varying load imbalance.

The ETF heuristic is known to produce well-performing static schedules by iteratively assigning tasks to processors with the earliest start time in each scheduling step [KKT00]. For this, it requires the communication delay and the execution time of tasks. A version of the heuristic under the LogP model has




```
function procedure1(r0)
    r1 = g1(r0)
    send(r1, (3,2))
    r3 = g3(r1)
    send(r3, 2)
    r5 = recv(5)
    r6 = recv(6)
    r7 = g7(r5, r6)
    return r7
end
```

```
function procedure2()
    r1 = recv(1)
    r4 = g4(r1)
    r6 = g6(r1, r4)
    send(r6, 1)
    r3 = recv(3)
    r5 = g5(r3)
    send(r5, 1)
end
```

```
function procedure3()
    r1 = recv(1)
    r2 = g2(r1)
    send(r2, 2)
end
```

Figure 8.3: A dataflow graph and its three generated processor specific procedures based on a schedule $S = ((1, 4, 3, 2, 6, 5, 7), (1, 3, 1, 2, 2, 2, 1))$. The result for the operation g_i in the i^{th} node is stored in r_i, and by executing all three generated procedures in parallel, the final result r_7 can be computed for an input r_0.

been considered by Kalinowski et al. [KKT00]. They propose to either simply use a classically computed ETF schedule and adjust the make-span afterward or to insert reserved time slots while creating schedules. Instead, DES simulation with the LogP model is considered here. It not only more accurately takes into account the possibility of overlap in computation and communication, but also accounts for (de)serialization overhead and the time required to temporary buffer results received out of order. The only issue that remains is that the schedules assume fixed execution time of tasks.

To estimate the distribution of execution time for tasks, instructions to read and store timestamps are inserted before every g_i in the generated procedures. The encompassing optimization method is then run for a few iterations to collect samples of the execution time. An ETF schedule can then be generated from each sample. While this instrumentation inevitably adds overhead that influences the recorded timestamps, it turns out that the measured execution times are accurate enough for schedule creation. This is due to the noise being relatively small for the most compute intensive tasks and these have the most impact on the overall execution time.

Each ETF schedule considers only a single sample of execution time. An EA, initialized with ETF schedules, can combine and evolve these into a more robust schedule.

Each individual in the EA population represents a valid schedule. The population is not only initialized with ETF schedules, but also with schedules produced by topologically sorting the dataflow graph while randomly breaking ties and random assignments of tasks to processors.

Similarly to Omara et al. [OA09], the crossover operator chooses to either mix the order of tasks or the assignment to processors of two schedules and the mutation operator either reassigns a tasks to another process or changes the order of two tasks without violating dependencies between tasks.

The fitness of an individual is determined by the reciprocal of the execution time of the schedule while taking into account the variance in task execution time and the parameters of the target parallel system. It is prohibitively expensive to deploy and run candidate schedules on the target system. Not only does deployment take time, but more importantly as schedules are inherently parallel, only one can be evaluated at a time. Therefore, the fitness of individuals in the population is based on an estimate of the execution time on the target system. While there are many models of parallel computation [Zha+07], the variant of the LogP model that is used here describes a parallel system with sufficient detail to serve as a proxy for the real execution time.

The LogP model [Cul+93] is a well known computational model for distributed

memory parallel systems in which the most performance-critical factors are modeled. The system with P processors is modeled with messages exchanged between processors incurring a latency of L units of time and where o units of time is paid whenever a process receives or transmits a message. In this model, the bandwidth of a network interconnecting processors limits the delivery of messages to one every g units of time.

The extension of the LogP model that is used here is that from Culler et al. [Cul+96] where o is further divided into the send overhead o_s and the receive overhead o_r since a small change in o_r results in a significantly longer execution time for the input models considered in this chapter, while a change to o_s has a much smaller effect. Following the work of Kielman et al. [KBV00], the parameters L, o_s, o_r and g are measured. Note that o_s and o_r only account for overhead of the message passing implementation and not for (de)serialization overhead.

To estimate the overall parallel execution time of a schedule, it is simulated with DES for the LogP parameters of the target parallel system and the sampled task execution times. This is a simulation modeling technique where the state of the modeled system changes at discrete points in time [San07], is used. The simulation consists of entities that can be in different states including active or time delayed. In addition, a Future Event List (FEL) keeps track of when the time-delayed entities are to be woken up [SBS13]. The entities for a LogP simulation are the processors and the network, and the FEL contains processors waiting for a message, computing processors, or the network waiting to deliver a message. Using co-routines, the structure of the simulation code reflects the generated processor specific procedures. Only send, receive, and compute operations need to be replaced by simulated counterparts.

8.4 Results

The results below are of an implementation written in Julia [Bez+17] since it provides powerful meta-programming capabilities for inspecting and generating expressions, and a parser to read a model description, while simultaneously performing JIT compilation through LLVM [LA] to achieve high performance. The latter is important since it does not conflict with the performance goals of parallelization. The send and receive operations, like those depicted in Figure 8.3, are eventually forwarded to an implementation of the MPI standard.

Three pharmacometrics models are considered. These models mainly differ in the structural portion s. The first model is a change-point model [JL08]. The

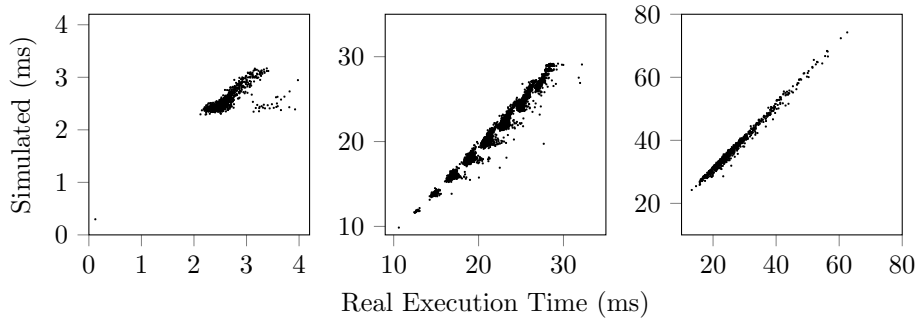


Figure 8.4: Scatter plots of the real runtime versus the simulated runtime for random schedules for a model that runs in less than $200\mu s$ on a single processor (left), the Nimotuzumab model (middle) and the WBC model (right). The estimates for runtime given by the LogP simulation are not perfect, i.e. R^2 is 0.49 (0.85 after removing outliers), 0.98 and 0.99 respectively. However, as the results in Figures 8.5 and 8.6 show, the correlation is sufficiently high for the purpose of finding better schedules using the DES simulation that, when deployed on a real parallel system, exhibit better performance.

second model from Rodriquez et al. [Rod+15] characterizes the pharmacokinetic behavior of Nimotuzumab, a humanized monoclonal antibody, in patients with advanced breast cancer. The third model given by Friberg et al. [Fri+02] describes the pharmacokinetics and pharmacodynamics of multiple drugs and WBC to study chemotherapy-induced myelosuppression.

The performance results are gathered on a Haswell system with 2 Xeon E5-2699 v3 @ 2.30GHz CPUs. Precautions have been taken to minimize noise in the system. Since software configurations play an important role in HPC workloads, these are noted here. CPU frequency scaling and SMT was disabled and processes were pinned. Intel MPI Version 2018 was used for message passing.

To assess if DES with LogP can serve as a proxy for the execution time of a schedule, Figure 8.4 shows the real and simulated execution time for random schedules with up to 36 processors. The coefficient of determination R^2 is 0.49, 0.98 and 0.99 respectively. Not all aspects of the underlying system are modeled. This includes operating system noise, garbage collection, and variation in communication delays. Due to the short duration of the tasks in the first model, these aspects introduce relatively large outliers that cause R^2 to decrease. After removing these, R^2 increases to 0.85. DES was able to predict the execution

time of all three examined models accurately. Note that the execution times of tasks was only measured once and the execution time for all schedules was predicted using this measurement. To measure the real execution time of a schedule, it was deployed without timing instrumentation. Hence, the results confirm that the overhead introduced by this instrumentation does not materially affect the measured execution times.

