NTNU – Trondheim
Norwegian University of
Science and Technology

Parallel Solutions of Stochastic Differential Equations

Agnes Koi Alexandersen

Master of Science in Physics and Mathematics
Submission date: June 2015
Supervisor: Arvid Næss, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

NTNU

Norwegian University of
Science and Technology

Parallel Solutions of Stochastic Differential Equations

Author:
Agnes Koi Alexandersen

Supervisor:
Arvid Naess

Department:
Department of Mathematical Science

June 2015

"As far as the laws of mathematics refer to reality, they are not certain, and as far as they are certain, they do not refer to reality."

Albert Einstein

Preface

This report is my master thesis for the conclusion for the Master's gram Industrial Mathematics at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisor Professor Arvid Næss for general guidance during the semester, Lill Harriet Koi and Torodd Eriksen for proofreading this thesis and Professor Anne Cathrine Elster and Thomas Løfsgaard Falch from the Department of Computer and Information Science for helping me with some issues regarding GPU programming and giving me access to one of their servers.

Abstract

A generalised Lotka-Volterra model for a pair of interacting populations of predator and pray is studied with and without the predator being harvested. The Monte Carlo method is introduced to propagate the initial distributions of the populations. The main purpose of this work is to develop a more time efficient solution to the Lotka-Volterra model and to see whether or not this is a good option to the path integration method. To compare runtimes the algorithm is implemented using the Message Passing Interface as well as on a GPU using CUDA.

Abbreviations

SDE	**S**tochastic **D**ifferential **E**quation
MPI	**M**essage **P**assing **I**nterface
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
PDF	**P**robability **D**ensity **F**unction
GPU	**G**raphics **P**rocessing **U**nit
API	**A**pplication **P**rogramming **I**nterface

Symbols

$\mathbb{R}$	The set of real numbers
$\mathbb{R}^+$	The set of real positive numbers
$\mathbb{C}$	The set of complex numbers
$\mathbb{R}^n$	The cartesian product $\underbrace{\mathbb{R} \times \mathbb{R} \times \cdots \times \mathbb{R}}_{n \text{ times}}$

Contents

Chapter 1

Preliminaries

1.1 Basic Properties [1]

Let Ω be the sample space and A be an event in Ω. A probability has the following properties

1. $0 \leq P\{A\} \leq 1$,
2. $P\{\Omega\} = 1$,
3. For each sequence $A_1, A_2, \ldots$ of mutually disjoint events the following holds

$$P\Big\{\bigcup_{i=1}^{\infty} A_i\Big\} = \sum_{i=1}^{\infty} P\{A\}.$$

A probability density function (PDF) is a function that describes the likelihood for this random variable to take on a given value. The PDF, $P(x)$, of a continuous distribution is defined as the derivative of the (cumulative) distribution function $D(x)$,

$$\begin{aligned} D'(x) &= \big[P(x)\big]_{-\infty}^{x} \\ &= P(x) - P(-\infty) = P(x), \end{aligned}$$

which implies that

$$D(x) = P(X \leq x) = \int_{-\infty}^{x} P\xi d\xi.$$

The PDF of the normal distribution is given by

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the expectation and σ is the standard deviation of the distribution.

A key tool in the analysis of sums of independent random variables is the concept of characteristic functions.

Definition 1.1. Given an $\mathbb{R}^d$-valued random vector $\overrightarrow{Z}$, the *characteristic function* of $\overrightarrow{Z}$ is given by

$$c(\theta) = \mathrm{E}\left[e^{i\theta\overrightarrow{Z}}\right],$$

for $\theta \in \mathbb{R}^d$.

Remark 1.1. When $\overrightarrow{Z}$ has a probability density function f, the characteristic function is essentially its Fourier transform. i.e

$$c(\theta) = \int_{\mathbb{R}^d} e^{i\theta\overrightarrow{Z}} f(\overrightarrow{Z}) d\overrightarrow{Z}.$$

1.2 Stochastic Processes [2]

A *stochastic process* is a mathematical model that evolves over time in a probabilistic manner. It is a collection of random variables, representing the evolution of some system of random values over time. It describes a process which, given an initial condition, can evolve in several and often infinitely many directions. In this section some of the basic concepts of probability theory is reviewed. We start by defining the concept of a Markov chain which will be very important later on.

1.2.1 Markov Chains

A sequence $x_1, x_2, \ldots$ of random elements of some set is a *Markov chain* if the conditional distribution of x_{n+1} given $x_1, x_2, \ldots, x_n$ depends on x_n only. The *state space* of the markov chain is the set in which the x_i take values. If the conditional distribution of x_{n+1} given x_n does not depend on n we say that the Markov chain has *stationary*

transition probabilities. This is the kind of Markov chains which will be used in the Monte Carlo method introduced later in this report.

Definition 1.2. If for times $0 \leq t_1 < t_2 < \cdots < t_m < t_{m+1} < \infty$ with corresponding spatial points $x_1, x_2, \ldots, x_m \in \mathbb{R}^n$ we have the equality

$$P(X_{t_{m+1}} \in S \mid X_{t_1} = x_1 \cap \cdots \cap X_{t_m} = x_m) = P(X_{t_{m+1}} \in S \mid X_{t_m} = x_m), \quad \forall S \in \mathcal{B}(\mathbb{R}^n),$$

the stochastic process X_t is called a Markov process. $\mathcal{B}(\mathbb{R}^n)$ is the *Borel set*. If the same property holds for non-negative integer times, the stochastic process is called a Markov chain.

1.3 Brownian Motion [3]

The specific stochastic process which we will study in this project is a Brownian motion.

Definition 1.3. A real-valued stochastic process $\{B(t) : t \geq 0\}$ is called a Brownian motion with start in $x \in \mathbb{R}$ if the following holds:

1. $B_0 = x$.
2. The process has independent increments, i.e, for all times $0 \leq t_1 \leq t_2 \leq \cdots \leq t_n$ the increments $B(t_n) - B(t_{n-1})$, $B(t_{n-1}) - B(t_{n-2})$, ..., $B(t_2) - B(t_1)$ are independent random variables.
3. For all $t \geq 0$ and $h > 0$, the increments $B(t+h) - B(t)$ are normally distributed with expectation zero and variance h.
4. Almost surely, the function $t \to B(t)$ is continuos.

We say that $\{B(t) : t \geq 0\}$ is a standard Brownian motion if $x = 0$.

One of the crucial facts about Brownian motion is that

$$(dB)^2 = dt.$$

To partially justify this statement we compute the expected value of $(B_{t+\delta} - B_t)^2$:

$$\begin{aligned} E\big[(B_{t+\delta} - B_t)^2\big] &= \mathrm{Var}\big[(B_{t+\delta} - B_t)\big] \\ &= \delta. \end{aligned}$$

1.4 The Stochastic Differential Equation

Throughout, the stochastic differential equations (SDE) will be interpreted in the Itô sense. The class of SDE considered is of the following form

$$dY_t = \mu(Y_t, t)dt + \sigma(Y_t, t)dW_t, \tag{1.1}$$

where $Y_t = (Y_{1,t}, \ldots, Y_{n,t})^T$ is the (n-dimensional) state space vector processes, $W_t = (W_{1,t}, \ldots, W_{m,t})^T$ is a standard (unit) scalar or vector (m-dimensional) Browninan motion process depending on the type of system being studied. The terms μ and σ are often referred to as the drift term and the diffusion term, respectively.

By observing that in the absence of the term σ in (1.1) the solution Y_t would have a purely deterministic behaviour and in the absence of μ, dY_t would be a pure white noise process.