The change-point model has many short-running tasks; the total execution time on one processor for the whole model is less than $200\mu s$ and won't be selected for parallelization by an expert. Here, an EA should find a schedule that uses only a single processor as long as it explores such schedules. For the sake of brevity, this model is not considered further.

Scalability in terms of speedup S_p with p processors is reported in Figures 8.5 and 8.6 for the remaining two models. The speedup S_p is given by the ratio between the sequential and parallel execution time T_1/T_p of an optimization method that takes multiple steps. As discussed in Section 8.3, the initial population of the EA consists of schedules created using sampled execution times of tasks. This together with the stochastic aspect of EAs and inherent noise in the system causes execution time to vary. Therefore, the 90% uncertainty intervals of S_p, estimated from the distribution of T_1/T_p where T_p is collected from multiple runs, is shown as well.

The Nimotuzumab model is quite compute intensive as the ODE system that describes its dynamics [Rod$^+$15] needs to be solved multiple times during each model evaluation. Executing this model on a single processor takes on the order of tens of milliseconds. It includes the data for 13 patients and compiles to 4029 dataflow nodes and 7778 dataflow edges. Scalability for up to 20 processors is shown in Figure 8.5. First, note that the best ETF schedule, selected by comparing the simulated execution times of the input ETF schedules of the EA, shows more robust scaling than a randomly chosen ETF schedule. Next, note the best ETF schedule performs poorly in some cases. For example, there are runs where the best ETF schedule for $p = 5$ turned out to be as good as the schedule used for sequential execution. Finally, observe that the EA was not only able to produce more stable performing schedules, but it was also able to repair the bad schedule for $p = 5$.

Next, Figure 8.6 shows the speedup achieved with the WBC model with data for 45 patients. The model compiles to 903 and 1838 dataflow nodes and edges respectively. Again, an arbitrary ETF schedule performs poorly when using more than 5 processors. The shortsightedness of the ETF heuristic causes performance to drop in some cases since once a sub-optimal decision is made in a scheduling step, no backtracking is performed. The best ETF

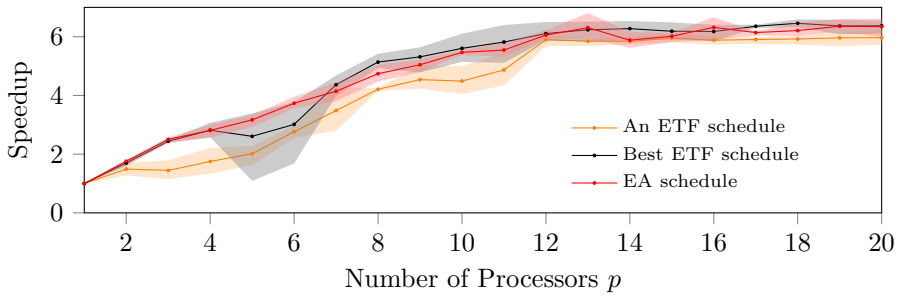


Figure 8.5: Scalability with 90% uncertainty intervals of an arbitrary ETF schedule, the best ETF schedule from the set of schedules used to seed the EA, and the best schedules found by the EA using DES simulation for the Nimotuzumab model. Speedup is limited due to the limited amount of parallelism in the model. The EA not only improved the subpar performing schedule with $p = 5$, but it also produced more stable performing schedules in general.

heuristic outperforms arbitrary schedules. The EA is sometimes able to improve performance stability, but more importantly, it is able to increase performance by 10% by combining ETF schedules.

Figures 8.5 and 8.6 support the claim that the correlation between the simulated and the real runtime need not be perfect. It only needs to be sufficiently high for the EA to be effective in practice. If the correlation would be too low, it would seem that better schedules are found from the perspective of the EA, but when those schedules are deployed, real execution time would vary greatly.

It is difficult to assess the speedup in absolute terms as it depends on the shape of the dataflow graph and the execution time of tasks. Since the scheduling problem is hard to solve in general, the maximum theoretical speedup in function of p could not be reported due to the size of the dataflow graphs. Therefore, the ratio between sequential execution time and the critical path length of the graphs is reported instead. The latter would be the execution time with infinite processors and no communication delay. For the Nimotuzumab model and the WBC model, this was respectively 9.27 and 8.28. Hence, with the best schedules, around 33% parallel overhead is still observable in the Nimotuzumab model when compared to this arguably unrealistic reference, while almost all parallelism is exploited in the WBC model.

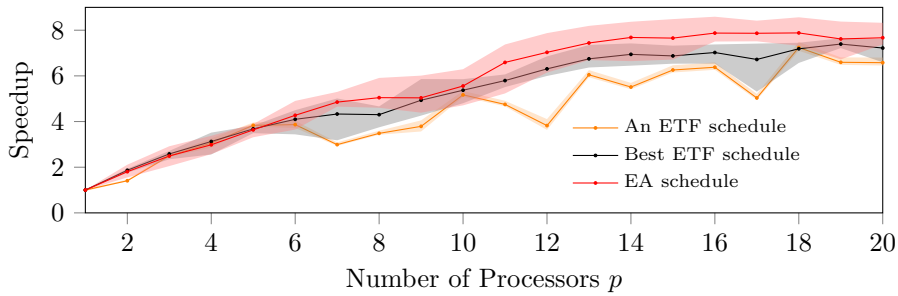


Figure 8.6: Scalability of random schedules with 90% uncertainty intervals and those found by the EA using DES simulation for the WBC model. The EA was able to improve upon schedules produced by ETF up to 25% by combining and evolving them.

8.5 Conclusion and Future Work

This chapter outlines an approach to automatically parallelize probabilistic models. The two key enabling observations are that the parallel structure of probabilistic model is revealed when the data is combined with the model and that the static schedules can be combined into more robust schedules that are able to deal not only with load imbalance, but also as load imbalance varies with the model parameter θ.

By modeling the execution time of the message passing system using the LogP model, it turns out that the execution time of a schedule is predictable with sufficient accuracy so that it can be used by an EA to find better schedules.

From static schedules, processor specific procedures with embedded communication primitives can be created to avoid the overhead typically present in many-task systems at the cost of fixing the execution order. It turns out that the performance benefits far outweigh this cost. Around 66% and 99% all of the estimated parallelism disregarding communication overhead is exploited in two evaluation models respectively.

Using parallel resources in both the back-end and front-end simultaneously could further improve performance. The trade-off between resource allocation in these two levels seems an interesting future research direction. Alternatively, embedding the task graph of the model into the optimization method task graph and searching for schedules for the two simultaneously could be another valid

approach.

Currently, a fixed number of steps were taken in the EA. Since a fixed computational time budget is typically known upfront to run the optimization method, future work will look into using EA convergence rate predictions to trade-off quantified speedup gained per unit time spent in the EA. This can then help decide if more evolutionary steps should be taken or to stop and use the current best schedule to start the actual run of the optimization method.

The basic LogP model has already been extended to heterogeneous networks. Incorporating that into the DES simulation model in the target parallel system is straightforward and would allow to also take into account the hierarchical nature of contemporary parallel systems while generating schedules.

The presented work could be extended to incorporate offloading computations to hardware accelerators like Graphics Processing Units (GPUs) by making task execution times and communication overhead resource specific. Computations will only be assigned to accelerators when the overhead is small.

Finally, the focus has been on testing how well models run on a single system although with a relatively high core count when compared to other contemporary systems. The next step is to scale up further with larger models and to consider the complexities of communicating across a network with different characteristics.

Chapter 9

Conclusions and Future Work

This thesis tackles computational aspects of PMX models on large parallel systems. The key challenges with these models are that the amount of computation is fixed and that they exhibits a high degree of variance in execution time. When computational methods on these models are mapped to larger parallel systems, they become sensitive to the latency of the interconnect.

These methods are classified into two components. The front-end consists of model evaluations and the samplers execute in the back-end. Techniques and improvements specific to either component are proposed.

First, an approximation that exploits the repeated dosing structure of the PMX models is proposed resulting in an up to 70x improvement made possible by numerically applying the method of averaging. Two techniques are shown to correct the error in the approximation. This leads to a self-adjusting mechanism that selects the error threshold by trading off accuracy and speed.

Next, the concept of "introducing useful computations during otherwise stalled cycles" is introduced with the goal to increase operational intensity. This idea is demonstrated in the context of Bayesian logistic regression with two samplers yielding improvements of up to 5x.

Third, a parallel version of the affine invariant sampler is mapped to larger parallel systems by observing that decisions based on a PRNG stream are completely predictable. Here, improvements range from 10x to 20x depending on the target. A scheduler that prioritizes some of the tasks at runtime is included

to further improve performance by up to 15%.

Fourth, scalability limits caused by load imbalance are relaxed with speculative execution in a SMC sampler. Reordering particle PRNG streams is shown to increase predictive accuracy of speculative execution. The sampler is first evaluated with up to 128 processors on three use-cases with differing load balance characteristics. A speedup of up to 2.72x is achieved. The same speculative sampler is also evaluated with 576 processors in a weak scaling setting showing that a higher efficiency can be maintained.

In the second part, two automated parallelization methods are presented to allocate resources in the front-end. The first is based on mapping conditional independence to parallel execution and employing an existing scheduling heuristic to schedule layers in the dataflow graph. This has been shown to perform well on larger models where parallel efficiency remains above 90%.

The second parallelization method combines static schedules produced by a scheduling heuristic through an EA to create novel schedules that are better able to deal with varying execution time. Again, the reduction in execution time is model dependent, but results show that almost all the available parallelism can be exploited with this technique resulting in a speedup of 6 and 8 for two test models.

A significant speedup can be achieved with all of the presented ideas. Putting this into perspective, runs that take up to weeks with contemporary methods can now be shortened to days or hours, aiding in the agility of the drug development process.

As the scalability results show, there are more opportunities in the back-end than in the front-end due to the inherent structure of the models. On the aspect of resource allocation, the current recommendation is to allocate processors hierarchically between the front-end and back-end following the hierarchical organization within the parallel system. More specifically, all cores within a CPU should cooperate on evaluating a model at a candidate parameter and CPUs should cooperate on the level of the back-end. This is advisable since a single model evaluation requires many messages and communicating through shared memory avoids the (de)serialization overhead. A more detailed allocation of resources could be an interesting avenue to consider.

In Chapters 7 and 8, speedup was reported not only in terms of a single value, but the uncertainty was also quantified. Execution time on contemporary systems is stochastic, and for that reason, the uncertainty of all the reported results should also be quantified. Research suggests that the distribution of execution time on parallel systems has a fat tail [HB15]. Removal of the synchronization barriers in the samplers in Chapters 5 and 6 not only reduces the variance in

overall execution time, but makes the fat tail thinner. It would be interesting to investigate by how much the variance is reduced.

All the presented samplers and parallelization were implemented specifically to showcase the ideas and achievable performance. A useful engineering project would involve developing a single software package that leverages all the findings. As noted earlier, it is important to have a variety of tools available since there is no panacea that outperforms all the others. With this in mind, future work should consider how to automatically select the best tool given a task.

The dataflow graph representation of the model enables powerful analysis techniques, some of which have been explored in the second part. Samplers like HMC require the gradient of the model and it is relatively easy to extract these from the model description without any input from the user although the gradients were defined manually for the results in Chapter 3. Another idea that can transparently be enabled in this representation is to abort computation early on. If a model evaluation is only used in an accept-reject ratio and it contains a product of likelihood evaluations, it is possible to determine if a sample will be rejected after computing a subset of the likelihood evaluations [Sol$^+$12; Hab$^+$18]. The benefit of the dataflow graph representation is that it is relatively easy to transparently insert code to abort computation. Furthermore, likelihoods evaluated in parallel could be aborted even sooner yielding super-linear speedup as discussed in Section 2.4.

It is reasonable to assume that the model will not be created in one go. Instead, multiple iterations will be required to fine-tune the details. Since the statistical parameters of the samplers still require tuning, a possible improvement to consider in this context is to reuse tuning settings at least partially to avoid having to spend too much time tuning each run from scratch. In the same way, a modeler will make small adjustments to the model after studying the results. A research question here is not only how the results change in function of a small change in the model, but also how computation changes. It could require less resources to simply "update" the previous results.

Except for the approximation presented in Chapter 3 and the reordering step in Chapter 6, all the techniques are transparent in that the output of the algorithms is unaltered, but this comes at a cost. As illustrated by Chapter 3, permitting a small error can yield great performance improvements. In essence, another easier problem is solved. This idea could also be explored in the context of MCMC samplers where knowingly introducing a negligible error could reduce the computational burden.

Another constraint adopted in this thesis is that given an algorithmic configuration, the output should not change when run on a different parallel system.

For example in Chapter 5, the number of walkers can be set regardless of the number of processors. The assignment of walkers to processors changes transparently. The motivation behind this is that it aids not only in reproducibility, an important property for the strictly regulated drug development process, but also in debuggability.

Similarly, the output remains the same even when some processors are delayed. Here, a relaxed reduction operation that completes with partial information could still result in useful output while reducing stalls.

These approximations could be useful during the initial model development phase, although the right statistical precautions need to be taken, since a slight error can yield very different results when the models are not well conditioned. The final "real" run can still rely on more precise proven methods.

As noted in Chapter 1, modelers currently switch between different projects to avoid having to wait for results. Once a modeler has completed the structure, it is scheduled for computation on lab machines shared across different groups within the same organization. Since each of these run in isolation, it is reasonable to assume that some computations will be repeated. This motivates the idea that it could be possible to speed up computation by sharing information across different projects either in an online or offline fashion. For example, numeric integration algorithms search for a step size that obeys tolerance requirements. It could be possible to speed up the search by initializing it with knowledge from prior runs.

An important assumption in this thesis is that all resources are identical and in most cases, latency was assumed to be constant between all processors. A direction that should be explored as well is to utilize more heterogeneous systems where a subset of the available hardware resources are better suited for a subset of the tasks.

Appendix A

Dutch Summary

Farmaceutische bedrijven moeten vroeg investeren in het ontwikkelen van een nieuwe medicijn. Initieel worden er verschillende kandidaat middelen geïdentificeerd en vervolgens gaan deze door verschillende fasen vooraleer ze beschikbaar gesteld worden aan een breder publiek. Pas als ze de laatste fase halen, zal er winst kunnen geboekt worden, maar veel van de kandidaten worden vroeg in het ontwikkelingsproces geschrapt. De voornaamste reden hiervoor is dat het blijkt dat de middelen onvoldoende werkzaam zijn met als resultaat dat niet alleen de investering, maar ook tijd en mankracht hierbij verloren gaat. Bijgevolg zijn farmaceutische bedrijven erg geïnteresseerd om het proces te optimaliseren door bijvoorbeeld sneller de meest doeltreffende middelen te identificeren. Een veelbelovende manier om dit te doen is met "in silico" technieken. Hierbij is Pharmacometrics een domein dat zowel wiskundige als statistische modellen gebruikt om de interactie tussen een individu en een middel te verklaren. De berekeningen voor deze modellen zijn rekenintensief. Het kan weken duren vooraleer resultaten beschikbaar zijn en hierdoor kunnen sommige beslissingen in het ontwikkelingsproces van de medicijn pas later genomen worden. Omwille van de duur van de berekeningen schakelen wetenschappers tussen verschillende projecten om zo het effect van de wachttijd te minimaliseren ten koste van extra mentale belasting. Deze thesis introduceert een aantal methodes en technieken om de berekening te versnellen door gebruik te maken van de rekencapaciteit van recente computersystemen om het werk van deze wetenschappers te vergemakkelijken.

Het aantal parallelle rekeneenheden in recente systemen is in recente jaren sterk toegenomen om de intrinsieke limieten van impliciete parallellisme en

geheugensnelheid te omzeilen. Het nadeel hiervan is dat het gebruik van deze expliciete vorm van parallellisme extra kennis vraagt. De onderzoekers missen vaak de nodige kennis uit computerwetenschappen. Daarom worden bestaande softwarepakketten gebruikt die de complexiteit van de onderliggende hardware abstraheren. Helaas blijkt dat de meest bekende softwarepakketten die van toepassing zijn voor Pharmacometrics beperkt gebruik maken van parallellisme.