1.5 Itō's Lemma

If a stochastic variable Y_t satisfies the SDE given in (1.1), then given any function $f(Y_t, t)$ which is twice differentiable with a continuous second derivative, we can write (by [14, p.13] [15])

$$\begin{aligned} df(Y_t, t) = &\left[\left(\frac{\partial}{\partial t} + \mu(Y_t, t)\frac{\partial}{\partial Y_t} + \frac{1}{2}\sigma^2(Y_t, t)\frac{\partial^2}{\partial Y_t^2}\right)f(Y_t, t)\right]dt \\ &+ \left[\sigma(Y_t, t)\frac{\partial}{\partial Y_t}f\right]dW_t. \end{aligned} \tag{1.2}$$

Proof. A Taylor series expansion on $f(Y_t, t)$ yields

$$df(Y_t, t) = \left(\frac{\partial}{\partial Y_t}dY_t + \frac{\partial}{\partial t}dt + \frac{1}{2}\frac{\partial^2}{\partial Y_t^2}dY_t^2 + \frac{\partial^2}{\partial Y_t \partial t}dY_t dt + \frac{1}{2}\frac{\partial^2}{\partial t^2} + \ldots\right)f(Y_t, t). \tag{1.3}$$

The increment of the Wiener process is normally distributed with mean equal to zero and variance equal to dt and we can hence write $dW_t = \epsilon\sqrt{dt}$. Introducing $\mu = \mu(Y_t, t)$ and $\sigma = \sigma(Y_t, t)$ we can express $(dY_t)^2$ as

$$(dY_t)^2 = (\mu dt + \sigma dW_t)^2 = \mu^2 dt^2 + \sigma^2 (dW_t)^2 + 2\mu\sigma dt dW_t \approx \sigma^2 (dW_t)^2 = \sigma^2 \epsilon^2 dt.$$

It can be shown that the variance of $\epsilon^2 dt$ is of order dt^2 and that as a result of this, $\epsilon^2 dt$ becomes nonstochastic and equal to its expected value of dt as dt tends to zero. Taking limits as dY_t and dt tend to zero in (1.3) and using this last result we obtain

$$df(Y_t, t) = \left(\frac{\partial}{\partial Y_t} dY_t + \frac{\partial}{\partial t} dt + \frac{1}{2} \frac{\partial^2}{\partial Y_t^2} \sigma^2 dt \right) f(Y_t, t).$$

Substituting for dY_t from (1.1) yields (1.2) and the proof is complete.

□

1.6 Existence and Uniqueness of Solution [4]

The conditions for existence and uniqueness of a solution of a SDE is given by the following theorem:

Theorem 1.1. *If the function $\mu, \sigma : \mathbb{R}^n \times [t_0, T] \to \mathbb{R}^n$ satisfies the following conditions*

1. *$\mu(y, t)$ and $\sigma(y, t)$ are jointly $\mathcal{L}^2$-measurable in $(y, t) \in \mathbb{R}^n \times [t_0, T]$.*
2. *There exists a constant $K > 0$ such that*

$$\| \mu(y, t) - \mu(x, t) \| \leq K \| y - x \|,$$
$$\| \sigma(y, t) - \sigma(x, t) \| \leq K \| y - x \|,$$

for all $t \in [t_0, T]$ and $x, y \in \mathbb{R}^n$.

3. There exists a constant $D > 0$ such that

$$\| \mu(y,t) \|^2 \leq D^2(1+ \| y \|^2)$$
$$\| \sigma(y,t) \|^2 \leq D^2(1+ \| y \|^2)$$
$$.$$

and in addition Y_{t_0} is $\mathcal{A}_{t_0}$-measurable with $E(\| Y_{t_0} \|) < \infty$, the SDE (1.1) has a pathwise unique solution Y_t on $[t_0, T]$ with

$$\sup_{t_0 \leq t \leq T} E(\| Y_t \|)^2 < \infty.$$

The last two requirements puts severe restrictions on the functions μ and σ. If the solution is contained in some bounded closed subset of $\mathbb{R}^n$ and μ and σ are bounded continuous functions with bounded derivatives on the same subset, then μ and σ satisfy all the requirements. We will here forth assume that this holds such that a unique solution of the SDE exists.

1.7 The Fokker-Planck Equation [5]

The Fokker-Planck (FP) equation describes physically the probability density of the velocity of a particle. For the itō process given by the SDE

$$dX_t = \mu(X_t, t)dt + \sqrt{2D(X_t, t)}dW_t,$$

where $\mu(X_t, t)$ is the drift term, $D(X_t, t)$ is the diffusion coefficient and W_t is a Wiener process, the one-dimensional FP equation for the PDF, $f(x, t)$, of the random variable X_t is given by

$$\frac{\partial}{\partial t} f(x,t) = -\frac{\partial}{\partial x}[\mu(x,t)f(x,t)] + \frac{\partial^2}{\partial x^2}[D(x,t)f(x,t)].$$

This can be extended to higher dimensions:

$$\frac{\partial}{\partial t} f(x,t) = -\sum_{i=1}^{n} \frac{\partial}{\partial x_i}[\mu_i(x,t)f(x,t)] + \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{\partial^2}{\partial x_i \partial x_j}[D_{ij}(x,t)f(x,t)]$$

where $\mu(X_t, t)$ is the drift vector and $D(X_t, t)$ is the diffusion matrix.

Chapter 2

The Lotka-Volterra model

2.1 Introduction

The Lotka-Volterra (LV) model is the simplest model of predator-prey interactions. It is based on linear per capita growth rates, and is given as

$$\begin{aligned} \dot{U} &= -mU + k\beta UV, \\ \dot{V} &= \alpha V - \beta UV. \end{aligned} \tag{2.1}$$

The model (2.1) describes oscillatory behaviour of the population sizes of two nonlinearly interacting species (predator-prey or parasite-host pairs). In this case $U(t)$ and $V(t)$ are the population sizes of the predator (or parasites) and prey (or host), respectively and $\dot{U}(t)$ and $\dot{V}(t)$ represents their growth rates over time. The preys are assumed to have unlimited food supply, and to reproduce exponentially unless subject to predatation. The term with α represents this exponential growth. mU represents the loss rate of the predators due to either natural death or emigration and it leads to an exponential decay in the absence of prey. The term βUV represents rate of predation upon the prey which is assumed to be proportional to the rate at which the predators and the prey meet. $k\beta UV$ is the growth rate of the predator population.

The generalized Lotka-Volterra model can be governed by the following pair of differential equations:

$$\begin{aligned} \dot{U} &= -mU + k\beta UV, \\ \dot{V} &= \alpha V[1 + h(t)] - \beta UV - \gamma V^2. \end{aligned} \tag{2.2}$$

The function $h(t)$ simulates the influence of temporal variation in the environmental conditions through those of the prey's reproduction rate α. The term with V^2 governs self-limitation in the growth of the prey population size in the absence of predators. From the standpoint of biology we are only interested in the dynamics of model (2.2) in the closed first quadrant $\mathbb{R}^+ \times \mathbb{R}^+$.

The simplest model of this kind is given by

$$h(t) = \xi(t), \tag{2.3}$$

where $h(t)$ is a zero-mean "physical" Gaussian white noise with the following properties

- $\langle \xi(t) \rangle = 0$.
- $\langle \xi(t)\xi(t+\tau) \rangle = D_\xi \delta(\tau)$.

where angular brackets denote probabilistic averaging. D_ξ is a positive constant related to the strength of the diffusion process and $\delta(\cdot)$ is the Dirac delta function. This model only accounts for purely random variations. The system (2.2) and (2.3) may be studied using the theory of Markov processes, which leads to the Fokker-Planck-Kolmogrov (FKP) partial differential equation for the joint probability density function (PDF) of the system's state variables. It s possible to find an analytical solution to the specific FKP equation for the stationary PDF $p_{UV}(u, v)$ of $U(t)$ and $V(t)$ corresponding to the system (2.2) and (2.3).