In deze thesis worden parallelle methoden in de context van computationele modellering onderverdeeld in twee klassen. In eerste instantie kan er parallel gewerkt worden in de "front-end", waar modelevaluaties zelf geparallelliseerd worden. Software voor modelevaluaties kan gezien worden als een klassiek programma dat als invoer een aantal numerieke parameters krijgt en als uitvoer de kwaliteit, uitgedrukt als een getal, voor deze parameters geeft.

Om parameters te zoeken die zorgen dat het model beter aanleunt bij de data worden er optimalisatie methodes of Markov Chain Monte Carlo samplers gebruikt. Deze worden geclassificeerd in de "back-end". Er worden enkel samplers beschouwd in deze thesis omdat deze het meest toepasselijke zijn voor Pharmacometrics. Samplers laten toe niet enkel één parameter te zoeken die zorgt dat een model zo goed mogelijk aanleunt bij de data, maar ook om de onzekerheid in de parameters te kwantificeren. De oplossing is dus niet één vector, maar een hele distributie benaderd door samples.

De ontwikkelde methodes en technieken gaan uit van eigenschappen van modelevaluaties die zeer typisch zijn voor Pharmacometrics. Dit wilt niet zeggen dat de methoden beperkt toepasselijk zijn in andere domeinen, maar wel dat ze de meeste snelheidswinst zullen geven als de modellen gelijkaardige eigenschappen hebben. De eerste eigenschap is dat de hoeveelheid data beperkt is en niet mag aangepast worden. Er nemen tijdens de eerste ontwikkelingsfases minder patiënten en gezonde vrijwilligers deel aan de studie terwijl er in de laatste fases juist wel meer data van meer individuen wordt verzameld. Bijgevolg is het niet mogelijk om parallelle rekeneenheden bezig te houden door het model zwaarder te maken. De tweede eigenschap is dat de tijd die nodig is voor modelevaluaties afhankelijk is van de keuze van de parameters. Als er in parallel meer evaluaties worden uitgevoerd, dan zal de traagste evaluatie de totale uitvoertijd bepalen.

Bestaande parallelle berekeningsmethoden maken beperkt gebruik van de onderliggende hardware. De efficiëntie ligt nog lager op grotere parallelle systemen waarbij het uitwisselen van berichten tussen rekeneenheden meer tijd in beslag neemt. Het zijn juist deze grotere systemen die de meeste snelheidswinst kunnen bieden.

De thesis beschouwt eerst de algemene structuur van farmaceutische modellen en stelt een methode voor die de numerieke simulatie van de differentiaalverge-

lijkingen tot 70x kan versnellen. Deze vergelijkingen, die de interactie met het medicijn beschrijven, hebben vaak een herhaaldelijke vorm, dit omdat een kandidaat een medicijn gewoonlijk meerdere keren toegediend krijgt over een langere periode. Er wordt getoond hoe dit kan uitgebuit worden door een foutmarge toe te staan in de berekeningen. Om bruikbaarheid te bevorderen, wordt deze methode uitgebreid met twee manieren om de foutmarge zonder verdere invoer van de gebruiker te bepalen.

Ten tweede worden Markov Chain Monte Carlo samplers beschouwd die een lage operationele intensiteit hebben omwille van de hoeveelheid data dat er geraadpleegd wordt tijdens het evalueren van het model. Hierbij worden nuttige berekeningen geïntroduceerd op het moment dat er anders zou gewacht worden. Hoewel de hoeveelheid data eerder beperkt is in de context van Pharmacometrics, blijven de methoden toepasselijk omdat ze ook eventueel gebruikt kunnen worden om andere vormen van vertraging te verbergen.

Ten derde wordt een gedecentraliseerde versie van een bekende parallelle affiene invariante sampler beschreven. De populariteit van deze sampler is te danken aan de eenvoud in het gebruik ervan. Om deze sampler naar een gedecentraliseerde omgeving te brengen wordt er gebruik gemaakt van de observatie dat keuzes die gemaakt worden aan de hand van getallen geproduceerd met een pseudotoevalsgenerator voorspelbaar zijn. Dit resulteert in een sampler die tot 20x sneller is dan de oorspronkelijke parallelle sampler. Door minder berichten uit te wisselen en door een synchronisatiepunt te verwijderen wordt deze snelheidswinst mogelijk gemaakt.

Ten vierde worden Sequentiële Monte Carlo sampler beschouwd. Deze zijn erg toepasselijk voor Pharmacometrics aangezien de niet-lineaire modellen in dit domein zorgen voor distributies met meerdere pieken. Deze sampler beschouwt verschillende parameters in parallel en combineert de resulterende kwaliteitsmaten in elke stap. Omdat de uitvoertijd nou samenhangt met de keuze van parameters voor de Pharmacometrics modellen wordt er langdurig gewacht totdat alle berekeningen op alle parameters voltooid zijn. Het blijkt dat de keuzes die afhankelijk zijn van deze berekeningen accuraat voorspelbaar zijn. Door speculatief verder te rekenen kunnen wachttijden beperkt worden. Indien er later tijdens het berekeningsproces wordt bepaald dat een speculatieve berekening fout blijkt te zijn, dan worden deze ongedaan gemaakt. Omdat dit zelden het geval is, kan deze speculatieve sampler tot 2.72x sneller samples verzamelen dan de standaard sampler.

Ten vijfde en zesde worden er twee automatische parallellisaties geïntroduceerd die toepasselijk zijn in de "front-end". De eerste parallellisatie bestaat uit het extraheren van de conditionele onafhankelijkheidsrelaties uit de dataflow graaf.

Deze relaties kunnen dan gebruikt worden om parallelle taken te identificeren die vervolgens toegekend kunnen worden aan processoren aan de hand van hun geschatte uitvoertijd met een bekende heuristiek. Voor een groot model liggen de versnellingen die hiermee gehaald worden dicht bij het theoretisch maximum op een systeem met twee Xeon CPU E5-2699 processoren, elk met 18 cores.

De tweede parallellisatie vertrekt van een meer algemeen uitgangspunt met als doel om meer parallellisme uit te buiten. De uitvoer van deze tweede methode is een verzameling functies. Als deze uitgevoerd worden in parallel, dan wordt de kwaliteit van één parameter bepaald. Deze functies bevatten naast de berekeningen ook de communicatie primitieven. De uitvoertijd van alle delen van de dataflow graaf wordt eerst gemeten. Deze metingen worden gebruikt om een schatting te maken van de totale uitvoertijd van het model aan de hand van discrete gebeurtenis simulatie. Een schema dat de toekenning van taken tot processoren en de volgorde van de taken bepaald wordt als invoer gegeven voor deze simulatie. Aangezien de uitvoertijd afhankelijk is van de parameter, wordt het meerdere keren met verschillende parameters gemeten om zo de distributie van uitvoertijd te schatten. Voor elke meting wordt er een schema gegenereerd aan de hand van een bestaande heuristiek. Vervolgens wordt er gedemonstreerd dat een evolutionair algoritme deze schemas kan combineren tot een meer robuust schema dat beter werkt dan die gegeneerd werden door de heuristiek. Er wordt getoond dat dit statisch maar robuust schema zo goed als alle parallellisme in de modellen uitgebuit.

Appendix B

Publications

This thesis includes the contents of the following scientific contributions:

[Nem+17a] Balazs Nemeth et al. "Improving Operational Intensity in Data Bound Markov Chain Monte Carlo". In: *Procedia Computer Science* 108 (2017), pp. 2348–2352. DOI: 10.1016/j.procs.2017.05.024

[Nem+17b] Balazs Nemeth et al. "Distributed Affine-Invariant MCMC Sampler". In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Sept. 2017. DOI: 10.1109/cluster.2017.68

[Nem+18a] Balazs Nemeth et al. "Relaxing Scalability Limits with Speculative Parallelism in Sequential Monte Carlo". In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Sept. 2018. DOI: 10.1109/cluster.2018.00065

[Nem+19] Balazs Nemeth et al. "Approximate Repeated Administration Models for Pharmacometrics". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2019, pp. 628–641. DOI: 10.1007/978-3-030-22734-0_46

[Nem+20a] Balazs Nemeth et al. "Automatic Parallelization of Probabilistic Models with Varying Load Imbalance". In: *International Symposium on Cluster, Cloud and Grid Computing (CCGRID) Workshop on High Performance Machine Learning Workshop*. 2020. DOI: 10.1109/CCGrid49817.2020.00-14

[Nem+20b] Balazs Nemeth et al. “From Conditional Independence to Parallel Execution in Hierarchical Models”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 161–174. DOI: 10.1007/978-3-030-50371-0_12

This thesis does not include the contents of the following scientific contributions:

[Nem+15] Balazs Nemeth et al. “The Limits of Architectural Abstraction in Network Function Virtualization”. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, May 2015. DOI: 10.1109/inm.2015.7140348

[Nem+18b] Balazs Nemeth et al. “Reproducible Roulette Wheel Sampling for Message Passing Environments”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 799–805. DOI: 10.1007/978-3-319-93701-4_63

Bibliography

[Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. "Compilers: Principles, Techniques, and Tools". 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[Amd67] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560.