Thus it seems appropriate to study the system (2.2) for a random $h(t)$ with some "hidden" periodicity, or for a periodic $h(t)$ with some random disorder. One approach implies just simple addition of a zero-mean Gaussian white noise $\xi(t)$, to the sinusoid, i.e.,

$$h(t) = \lambda \sin \nu t + \xi(t). \tag{2.4}$$

Another approach implies random white-noise temporal variation in the phase of the sinusoid, i.e.,

$$\begin{aligned} h(t) &= \lambda \sin Z(t), \\ \dot{Z} &= \nu + \xi(t), \end{aligned} \tag{2.5}$$

where $\xi(t)$ is as mentioned above.

2.2 Harvesting

Threshold policy (TP) is a way to control the population dynamics. By introducing harvesting every time the population is above a certain threshold, and removing it once the population falls below that level, the population sizes can be controlled. There are several ways to model the harvesting and the general model is given by

$$\begin{aligned} \dot{U} &= -mU + k\beta UV - h_1, \\ \dot{V} &= \alpha V[1 + h(t)] - \beta UV - \gamma V^2 - h_2, \end{aligned}$$

where h_i, $i = 1, 2$ are exploited terms of ith species standing for harvesting. We will now have a closer look at this function and how it affects the dynamics of the system. Let u denote the population of the species, T the population threshold and ϵ the harvesting rate, a simple model for the harvesting would be

$$\phi(u) = \begin{cases} 0 & : u < T \\ \epsilon & : u \geq T. \end{cases}$$

This function is discontinuous at $u = T$ and is therefore not suitable for real-life applications. Another approach is to introduce a gradual increase in the harvesting rate as the population increases above the threshold. Let ϵ denote the maximal harvesting rate and α the threshold for where the maximal harvesting starts. The new model is given by

$$\psi(u) = \begin{cases} 0 & : u < T \\ \frac{\epsilon(u-T)}{\alpha - T} & : T \geq u < \alpha \\ \epsilon & : u \geq \alpha. \end{cases}$$

The last approach we will give here is to introduce a gradual continuos increase in the harvesting rate, i.e.

$$H(u) = \begin{cases} 0 & : u < T \\ \frac{h(u-T)}{h+(u-T)} & : u \geq T, \end{cases}$$

where h is the upper limit for the harvesting rate .

Introducing this to (2.2) yields

$$\begin{aligned} \dot{U} &= -mU + k\beta UV - H(U), \\ \dot{V} &= \alpha V[1 + h(t)] - \beta UV - \gamma V^2. \end{aligned} \tag{2.6}$$

Figure 2.1 shows how these three different harvesting functions increases as a function of u.

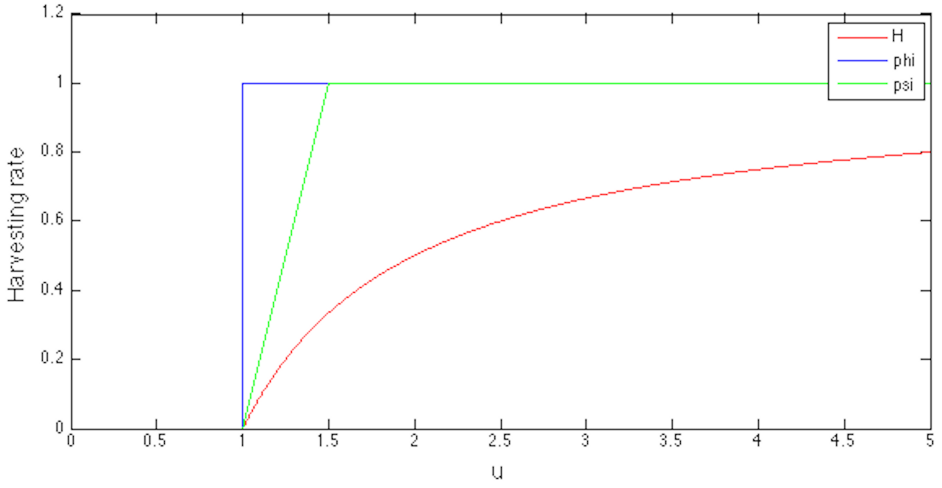


FIGURE 2.1: The three different harvesting functions with $T = h = \epsilon = 1, \alpha = 1.5$.

2.3 Equilibrium Points and Stationary Solutions

The deterministic system corresponding to (2.6) is given by

$$\begin{aligned} \dot{u} &= -u[m - k\beta v] - H(u) \\ \dot{v} &= v[\alpha - \beta u - \gamma v]. \end{aligned} \tag{2.7}$$

By restring us to the case where $u < T$, the harvesting function is equal to zero and equation (2.7), becomes (2.2). In order to find the equilibrium points, we set the two equations in (2.2) equal to zero and solve for $\dot{U}$ and $\dot{V}$. The result is the two equilibrium points $(u, v) = (0, 0)$ and $(u, v) = (0, \frac{\alpha}{\gamma})$, where in both cases one or both of the species are extinct. We are more interested in the equilibrium points where both species are alive. We introduce a mapping $x \to \exp(x)$ which is a bijection from $\mathbb{R}$ to $(0, \infty)$ given by

$$x = \ln(u) \quad y = \ln(v).$$

Inserting this into (2.7) gives

$$\begin{aligned} \dot{x} &= -m + k\beta \exp(y) - \tilde{H}(x) \\ \dot{y} &= \alpha - \beta \exp(x) - \gamma \exp(y), \end{aligned} \tag{2.8}$$

where $\tilde{H}(x)$ is given by

$$\tilde{H}(x) = \begin{cases} 0 & \text{if } x < \ln(T) \\ \frac{h(1 - T\exp(-x))}{h + (\exp(x) - T)} & \text{if } x \geq \ln(T). \end{cases}$$

The restriction is now $x < \ln(T)$ and the equilibrium point is $(x, y) = (\ln(\frac{\alpha}{\beta} - \frac{\gamma m}{k\beta^2}), \ln(\frac{m}{k\beta}))$. A linearisation at this equilibrium point yields

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} 0 & m \\ \frac{\gamma m}{k\beta} - \alpha & -\frac{\gamma m}{k\beta} \end{bmatrix} \begin{bmatrix} x - \ln(\frac{\alpha}{\beta} - \frac{\gamma m}{k\beta^2}) \\ y - \ln(\frac{m}{k\beta}) \end{bmatrix}.$$

The matrix in the above equation has the characteristic function $\lambda^2 + \frac{\gamma m}{k\beta}\lambda + m(\alpha - \frac{\gamma m}{k\beta}) = 0$. The roots of this equation are

$$\lambda_{\pm} = \frac{-\frac{\gamma m}{k\beta} \pm \sqrt{\left(\frac{\gamma m}{k\beta}\right)^2 - 4m(\alpha - \frac{\gamma m}{k\beta})}}{2}. \tag{2.9}$$

We are only interested in the equilibrium point which are physically acceptable, that is $u, v > 0$, which is equivalent to $(\frac{\alpha}{\beta} - \frac{\gamma m}{k\beta^2}), \frac{m}{k\beta} > 0$. If this is the case and all parameters in the above equation are positive then $\mathrm{Re}(\lambda_{\pm}) < 0$ and the equilibrium point is stable and therefore we can expect a global maximum of the probability distribution near this point.