[And+03] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. "An Introduction to MCMC for Machine Learning". In: *Machine Learning* 50.1/2 (2003), pp. 5–43. DOI: 10.1023/a:1020281327116.

[Ang+14] Elaine Angelino, Eddie Kohler, Amos Waterland, Margo Seltzer, and Ryan P. Adams. "Accelerating MCMC via Parallel Predictive Prefetching". In: *30th Conference on Uncertainty in Artificial Intelligence (UAI)*. 2014.

[Ban+19] Marco Banterle, Clara Grazian, Anthony Lee, and Christian P. Robert. "Accelerating Metropolis-Hastings Algorithms by Delayed Acceptance". In: *Foundations of Data Science* 1.2 (2019), pp. 103–128. DOI: 10.3934/fods.2019005.

[BC05] Daniel Bovet and Marco Cesati. "Understanding The Linux Kernel". 3rd ed. Oreilly & Associates Inc, 2005. ISBN: 0596005652.

[BCW13] Anoop Korattikara Balan, Yutian Chen, and Max Welling. "Austerity in MCMC Land: Cutting the Metropolis-Hastings Budget". In: *CoRR* abs/1304.5299 (2013). arXiv: 1304.5299. URL: http://arxiv.org/abs/1304.5299.

[Bez+17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (Jan. 2017), pp. 65–98. DOI: 10.1137/141000671.

[BH77] Henry C. Baker and Carl Hewitt. "The Incremental Garbage Collection of Processes". In: *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*. ACM Press, 1977. DOI: 10.1145/800228.806932.

[BJP91] Micah Beck, Richard Johnson, and Keshav Pingali. "From Control Flow to Dataflow". Tech. rep. 2. June 1991, pp. 118–129. DOI: 10.1016/0743-7315(91)90016-3.

[Bła+07] Jacek Błażewicz, Klaus H Ecker, Erwin Pesch, Günter Schmidt, and Jan Weglarz. "Handbook on Scheduling: From Theory to Applications". Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-32220-7.

[BM07] Heiko Bauke and Stephan Mertens. "Random Numbers for Large-Scale Distributed Monte Carlo Simulations". In: *Physical Review E* 75.6 (June 2007). DOI: 10.1103/physreve.75.066701.

[Bro+10] Franois Broquedis, Jerome Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications". In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, Feb. 2010. DOI: 10.1109/pdp.2010.67.

[Cal14] Ben Calderhead. "A General Construction for Parallelizing Metropolis-Hastings Algorithms". In: *Proceedings of the National Academy of Sciences* 111.49 (Nov. 2014), pp. 17408–17413. DOI: 10.1073/pnas.1408184111.

[Car+17] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. "Stan: A Probabilistic Programming Language". In: *Journal of Statistical Software* 76.1 (2017). DOI: 10.18637/jss.v076.i01.

[CC00] B.a. Cipra and B.a. Cipra. “The Best of the 20th Century: Editors Name Top 10 Algorithms”. In: *SIAM News* 33.4 (2000), pp. 05–00.

[CF10] J. Andrés Christen and Colin Fox. “A General Purpose Sampling Algorithm for Continuous Distributions (the t-walk)”. In: *Bayesian Analysis* 5.2 (June 2010), pp. 263–281. DOI: 10.1214/10-ba603.

[CG92] George Casella and Edward I. George. “Explaining the Gibbs Sampler”. In: *The American Statistician* 46.3 (Aug. 1992), pp. 167–174. DOI: 10.1080/00031305.1992.10475878.

[Cha20] Veena S. Chakravarthi. “A Practical Approach to VLSI System on Chip (SoC) Design”. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-23049-4.

[Che06] Yu Chen. “Using Genetic Algorithms to Schedule Multiprocessor Systems under LOGP model”. MA thesis. Concordia University, 2006.

[Cio+14] Arthur A Ciociola, Lawrence B Cohen, Prasad Kulkarni, Costas Kefalas, Alan Buchman, Carol Burke, Tedd Cain, Jason Connor, Eli D Ehrenpreis, John Fang, Ronnie Fass, Robyn Karlstadt, Dan Pambianco, Joseph Phillips, Mark Pochapin, Paul Pockros, Philip Schoenfeld, and Raj Vuppalanchi. “How Drugs are Developed and Approved by the FDA: Current Process and Future Directions”. In: *American Journal of Gastroenterology* 109.5 (May 2014), pp. 620–623. DOI: 10.1038/ajg.2013.407.

[Con+18] Patrick R. Conrad, Andrew D. Davis, Youssef M. Marzouk, Natesh S. Pillai, and Aaron Smith. “Parallel Local Approximation MCMC for Expensive Models”. In: *SIAM/ASA Journal on Uncertainty Quantification* 6.1 (Jan. 2018), pp. 339–373. DOI: 10.1137/16m1084080.

[CR17] Nicolas Chopin and James Ridgway. “Leave Pima Indians Alone: Binary Regression as a Benchmark for Bayesian Computation”. In: *Statistical Science* 32.1 (Feb. 2017), pp. 64–87. ISSN: 0883-4237. DOI: 10.1214/16-sts581.

[CS18] Federico Della Croce and Rosario Scatamacchia. “The Longest Processing Time Rule for Identical Parallel Machines Revisited”. In: *Journal of Scheduling* 23.2 (Dec. 2018), pp. 163–176. DOI: 10.1007/s10951-018-0597-6.

[Cul+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. "LogP: Towards a Realistic Model of Parallel Computation". In: *ACM SIGPLAN Notices* 28.7 (July 1993), pp. 1–12. DOI: 10.1145/173284.155333.

[Cul+96] D.E. Culler, Lok Tin Liu, R.P. Martin, and C.O. Yoshikawa. "Assessing Fast Network Interfaces". In: *IEEE Micro* 16.1 (1996), pp. 35–43. DOI: 10.1109/40.482310.

[Cul86] David E Culler. "Dataflow Architectures". In: *Annual Review of Computer Science* 1.1 (1986), pp. 225–253.

[DFG01] Arnaud Doucet, Nando Freitas, and Neil Gordon, eds. "Sequential Monte Carlo Methods in Practice". Springer New York, 2001. DOI: 10.1007/978-1-4757-3437-9.

[Din+07] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P. Sadayappan, and Chau-Wen Tseng. "Dynamic Load Balancing of Unbalanced Computations Using Message Passing". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007. DOI: 10.1109/ipdps.2007.370581.

[DK14] Pradip S Devan and RK Kamat. "A Review - LOOP Dependence Analysis for Parallelizing Compiler". In: *International Journal of Computer Science and Information Technologies* 5.3 (2014), pp. 4038–4046.

[Dre07] Ulrich Drepper. "What Every Programmer Should Know About Memory". 2007.

[Dun+15] Adrian Dunne, Willem de Winter, Chyi-Hung Hsu, Shiferaw Mariam, Martine Neyens, José Pinheiro, and Xavier Woot de Trixhe. "The Method of Averaging Applied to Pharmacokinetic/Pharmacodynamic Indirect Response Models". In: *Journal of Pharmacokinetics and Pharmacodynamics* 42.4 (July 2015), pp. 417–426. DOI: 10.1007/s10928-015-9426-0.

[EGC16] Victor Eijkhout, Robert van de Geijn, and Edmond Chow. "Introduction To High Performance Scientific Computing". Zenodo, 2016. DOI: 10.5281/ZENODO.49897.