When applying Itō's lemma to the logarithmic transformation $X_t = \ln(U_t)$ and $Y_t = \ln(V_t)$, and using that the SDE in the absence of the harvesting term with the function $h(t)$ as described by (2.3) we get

$$\begin{aligned} dX_t &= (-m + k\beta \exp(Y_t))dt \\ dY_t &= ((\alpha - \frac{1}{2}\alpha^2 D_\xi) - \beta \exp(X_t) - \gamma \exp(Y_t))dt + \alpha\sqrt{D_\xi} dB_t. \end{aligned}$$

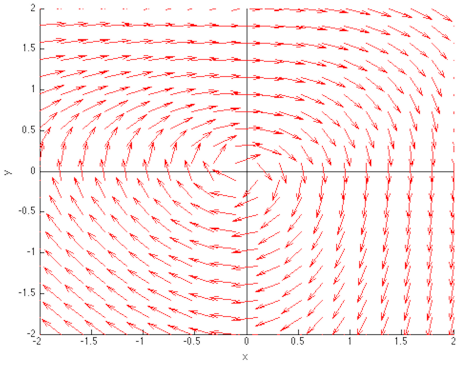
We can rewrite these equations to

$$\begin{aligned} dX_t &= \frac{\partial G(X_t, Y_t)}{\partial Y_t} dt \\ dY_t &= -\left(\frac{\partial G(X_t, Y_t)}{\partial X_t} - \frac{\gamma}{k\beta}\frac{\partial G(X_t, Y_t)}{\partial Y_t}\right)dt + \alpha\sqrt{D_\xi} dB_t, \end{aligned}$$

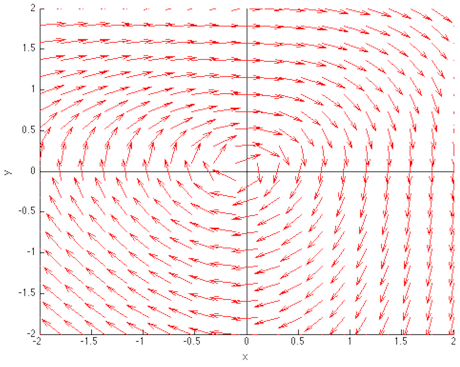
where $G(x, y) = k\beta \exp(y) - my + \beta \exp(x) - ((\alpha - \frac{1}{2}\alpha^2 D_\xi) - \frac{\gamma m}{k\beta})x$. Using the Fokker-Planck equation it can be shown that the stationary solution is given by

$$p_{XY}(x, y) = C \exp\left(-\frac{2\gamma}{k\beta D_\xi \alpha^2} G(x, y)\right).$$

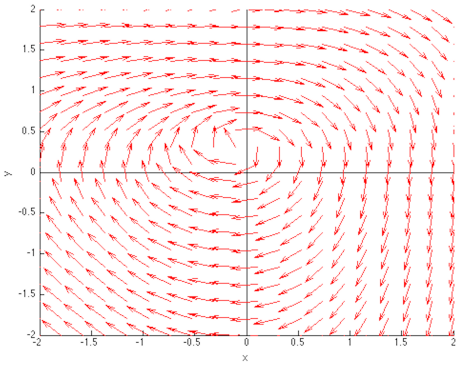
A critical point of $G(x, y)$ and a global maximum is $(x, y) = \left(\ln(\frac{1}{\beta}(\alpha - \frac{1}{2}\alpha^2 D_\xi - \frac{\gamma m}{k\beta})), \ln(\frac{m}{k\beta})\right)$ and it is found by solving $\frac{\partial G(x,y)}{\partial x} = 0$ and $\frac{\partial G(x,y)}{\partial y} = 0$. This equilibrium point is close to the equilibrium point found in (2.9) if D_ξ is small.



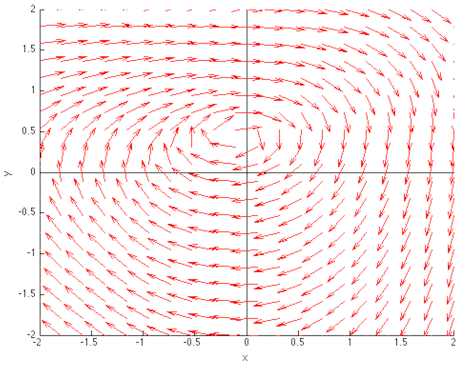
(A) Without harvesting, $\ln(T) = 0$.



(B) With harvesting, $\ln(T) = 0$.



(C) With harvesting, $\ln(T) = -1$.



(D) With harvesting, $\ln(T) = -1$ and $h = 2$.

FIGURE 2.2: Phase diagram for (2.8) with $\alpha = 1$, $\beta = 1$, $\gamma = 1$, $m = 1$ and $k = 1$.

From Figure 2.2 we can see that introducing harvesting does not affect the dynamics of the system: There is still only one real-valued equilibrium point which is a stable spiral point. We also notice that by changing the harvesting threshold to $\ln(T) = -1$ the position of the equilibrium point is shifted towards a higher population of the preys. This is not surprising as harvesting of the predators will favour the population of the preys. By increasing the maximal harvesting rate h, more harvesting of the predators is allowed and hence an increase in the preys population is expected. This is confirmed in Figure 2.2d where the equilibrium point is shifted further in the same direction as it was when changing the harvesting threshold.

Chapter 3

Monte Carlo Methods

Suppose that we want to compute the probability of a "straight" in a hand of four cards that are randomly dealt from a deck of 52 cards. This could be solved analytically, but a more straightforward approach would be to deal a large number of hands and to estimate the probability of a straight with the fraction of the hands on which straights where dealt. This approach would take hours or maybe even days to complete in practice, but with the use of computer power we could simulate this situation in seconds.

To improve the computation time we will introduce Monte Carlo simulation which does several, and often millions, of independent computations. It is sometimes straightforward to have different processors compute different values, and then use an appropriate (weighted) average of these values to produce a final answer. In this way, the communication between processors is minimised, so that parallel processing is easily facilitated. Certain Monte Carlo algorithms are so ideally suited to parallel computation that they would be labeled as "embarrassingly parallelisable".

3.1 Introduction

The expression *Monte Carlo method* (MC) is very general. They are a class of computational techniques based on synthetically generating random variables to deduce the implication of the probability distribution. The MC techniques are often the only practical way to evaluate difficult integrals or to sample random variables governed by

complicated probability density functions. The methods rely on repeated random sampling to obtain the distribution of an unknown probabilistic entity. The methods vary, but most of them follow a particular pattern

1. Define a domain for the possible inputs.
2. Generate random inputs from a probability distribution over the domain.
3. Perform a deterministic computation of the inputs.
4. Repeat for N trials.
5. Use the N trials to study the distribution of the statistics.

3.1.1 Random Numbers

Most of the Monte Carlo sampling techniques require a *random number generator*, which generates uniform statically independent values on the half open interval $[0, 1)$. Different software libraries offers different random number generators such as CERNLIB and CURAND. All of the algorithms produce a periodic sequence of numbers, and in order to obtain effectively random number one must therefore not use more than a small subset of a single period. A short period is therefore a problem for many of the random number generators. Some of the simulation methods do not always require truly random numbers to be useful and most of the useful techniques use deterministic, pseudorandom sequences which makes it easy to test and re-run simulations. A quality that is necessary to make good simulations is for the pseudorandom numbers to appear "random enough" in a certain sense.

The most common means of generating uniform random variables on a computer is to use mathematical algorithms. The most widely used algorithms are *linear congruential generators*. These algorithms follow recursions of the form

$$u_{i+1} = (au_i + b) \bmod (m),$$

where

- u_0 is the initial seed.

- a is called the *multiplier*.
- b is called the *increment*.
- m is called the *modulus*.

Given this recursion we obtain our random numbers via

$$U_i = \frac{u_i}{n}.$$

The sequences U_i can not be truly random. The hope is that if a, b and m are carefully chosen, U_i can behave like a truly random uniform sequence.

We will use the C library function *rand()* which is based upon repeated addition and shift to generate pseudo-random numbers and the CUDA library function *cuRAND*. *cuRAND* uses the XORWOW algorithm which provides fast and simple random number generators that seem to do very well on tests of randomness.

3.1.2 Monte Carlo Method [6]

The goal is to compute $\alpha = \mathrm{E}[X]$, where X is a random variable that can be generated in finite time on a computer. By generating independent and identically distributed copies $X_1, X_2, \ldots, X_n$ of the random variable X, α can be found via

$$\alpha_n = \frac{1}{n}(X_1 + \cdots + X_n).$$

In order to prove convergence of α we need to introduce some theorems and definitions.

Definition 3.1. Let $(Z_n : 1 \leq n \geq \infty)$ be a sequence of random variables. Then Z_n converges in probability to Z_∞ as $n \to \infty$ if for each $\epsilon > 0$,

$$p\{|Z_n - Z_\infty| > \epsilon\} \to 0$$

as $n \to \infty$.

3.1.2.1 Weak Law of Large Numbers (WLLN)

Let $X_1, X_2, \ldots$ be an independent and identically distributed sequence of random variables for which $\mathrm{E}[X_1] < \infty$. Then

$$\frac{1}{n}(X_1 + \cdots + X_n) \xrightarrow{P} \mathrm{E}[X_1]$$

as $n \to \infty$. Hence, the WLLN establishes that $\alpha_n \xrightarrow{P} \alpha$ as $n \to \infty$ and hence that the Monte Carlo method converges.

Theorem 3.1. *(Markov's inequality) Let W be a non-negative random variable. Then,*

$$P(W > w) \leq \frac{E[W]}{w}.$$

Proof.

$$P(W > w) = \mathrm{E}[I(W > w)].$$

since $W/w \geq 1$ on $\{W > w\}$ it follows that

$$\begin{aligned} \mathrm{E}[I(W > w)] &\leq \mathrm{E}[\frac{W}{w}]I(W > w) \\ &\leq \frac{\mathrm{E}[W]}{w}. \end{aligned}$$

□

Theorem 3.2. *(Chebyshev's inequality) Let Γ be a random variable for which var$(\Gamma) < \infty$. Then*

$$P\{|\Gamma - E[\Gamma]| > \epsilon\} \leq \frac{var(\Gamma)}{\epsilon^2}.$$

Proof. Use Markov's Inequality and put $W = (\Gamma - \mathrm{E}[\gamma])^2$ and $w = \epsilon^2$. □

We are now ready to prove the WLLN when $\mathrm{var}(X_1) < \infty$.

Proof. (of the WLLN) Let $\Gamma = X_1 + \cdots + X_n$. Using Chebychev's Inequality on γ yields

$$P\{|\Gamma - \mathrm{E}[\Gamma]| > n\epsilon\} \leq \frac{\mathrm{var}(\Gamma)}{n^2\epsilon^2}.$$

Rewriting this we get

$$P\left\{\left|\frac{\Gamma}{n} - \mathrm{E}[X_1]\right| > \epsilon\right\} \leq \frac{\mathrm{var}(X_1)}{n\epsilon^2}.$$

Next we use the fact that $\mathrm{var}(X_1) < \infty$, and take the limit as $n \to \infty$:

$$P\{\left|\frac{X_1 + \cdots + X_n}{n} - \mathrm{E}[X_1]\right| > \epsilon\} \leq \frac{\mathrm{var}(X_1)}{n\epsilon^2} \to 0.$$

□

By using the strong law of large numbers the convergence of the Monte Carlo method can be shown in a stronger sense.

3.1.2.2 The Strong Law of Large Numbers (SLLN)

Definition 3.2. Let Z_n, $1 \leq n \leq \infty$ be a sequence of random variables. We say that Z_n converges almost surely to Z_∞ as $n \to \infty$ if $P\{A\} = 1$, where A is the event defined by

$$A = \{\omega : Z_n(\omega) \to Z_\infty(\omega) \text{ as } n \to \infty\}.$$

Theorem 3.3. *(SLLN) Let $X_1, X_2, \ldots$ be an independent and identically distributed sequence of random variables for which $[E] < \infty$. Then*

$$\frac{1}{n}(X_1 + \cdots + X_n) \xrightarrow{a.s.} E[X_1]$$

as $n \to \infty$. Hence, the SLLN implies that the Monte Carlo method converges almost surely as it follows that $\alpha_n \xrightarrow{a.s.} \alpha$ as $n \to \infty$.

The SLLN implies the WLLN and it is a more sophisticated result.

3.1.2.3 Rate of Convergence

To study the rate of convergence of the Monte Carlo method we need the central limit theorem (CLT). We will first introduce the definition of *convergence in distribution*, or equivalent, *weak convergence*, before introducing the theorem.

Definition 3.3. Let Z_n, $1 \leq n \leq \infty$, be a sequence of random variables. We say that Z_n *converges in distribution* to Z_∞ as $n \to \infty$ if

$$\lim_{n\to\infty} P\{Z_n \leq z\} \to P\{Z_\infty \leq z\}$$

at each continuity point z of $P\{Z_\infty \leq \cdot\}$.

Theorem 3.4. *(Central Limit Theorem) Let X_n, $n \geq 1$, be a sequence of independent and identically distributed random variables with $0 < var(X_1) = \sigma^2 < \infty$. Then the following holds*

$$\lim_{n\to\infty} \frac{X_1 \dots X_n - nE[X_1]}{\sqrt{n}\sigma} \Rightarrow N(0,1),$$

where $N(0,1)$ is a normal random variable with mean zero and a unit variance.

Proof. We put

$$\tilde{X}_i = \frac{X_i - \mathrm{E}[X_1]}{\sigma},$$

where $X_1, X_2, \dots$ is a sequence of independent and identically distributed random variables with $0 \leq \sigma^2 = \mathrm{var}(x) < \infty$. We note that $\mathrm{E}[\tilde{X}_i] = 0$ and $\mathrm{var}(\tilde{X}_i) = \mathrm{E}[\tilde{X}_i{}^2] = 1$. This implies that

$$\mathrm{E}\Big[e^{i\theta\tilde{X}_i}\Big] = 1 - \frac{\theta^2}{2} + O(\theta^2)$$

as $\theta \to \infty$. Hence

$$\mathrm{E}\Big[e^{i\theta\frac{X_1+\dots+X_n-n\mathrm{E}[X_1]}{\sqrt{n}\sigma}}\Big] = \mathrm{E}\Big[e^{i\theta\sum_{j=1}^n \frac{\tilde{X}_j}{\sqrt{n}}}\Big] = \mathrm{E}\Big[\prod_{j=1}^n e^{i\theta\frac{\tilde{X}_j}{\sqrt{n}}}\Big].$$

Since the expectations are independent and the distributions are identical we get

$$\begin{aligned} &= \prod_{j=1}^n \mathrm{E}\Big[e^{i\theta\frac{\tilde{X}_j}{\sqrt{n}}}\Big] = \Big(\mathrm{E}\Big[e^{i\theta\frac{\tilde{X}_1}{\sqrt{n}}}\Big]\Big)^n \\ &= \Big(1 - \frac{\theta^2}{2n} + O(\frac{1}{n})\Big)^n \to e^{-\frac{\theta^2}{2}}. \end{aligned}$$

Next we note that for a $N(0,1)$ random variable, $e^{-\frac{\theta^2}{2}}$ is the characteristic function and hence

$$\lim_{n\to\infty} \frac{X_1 + \dots + X_n - n\mathrm{E}[X_1]}{\sqrt{n}\sigma} \Rightarrow N(0,1).$$

□

The CLT implies that

$$\lim_{n\to\infty} n^{\frac{1}{2}}(\alpha_n - \alpha) \Rightarrow \sigma N(0,1).$$

This tells us that the rate of convergence for the Monte Carlo method is $n^{-\frac{1}{2}}$, so if the computations requires high accuracy, the method is not suitable.