[Erm+19] Sergey Ermakov, Brian J. Schmidt, Cynthia J. Musante, and Craig J. Thalhauser. "A Survey of Software Tool Utilization and Capabilities for Quantitative Systems Pharmacology: What We Have and What We Need". In: *CPT: Pharmacometrics & Systems Pharmacology* 8.2 (Jan. 2019), pp. 62–76. DOI: 10.1002/psp4.12373.

[Fen+07] Xizhou Feng, Kirk W. Cameron, Carlos P. Sosa, and Brian Smith. "Building the Tree of Life on Terascale Systems". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007. DOI: 10.1109/ipdps.2007.370214.

[Fer+14] Alejandro Fernández, Vicenç Beltran, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. "Task-Based Programming with OmpSs and Its Application". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 601–612. DOI: 10.1007/978-3-319-14313-2_51.

[Fid+19] Matthew Fidler, Justin J. Wilkins, Richard Hooijmaijers, Teun M. Post, Rik Schoemaker, Mirjam N. Trame, Yuan Xiong, and Wenping Wang. "Nonlinear Mixed-Effects Model Development and Simulation Using nlmixr and Related R Open-Source Packages". In: *CPT: Pharmacometrics & Systems Pharmacology* 8.9 (July 2019), pp. 621–633. DOI: 10.1002/psp4.12445.

[Fil10] Yuval Filmus. "Two Proofs of the Central Limit Theorem". 2010.

[For+13] Daniel Foreman-Mackey, David W. Hogg, Dustin Lang, and Jonathan Goodman. "emcee: The MCMC Hammer". In: *Publications of the Astronomical Society of the Pacific* 125.925 (Mar. 2013), pp. 306–312. DOI: 10.1086/670067.

[Fri+02] Lena E. Friberg, Anja Henningsson, Hugo Maas, Laurent Nguyen, and Mats O. Karlsson. "Model of Chemotherapy-Induced Myelosuppression With Parameter Consistency Across Drugs". In: *Journal of Clinical Oncology* 20.24 (Dec. 2002), pp. 4713–4721. DOI: 10.1200/jco.2002.02.140.

[GD17] Wei Gong and Qingyun Duan. "An Adaptive Surrogate Modeling-Based Sampling Strategy for Parameter Optimization and Distribution Estimation (ASMO-PODE)". In: *Environmental Modelling & Software* 95 (Sept. 2017), pp. 61–75. DOI: 10.1016/j.envsoft.2017.05.005.

[Gév+18] Gábor E. Gévay, Tilmann Rabl, Sebastian Breß, Loránd Madai-Tahy, and Volker Markl. "Labyrinth: Compiling Imperative Control Flow to Parallel Dataflows". 2018. arXiv: 1809.06845 [cs.DC].

[Gou+17] Robert J. B. Goudie, Rebecca M. Turner, Daniela De Angelis, and Andrew Thomas. "MultiBUGS: A Parallel Implementation of the Bugs Modelling Framework for Faster Bayesian Inference". 2017. arXiv: 1704.03216 [stat.CO].

[Gra+03] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. "Introduction to Parallel Computing". 2nd ed. Addison-Wesley, 2003. ISBN: 9780201648652.

[Gra+06] Richard Graham, Galen Shipman, Brian Barrett, Ralph Castain, George Bosilca, and Andrew Lumsdaine. "Open MPI: A High-Performance, Heterogeneous MPI". In: *2006 IEEE International Conference on Cluster Computing*. IEEE, 2006. DOI: 10.1109/clustr.2006.311904.

[GW10] Jonathan Goodman and Jonathan Weare. "Ensemble Samplers with Affine Invariance". In: *Communications in Applied Mathematics and Computational Science* 5.1 (Jan. 2010), pp. 65–80. DOI: 10.2140/camcos.2010.5.65.

[GXG18] Hong Ge, Kai Xu, and Zoubin Ghahramani. "Turing: A Language for Flexible Probabilistic Inference". In: *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*. Ed. by Amos Storkey and Fernando Perez-Cruz. Vol. 84. Proceedings of Machine Learning Research. Playa Blanca, Lanzarote, Canary Islands: PMLR, Apr. 2018, pp. 1682–1690. URL: http://proceedings.mlr.press/v84/ge18b.html.

[Hab+18] Tom Haber, Valdemar Melicher, Thomas Kovac, Balazs Nemeth, and Johan Claes. "DiffMEM". 2018. URL: https://bitbucket.org/tomhaber/diffmem.

[Hal85] Robert H. Halstead. "MULTILISP: A Language for Concurrent Symbolic Computation". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.4 (Oct. 1985), pp. 501–538. DOI: 10.1145/4472.4478.

[Har16] Richard K. Harrison. "Phase II and Phase III Failures: 2013–2015". Nov. 2016. DOI: 10.1038/nrd.2016.184.

[HAR94] E.S.H. Hou, N. Ansari, and Hong Ren. "A Genetic Algorithm for Multiprocessor Scheduling". In: *IEEE Transactions on Parallel and Distributed Systems* 5.2 (1994), pp. 113–120. DOI: 10.1109/71.265940.

[Hay+14] Michael Hay, David W Thomas, John L Craighead, Celia Economides, and Jesse Rosenthal. "Clinical Development Success Rates for Investigational Drugs". In: *Nature Biotechnology* 32.1 (Jan. 2014), pp. 40–51. DOI: 10.1038/nbt.2786.

[HB15] Torsten Hoefler and Roberto Belli. "Scientific Benchmarking of Parallel Computing Systems". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*. ACM Press, 2015. DOI: 10.1145/2807591.2807644.

[Hel+17] Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. "HPX - An Open Source C++ Standard Library for Parallelism and Concurrency". In: *Proceedings of Open Source Supercomputing* (2017), p. 5.

[Hen+12] Niel Hens, Ziv Shkedy, Marc Aerts, Christel Faes, Pierre Van Damme, and Philippe Beutels. "Modeling Infectious Disease Parameters Based on Serological and Social Contact Data". Springer New York, 2012. DOI: 10.1007/978-1-4614-4072-7.

[Hin+05] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (Sept. 2005), pp. 363–396. DOI: 10.1145/1089014.1089020.

[HLR07] Torsten Hoefler, Andre Lichei, and Wolfgang Rehm. "Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007. DOI: 10.1109/ipdps.2007.370593.

[Hoe+15] Eef Hoeben, Willem De Winter, Martine Neyens, Damayanthi Devineni, An Vermeulen, and Adrian Dunne. "Population Pharmacokinetic Modeling of Canagliflozin in Healthy Volunteers and Patients with Type 2 Diabetes Mellitus". In: *Clinical Pharmacokinetics* 55.2 (Aug. 2015), pp. 209–223. DOI: 10.1007/s40262-015-0307-x.

[HR20] Tom Haber and Frank van Reeth. “Improving the Runtime Performance of Non-Linear Mixed-Effects Model Estimation”. In: *Euro-Par 2019: Parallel Processing Workshops*. Springer International Publishing, 2020, pp. 560–571. DOI: 10.1007/978-3-030-48340-1_43.

[JL08] Uwe Jensen and Constanze Lütkebohmert. “Change-Point Models”. In: *Encyclopedia of Statistics in Quality and Reliability* 1 (2008).

[JRS11] P. Jacob, C. P. Robert, and M. H. Smith. “Using Parallel Computation to Improve Independent Metropolis–Hastings Based Estimation”. In: *Journal of Computational and Graphical Statistics* 20.3 (Jan. 2011), pp. 616–635. DOI: 10.1198/jcgs.2011.10167.

[JY13] Momin Jamil and Xin She Yang. “A Literature Survey of Benchmark Functions for Global Optimisation Problems”. In: *International Journal of Mathematical Modelling and Numerical Optimisation* 4.2 (2013), p. 150. DOI: 10.1504/ijmmno.2013.055204.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors”. In: *ACM Computing Surveys* 31.4 (Dec. 1999), pp. 406–471. DOI: 10.1145/344588.344618.

[KBV00] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. “Fast Measurement of LogP Parameters for Message Passing Platforms”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000, pp. 1176–1183. DOI: 10.1007/3-540-45591-4_162.