3.1.2.4 Error estimates

As we proved earlier, the rate of convergence of the Monte Carlo method is slow and the error estimate is therefore of interest. We return to the CLT and assume that our goal is to compute $\alpha = \mathrm{E}[X]$, where $0 < \sigma^2 = \mathrm{var}(x) < \infty$. Suppose that we run n independent experiments and thereby generating independent and identically distributed copies $X_1, X_2, \ldots, X_n$. Our estimator becomes

$$\alpha_n = \frac{1}{n}(X_1 + \cdots + X_n).$$

Using the CLT we get that for any $x \geq 0$,

$$\lim_{n\to\infty} P\Big\{ -z \leq \frac{n^{\frac{1}{2}}}{\sigma}(\alpha_n - \alpha) \leq z\Big\} \rightarrow P\{-z \leq N(0,1) \leq z\}.$$

If we choose z such that $P\{-z \leq N(0,1) \leq z\} = 1 - \delta$. The event given by

$$\Big\{ -z \leq \frac{n^{\frac{1}{2}}}{\sigma}(\alpha_n - \alpha) \leq z\Big\}$$

is identical to

$$\Big\{\alpha \in \Big[\alpha_n - \frac{\sigma z}{\sqrt{n}}, \alpha_n + \frac{\sigma z}{\sqrt{n}}\Big]\Big\},$$

which implies that we may conclude that

$$\lim_{n\to\infty} P\Big\{\alpha \in \Big[\alpha_n - \frac{\sigma z}{\sqrt{n}}, \alpha_n + \frac{\sigma z}{\sqrt{n}}\Big]\Big\} \Rightarrow 1 - \delta.$$

This implies that the random interval $\Big[\alpha_n - \frac{\sigma z}{\sqrt{n}}, \alpha_n + \frac{\sigma z}{\sqrt{n}}\Big]$ contains the parameter α with a probability of $1 - \delta$. We call the interval a $100(1-\delta)\%$ confidence interval for α.

Thus, the error estimates associated with Monte Carlo computations have their basis in the statistical notion of confidence intervals.

As σ^2 is rarely known a priori we can use the estimate

$$s_n = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \alpha_n)^2}$$

which gives us the following

$$\lim_{n\to\infty} P\Big\{\alpha \in \Big[\alpha_n - \frac{zs_n}{\sqrt{n}}, \alpha_n + \frac{zs_n}{\sqrt{n}}\Big]\Big\} \to 1 - \delta.$$

3.1.3 Kinetic Monte Carlo Method [7]

The kinetic Monte Carlo (KMC) method is a Monte Carlo method which is intended to simulate the time evolution of some processes which typically occur with a given known rate in nature. These rates are the input to the KMC algorithm and the method itself cannot predict them. The algorithm reads as follows

1. Set the time $t = 0$.
2. Form a list of all possible rates in the system k_i.
3. Calculate the cumulative function $K_i = \sum_{j=1}^{i} k_j$ for $i = 1, \ldots, N$, where N is te total number of transitions.
4. Get a random uniform number, u, from the half open interval $(0, 1]$.
5. Find the i for which $K_{i-1} < uK_N \leq K_i$. This is the event to carry out i.
6. Carry out event i.
7. Get a new uniform number $u' \in (0, 1]$.
8. Increase the time with $\Delta t = K_N^{-1} \ln(\frac{1}{u'})$, i.e $t = t + \Delta t$.
9. Recalculate all rates k_i which may have changed due to the transitions. If appropriate, remove or add new transitions i. Update N and the list of events accordingly.
10. Return to step 2.

Example 3.1. *We want to solve the simplified version of the Lotka-Volterra equations given in (2.2). We ignore the term which governs self-limitation in the growth rate of the prey population size in the absence of predators. The system of differential equations now reads*

$$\begin{aligned} \dot{U} &= -mU + k\beta UV = U(-m + k\beta V) = f(U,V), \\ \dot{V} &= \alpha V[1 + h(t)] - \beta UV = V(\alpha(1 + h(t)) - \beta U) = G(U,V). \end{aligned} \tag{3.1}$$

We let $h(t)$ be the zero-mean "physical" Gaussian white noise given in (2.3). The procedure is as follows

1. *Catalog all the rates k_i for system i, where k_i is the transition of the system from i to j.*
2. *Partial sum the rates, $s_j = \sum_{i=1}^{j} k_i, j = 1 \dots n$.*
3. *Choose a uniform random variable u from $(0,1)$ and find j such that $s_{j-1} < us_n \leq s_j$.*
4. *Update all k_i for state j.*
5. *Choose a uniform random variable b from $(0,1)$ and calculate the time step given by $\Delta t = -\ln(b)/s_n$.*
6. *Repeat from step one till the required time span is covered.*