[Ker10] Michael Kerrisk. “The Linux Programming Interface: A Linux and UNIX System Programming Handbook”. No Starch Press, 2010. ISBN: 1593272200.

[KK07] Christoph Kessler and Jörg Keller. “Models for Parallel Computing: Review and Perspectives”. In: *Mitteilungen-Gesellschaft für Informatik eV, Parallel-Algorithmen und Rechnerstrukturen* 24 (2007), pp. 13–29. ISSN: 01770454.

[KKT00] Tomasz Kalinowski, Iskander Kort, and Denis Trystram. “List Scheduling of General Task Graphs under LogP”. In: *Parallel Computing* 26.9 (July 2000), pp. 1109–1128. DOI: 10.1016/s0167-8191(00)00031-4.

[KL04] Estelle Kuhn and Marc Lavielle. “Coupling a Stochastic Approximation Version of EM with an MCMC Procedure”. In: *ESAIM: Probability and Statistics* 8 (Aug. 2004), pp. 115–131. DOI: 10.1051/ps:2004007.

[LA] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. DOI: 10.1109/cgo.2004.1281665.

[LaG+11] James LaGrone, Ayodunni Aribuki, Cody Addison, and Barbara Chapman. “A Runtime Implementation of OpenMP Tasks”. In: *OpenMP in the Petascale Era*. Springer Berlin Heidelberg, 2011, pp. 165–178. DOI: 10.1007/978-3-642-21487-5_13.

[Lam09] Christopher Lameter. “Shoot First and Stop the OS Noise”. In: *Linux Symposium*. Citeseer. 2009, p. 159.

[Li+13] Shigang Li, Jingyuan Hu, Xin Cheng, and Chongchong Zhao. “Asynchronous Work Stealing on Distributed Memory Systems”. In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, Feb. 2013. DOI: 10.1109/pdp.2013.35.

[LMT01] Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. “A “parareal” in Time Discretization of PDEs”. In: *Comptes Rendus de l’Académie des Sciences. Série I. Mathématique* 332 (Jan. 2001).

[Lun+00] David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. “WinBUGS - a Bayesian Modelling Framework: Concepts, Structure, and Extensibility”. In: *Statistics and Computing* 10.4 (2000), pp. 325–337. DOI: 10.1023/a:1008929526011.

[Mac03] David JC MacKay. “Information Theory, Inference and Learning Algorithms”. Cambridge university press, 2003.

[Mar+12] Petar Marendić, Jan Lemeire, Tom Haber, Dean Vučinić, and Peter Schelkens. “An Investigation into the Performance of Reduction Algorithms under Load Imbalance”. In: *Euro-Par 2012 Parallel Processing*. Springer Berlin Heidelberg, 2012, pp. 439–450. DOI: 10.1007/978-3-642-32820-6_44.

[Mar+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[MDJ06] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. "Sequential Monte Carlo Samplers". In: *Journal of the Royal Statistical Society. Series B (Statistical Methodology)* 68.3 (2006), pp. 411–436. ISSN: 13697412, 14679868. URL: `http://www.jstor.org/stable/3879283`.

[Mil+13] P A Milligan, M J Brown, B Marchant, S W Martin, P H van der Graaf, N Benson, G Nucci, D J Nichols, R A Boyd, J W Mandema, S Krishnaswami, S Zwillich, D Gruben, R J Anziano, T C Stock, and R L Lalonde. "Model-Based Drug Development: A Rational Approach to Efficiently Accelerate Drug Development". In: *Clinical Pharmacology & Therapeutics* 93.6 (Mar. 2013), pp. 502–514. DOI: `10.1038/clpt.2013.54`.

[Min10] Michael Minion. "A Hybrid Parareal Spectral Deferred Corrections Method". In: *Communications in Applied Mathematics and Computational Science* 5.2 (Dec. 2010), pp. 265–301. DOI: `10.2140/camcos.2010.5.265`.

[MML17] Ruben Mayer, Christian Mayer, and Larissa Laich. "The TensorFlow Partitioning and Scheduling Problem: It's the Critical Path!" In: *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning - DIDL '17*. ACM Press, 2017. DOI: `10.1145/3154842.3154843`.

[MS96] Larry W. McVoy and Carl Staelin. "lmbench: Portable Tools for Performance Analysis". In: *Proceedings of the USENIX Annual*

Technical Conference, San Diego, California, USA, January 22-26, 1996. USENIX Association, 1996, pp. 279–294.

[Mur+16] Lawrence M. Murray, Sumeetpal S. Singh, Pierre E. Jacob, and Anthony Lee. "Anytime Monte Carlo". 2016.

[Mur10] Lawrence Murray. "Distributed Markov Chain Monte Carlo". In: *LCCC: NIPS workshop on learning on cores, clusters and clouds* (Jan. 2010).

[Nem+15] Balazs Nemeth, Xavier Simonart, Neal Oliver, and Wim Lamotte. "The Limits of Architectural Abstraction in Network Function Virtualization". In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, May 2015. DOI: 10.1109/inm.2015.7140348.

[Nem+17a] Balazs Nemeth, Tom Haber, Thomas J. Ashby, and Wim Lamotte. "Improving Operational Intensity in Data Bound Markov Chain Monte Carlo". In: *Procedia Computer Science* 108 (2017), pp. 2348–2352. DOI: 10.1016/j.procs.2017.05.024.

[Nem+17b] Balazs Nemeth, Tom Haber, Jori Liesenborgs, and Wim Lamotte. "Distributed Affine-Invariant MCMC Sampler". In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Sept. 2017. DOI: 10.1109/cluster.2017.68.

[Nem+18a] Balazs Nemeth, Tom Haber, Jori Liesenborgs, and Wim Lamotte. "Relaxing Scalability Limits with Speculative Parallelism in Sequential Monte Carlo". In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Sept. 2018. DOI: 10.1109/cluster.2018.00065.

[Nem+18b] Balazs Nemeth, Tom Haber, Jori Liesenborgs, and Wim Lamotte. "Reproducible Roulette Wheel Sampling for Message Passing Environments". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 799–805. DOI: 10.1007/978-3-319-93701-4_63.

[Nem+19] Balazs Nemeth, Tom Haber, Jori Liesenborgs, and Wim Lamotte. "Approximate Repeated Administration Models for Pharmacometrics". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2019, pp. 628–641. DOI: 10.1007/978-3-030-22734-0_46.

[Nem+20a] Balazs Nemeth, Tom Haber, Jori Liesenborgs, and Wim Lamotte. "Automatic Parallelization of Probabilistic Models with Varying Load Imbalance". In: *International Symposium on Cluster, Cloud and Grid Computing (CCGRID) Workshop on High Performance Machine Learning Workshop*. 2020. DOI: 10.1109/CCGrid49817.2020.00-14.

[Nem+20b] Balazs Nemeth, Tom Haber, Jori Liesenborgs, and Wim Lamotte. "From Conditional Independence to Parallel Execution in Hierarchical Models". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 161–174. DOI: 10.1007/978-3-030-50371-0_12.

[NWX14] Willie Neiswanger, Chong Wang, and Eric P. Xing. "Asymptotically Exact, Embarrassingly Parallel MCMC". In: *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*. UAI'14. Quebec City, Quebec, Canada: AUAI Press, 2014, pp. 623–632. ISBN: 9780974903910.

[OA09] Fatma A. Omara and Mona M. Arafa. "Genetic Algorithms for Task Scheduling Problem". In: *Foundations of Computational Intelligence Volume 3*. Springer Berlin Heidelberg, 2009, pp. 479–507. DOI: 10.1007/978-3-642-01085-9_16.

[OF14] Joel S. Owen and Jill Fiedler-Kelly. "Introduction to Population Pharmacokinetic/Pharmacodynamic Analysis with Nonlinear Mixed Effects Models". John Wiley & Sons, Inc, June 2014. DOI: 10.1002/9781118784860.

[P M12] Kevin P. Murphy. "Machine Learning: A Probabilistic Perspective". 2012. ISBN: 9780262306164.

[Pad11] David Padua, ed. "Encyclopedia of Parallel Computing". Springer US, 2011. DOI: 10.1007/978-0-387-09766-4.

[PB00] José C Pinheiro and Douglas M Bates. "Mixed-effects models in S and S-PLUS". Springer, 2000. ISBN: 0387989579.