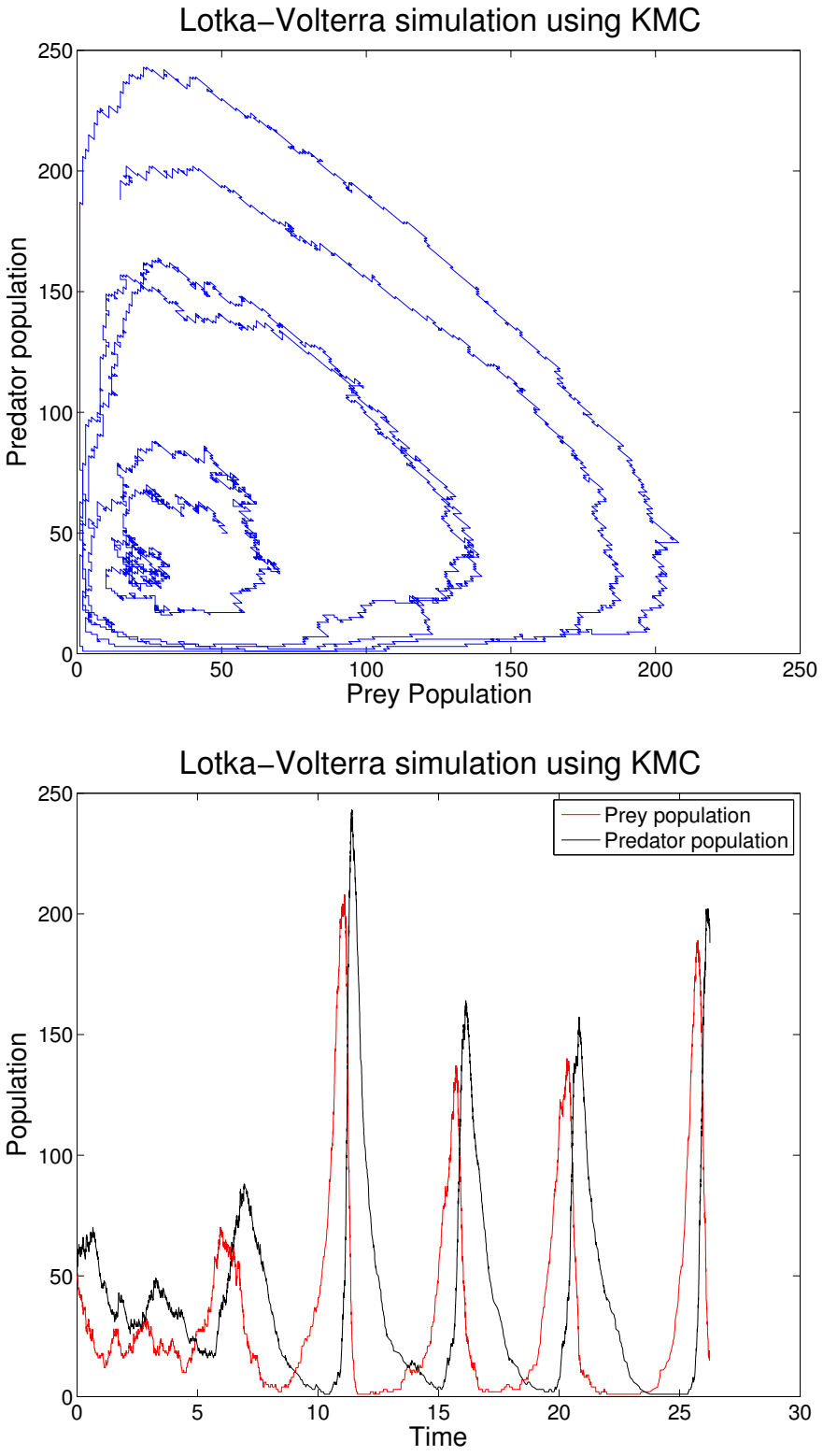


FIGURE 3.1: Simulation of the simplified Lotka-Volterra model using the KMC algorithm. The number of time steps used is 5000, $\alpha = 2$, $\beta = 0.05$, $\gamma = 1.5$ and the initial prey and predator populations are 50.

To locate the critical points of (3.1) *we solve* $\dot{U} = 0$ *and* $\dot{V} = 0$

$$\begin{aligned} U(-m + k\beta V) &= 0 \\ V(\alpha(1 + h(t)) - \beta U) &= 0 \\ \implies U = \frac{\alpha(1 + h(t))}{\beta}, \quad V = \frac{m}{k\beta} \quad & \text{and } U = 0, \quad V = 0, \end{aligned}$$

where the first equilibrium point is co-existence and the latter equilibrium point is extinction of both the species [18] .

Definition 3.4. The stability matrix, also called the *community matrix* is given by

$$A(U^*, V^*) = \begin{pmatrix} \partial_U f & \partial_V f \\ \partial_U g & \partial_V g \end{pmatrix},$$

where (U^*, V^*) is the equilibrium point.

Theorem 3.5. *The steady state solutions* U^* *and* V^* *are stable if the trace and the determinant of the community matrix are negative and positive respectively.*

Proof. Recall that the characteristic polynomial of a square matrix A is defined to be

$$p(\lambda) = \mathrm{Det}(A - \lambda I).$$

For the 2×2 matrix A given by

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \tag{3.2}$$

we have

$$p(\lambda) = \begin{vmatrix} a - \lambda & b \\ c & d - \lambda \end{vmatrix} = \lambda^2 - (a + d)\lambda + (ad - bc).$$

If we now recall that the trace of a matrix is defined to be the sum of its diagonal elements, we get that $\mathrm{Tr}(A) = a + d$ and $\mathrm{Det}(A) = ad - bc$. Introducing these to the equation of the characteristic polynomial yields $p(\lambda) = \lambda_A^2 - \mathrm{Tr}(A)\lambda + \mathrm{Det}(A)$. The eigenvalues are the roots of $p(\lambda)$, so the quadratic formula immediately gives us that the eigenvalues will be real if and only if the discriminant $\mathrm{Tr}(A)^2 - 4\,\mathrm{Det}(A) > 0$. The conditions for the signs in the case of real eigenvalues are as follows

- If $\mathrm{Det}(A) < 0$, the eigenvalues are real and of opposite sign and the phase portrait is a saddle, which is always unstable.
- If $0 < D < T^2/4$, the eigenvalues are real, distinct and of the same sign. The phase portrait is a stable node if $T < 0$ and an unstable node if $T > 0$.
- If $0 < T^2/4 < D$, the eigenvalues are neither real nor purely imaginary and the phase portrait is a stable spiral if $T < 0$ and an unstable spiral if $T > 0$.

□

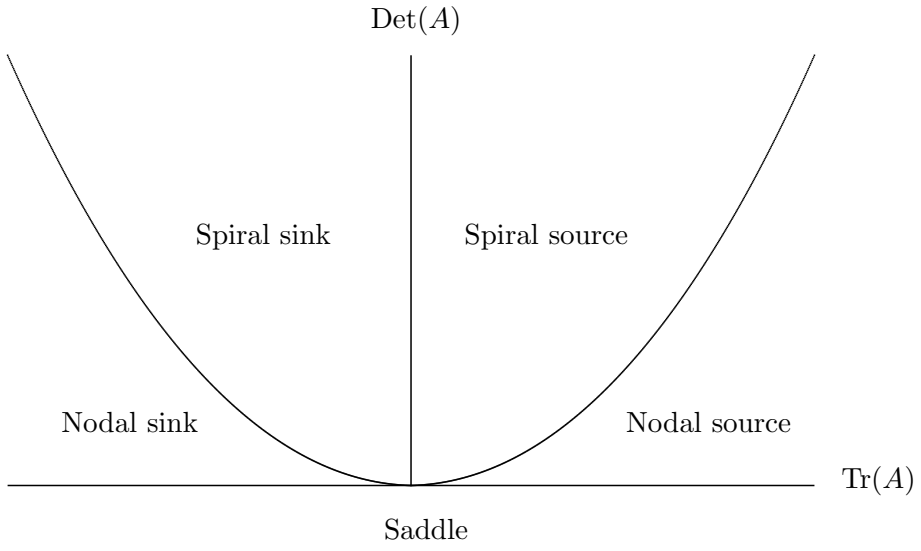


FIGURE 3.2: The trace-determinant diagram.

Example 3.2. *(3.1 cont.) To determine the stability of the equilibrium points we need to study the community matrix.*

$$A(U^*, V^*) = \begin{pmatrix} -m + k\beta V & k\beta U \\ -\beta V & \alpha - \beta U \end{pmatrix}.$$

The trace and the determinant becomes

$$\mathrm{Det}(A(U^*, V^*)) = (-m + k\beta V^*)(\alpha - \beta U^*) + \beta^2 k V^* U^*,$$
$$\mathrm{Tr}(A(U^*, V^*)) = -m + k\beta V^* + \alpha - \beta U^*.$$

Let us start with the stability analysis of the trivial solution of $U^* = 0$ and $V^* = 0$, which means extinction of both species. One has $\mathrm{Tr}(A(0,0)) = -m + \alpha$ and $\mathrm{Det}(A(0,0)) =$

$-m\alpha$. *Since* $m > 0$ *and* $\alpha > 0$, $\mathrm{Det}(A(0,0)) < 0$ *so extinction of both species never occurs in our model.*

The semi-trivial solutions are given by $U^* = 1$ *and* $V^* = 0$ *or* $U^* = 0$ *and* $V^* = 1$ *and they mean that one of the species are extinguished. We start with the extinction of species one, one has* $\mathrm{Tr}(A(0,1)) = -m + \alpha + k\beta$ *and* $\mathrm{Det}(A(0,1)) = (-m + k\beta)\alpha$. *For these solutions to be stable, it is necessary that* $k\beta > m$ *regardless of the value of* α. *A similar analysis with the species two extinction lead us to conclude that the solution is stable for* $\alpha < \beta$ *regardless of the value of* m.