[Pre+07] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. "Numerical Recipes: The Art of Scientific Computing". 3rd ed. Cambridge university press, 2007. ISBN: 9780521880688.

[Pre75] David L. Presberg. "The Paralyzer: Ivtran's Parallelism Analyzer and Synthesizer". In: vol. 10. 3. Association for Computing Machinery (ACM), Mar. 1975, pp. 9–16. DOI: 10.1145/390015.808396.

[Pro18] Jerzy Proficz. “Improving All-Reduce Collective Operations for Imbalanced Process Arrival Patterns”. In: *The Journal of Supercomputing* 74.7 (Apr. 2018), pp. 3071–3092. DOI: 10.1007/s11227-018-2356-z.

[Qiu+16] Junfei Qiu, Qihui Wu, Guoru Ding, Yuhua Xu, and Shuo Feng. “A Survey of Machine Learning for Big Data Processing”. In: *EURASIP Journal on Advances in Signal Processing* 2016.1 (May 2016). DOI: 10.1186/s13634-016-0355-x.

[Ras03] CE. Rasmussen. “Gaussian Processes to Speed up Hybrid Monte Carlo for Expensive Bayesian Integrals”. In: *Bayesian Statistics 7* (2003), pp. 651–659.

[Rau+09] A. Raue, C. Kreutz, T. Maiwald, J. Bachmann, M. Schilling, U. Klingmüller, and J. Timmer. “Structural and Practical Identifiability Analysis of Partially Observed Dynamical Models by Exploiting the Profile Likelihood”. In: *Bioinformatics* 25.15 (June 2009), pp. 1923–1929. DOI: 10.1093/bioinformatics/btp358.

[RC99] Christian P. Robert and George Casella. “The Metropolis-Hastings Algorithm”. In: *Springer Texts in Statistics*. Springer New York, 1999, pp. 231–283. DOI: 10.1007/978-1-4757-3071-5_6.

[Rod+15] Leyanis Rodríguez-Vera, Mayra Ramos-Suzarte, Eduardo Fernández-Sánchez, Jorge Luis Soriano, Concepción Peraire Guitart, Gilberto Castañeda Hernández, Carlos O. Jacobo-Cabral, Niurys de Castro Suárez, and Helena Colom Codina. “Semimechanistic Model to Characterize Nonlinear Pharmacokinetics of Nimotuzumab in Patients with Advanced Breast Cancer”. In: *The Journal of Clinical Pharmacology* 55.8 (Apr. 2015), pp. 888–898. DOI: 10.1002/jcph.496. URL: https://doi.org/10.1002/jcph.496.

[Sal+11] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. “Parallel Random Numbers: As Easy As 1, 2, 3”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’11*. ACM Press, 2011. DOI: 10.1145/2063384.2063405.

[San07] Paul J. Sanchez. “Fundamentals of Simulation Modeling”. In: *2007 Winter Simulation Conference*. IEEE, Dec. 2007. DOI: 10.1109/wsc.2007.4419588.

[SBS13] Thomas J. Schriber, Daniel T. Brunner, and Jeffrey S. Smith. "Inside Discrete-Event Simulation Software: How It Works and Why It Matters". In: *2013 Winter Simulations Conference (WSC)*. IEEE, Dec. 2013. DOI: 10.1109/wsc.2013.6721439.

[Sco+16] Steven L. Scott, Alexander W. Blocker, Fernando V. Bonassi, Hugh A. Chipman, Edward I. George, and Robert E. McCulloch. "Bayes and Big Data: The Consensus Monte Carlo Algorithm". In: *International Journal of Management Science and Engineering Management* 11.2 (Feb. 2016), pp. 78–88. DOI: 10.1080/17509653.2016.1142191.

[Ski91] D. B. Skillicorn. "Models for Practical Parallel Computation". In: *International Journal of Parallel Programming* 20.2 (Apr. 1991), pp. 133–158. DOI: 10.1007/bf01407840.

[Sol+12] Antti Solonen, Pirkka Ollinaho, Marko Laine, Heikki Haario, Johanna Tamminen, and Heikki Järvinen. "Efficient MCMC for Climate Model Parameter Estimation: Parallel Adaptive Chains and Early Rejection". In: *Bayesian Analysis* 7.3 (Sept. 2012), pp. 715–736. DOI: 10.1214/12-ba724.

[SS06] Devinderjit Sivia and John Skilling. "Data Analysis: A Bayesian Tutorial". 2nd ed. Oxford Science Publications. Oxford University Press, 2006. ISBN: 9780198568322.

[Thi18] Samuel Thibault. "On Runtime Systems for Task-based Programming on Heterogeneous Platforms". PhD thesis. Université de Bordeaux, 2018.

[Tho10] Madeleine B. Thompson. "A Comparison of Methods for Computing Autocorrelation Time". 2010. arXiv: 1011.0175 [stat.CO].

[Tra18] Mirjam N. Trame. "PAGE 2018 nlmixr Workshop Materials". 2018. URL: https://github.com/nlmixrdevelopment/PAGE-2018.

[Tsa+05] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications". In: *Proceedings of the 19th annual international conference on Supercomputing - ICS '05*. ACM Press, 2005. DOI: 10.1145/1088149.1088190.

[Web18] Sebastian Weber. "Supporting Drug Development as a Bayesian in Due Time?!" In: *PAGE* (2018).

[Win+17] Willem Winter, Adrian Dunne, Xavier Woot Trixhe, Damayanthi Devineni, Chyi-Hung Hsu, Jose Pinheiro, and David Polidori. "Dynamic Population Pharmacokinetic /Pharmacodynamic Modelling and Simulation Supports Similar Efficacy in Glycosylated Haemoglobin Response with Once or Twice-Daily Dosing of Canagliflozin". In: *British Journal of Clinical Pharmacology* 83.5 (Jan. 2017), pp. 1072–1081. DOI: 10.1111/bcp.13180.

[WM95] Wm. A. Wulf and Sally A. McKee. "Hitting the memory wall". In: *ACM SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. DOI: 10.1145/216585.216588.

[Wol95] Michael Joseph Wolfe. "High Performance Compilers for Parallel Computing". Ed. by Carter Shanklin and Leda Ortega. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0805327304.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (Apr. 2009), pp. 65–76. DOI: 10.1145/1498765.1498785.

[Yil12] Ilker Yildirim. "Bayesian Inference: Gibbs Sampling". In: *Technical Note, University of Rochester* (2012).

[Zeu+11] Thomas Zeugmann, Pascal Poupart, James Kennedy, Xin Jin, Jiawei Han, Lorenza Saitta, Michele Sebag, Jan Peters, J. Andrew Bagnell, Walter Daelemans, Geoffrey I. Webb, Kai Ming Ting, Kai Ming Ting, Geoffrey I. Webb, Jelber Sayyad Shirabad, Johannes Fürnkranz, Eyke Hüllermeier, Stan Matwin, Yasubumi Sakakibara, Pierre Flener, Ute Schmid, Cecilia M. Procopiuc, Nicolas Lachiche, and Johannes Fürnkranz. "Particle Swarm Optimization". In: *Encyclopedia of Machine Learning*. Springer US, 2011, pp. 760–766. DOI: 10.1007/978-0-387-30164-8_630.

[Zha+07] Yunquan Zhang, Guoliang Chen, Guangzhong Sun, and Qiankun Miao. "Models of Parallel Computation: A Survey and Classification". In: *Frontiers of Computer Science in China* 1.2 (May 2007), pp. 156–165. DOI: 10.1007/s11704-007-0016-1.

[ZQR16] Dongfang Zhao, Kan Qiao, and Ioan Raicu. "Towards Cost-Effective and High-Performance Caching Middleware for Distributed Systems". In: *International Journal of Big Data Intelligence* 3.2 (2016), p. 92. DOI: 10.1504/ijbdi.2016.077358.

[ZS13] Wei Zheng and Rizos Sakellariou. “Stochastic DAG Scheduling Using a Monte Carlo Approach”. In: *Journal of Parallel and Distributed Computing* 73.12 (Dec. 2013), pp. 1673–1689. DOI: 10.1016/j.jpdc.2013.07.019.