The non-trivial solution $(U^*, V^*) = (\frac{\alpha}{\beta}, \frac{m}{k\beta})$ *leads to* $\mathrm{Det}(A(U^*, V^*)) = \alpha m$ *and* $\mathrm{Tr}(A(U^*, V^*)) = 0$, *so the solution is stable as* α *and* m *are both positive. The system of equations given in* (3.1) *are separable which means that the system can be solved by direct integration. Dividing the first equation by the second equation we find the solution*

$$\begin{aligned}
\frac{\frac{dU}{dt}}{\frac{dV}{dt}} &= \frac{U(-m + k\beta V)}{V(\alpha - \beta V)} \\
\frac{dU}{dV} &= \frac{U}{(\alpha - \beta U)} \frac{V}{(-m + k\beta V)} \\
&\implies \int \frac{\alpha}{U} - \beta dU = \int k\beta - \frac{m}{V} dV \\
C &= \alpha \ln(U) - \beta U + m \ln(V) - k\beta V,
\end{aligned}$$

where C *is a constant.*

The level sets for $z = \alpha \ln(U) - \beta U + m \ln(V) - k\beta V$ *is given in 3.3*

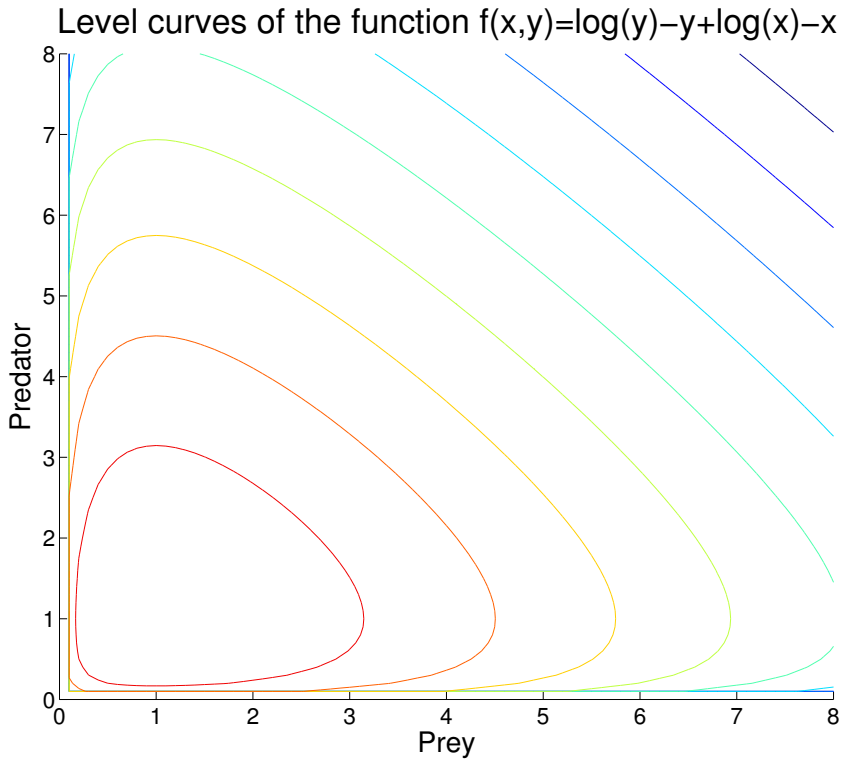


FIGURE 3.3: Predator-Prey dynamics as described by the level curves of a conserved quantity. The values used for the constants are $\alpha = \beta = k = m = 1$. There are equilibrium points at $U = 0$, $V = 0$ and $U = 1$,$V = 1$.

Chapter 4

Parallel Computing

The speed of computation plays a vital role in many different fields of science and parallel computing is often necessary for a program to complete within a required period of time. For example in financial markets, a tenth of a second can mean hundreds of thousands of dollars.

Parallel computing operates on the principle that large problems often can be divided into smaller problems, which can be solved concurrently. In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem

4.1 Introduction [8] [9]

When the GPU was created, its main purpose was to improve the graphic experience in game consoles. In the beginning, the functionality was limited to accelerating the memory-intensive work of texture mapping and rendering polygons, but within this wealthy industry there was a demand for powerful and inexpensive hardware that was highly parallel and could offer high data throughput. The programmable shader was introduced in the early 2000's, giving the user more control and abilities to manipulate data and the GPU obtained more and more CPU-like capabilities. These new capabilities made it possible to program algorithms for non-graphical applications on the GPU.

The reason why a GPU can achieve high performance is due to its parallel structure. In contrast to CPUs, a GPU consists of hundreds, even thousands of multi-processors.

Although the performance of each multi-processor is usually inferior to CPU, the overall performance can outreach it. However, in order to get this performance improvement the algorithm need to be parallel. We call an algorithm parallel if it contains segments that are independent.

Traditionally, to solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed sequentially on a computer, only one instruction at a time.

Parallel computing, on the other hand, uses many parallel processing streams to solve a problem. This is realised by breaking the problem into parallel parts where each part of the algorithm can be executed simultaneously and often independently.

The advantage of parallel programming over serial computing is increased computing performance. It allows one to solve problems that don't fit on a single CPU and problems that can't be solved in a reasonable amount of time. By using parallel computing we can solve larger problems, the same problem faster or we can solve more cases of the same problem. The three main techniques to improve parallel performance are reducing latency, increasing throughput and reducing CPU power consumption. These three factors are often interrelated so the total efficiency is maximised when they are balanced. Amdahl's law and Gustafsons-Barsis' Law reflects two different goals for optimising.

4.1.1 Terminology

Parallel computer programs are more difficult to write than sequential programs, because concurrency introduces new classes of potential software bugs, of which *race conditions* are the most common. Race conditions [19] arise when an application depends on the sequence or timing of processes or threads for it to operate properly. We call a race condition critical when the result is an invalid execution or contains bugs, and non-critical when the results contains unanticipated behaviour. Race conditions have a reputation of being difficult to reproduce and debug, since the end result is nondeterministic and depends on the relative timing between interfering threads.

Example 4.1. *Suppose that we have P processes and that each process j generates a sequence $\{X_i^j : 0 \leq i < n\}$. Suppose next that we want our final sequence to be the sum of the P sequences at each point i. One way to solve this is to let each process add*

their X_i to the i'th position in the final sequence, which will minimise the total memory requirement. Each process performs a read followed by a write and a problem arises as the value in the i'th position can be changed by another process between the read and the write. This is a typical example of a race condition. The correct way of doing this is by letting each process generate the sequence and write it to it's own memory space before one of the processes calculates the average over each of the i'th positions. This will require $(P-1)n$ time more memory than the first approach and the efficiency of the program will decrease drastically, but it is necessary to avoid a race condition.

Serial

$\sum_{j=0}^{P} X_1^j$	$\sum_{j=0}^{P} X_2^j$	$\sum_{j=0}^{P} X_3^j$	$\cdots$	$\sum_{j=0}^{P} X_n^j$

Parallel

X_1^1	X_2^1	X_3^1	$\cdots$	X_n^1
X_1^2	X_2^2	X_3^2	$\cdots$	X_n^2
$\vdots$		$\cdots$		
X_1^P	X_2^p	X_3^P	$\cdots$	X_n^P

FIGURE 4.1: The figure illustrates that the parallel version requires $(P-1)n$ times more memory that the serial version in order to avoid race conditions.

The time it takes to complete a task is called *latency*. The scale can be anywhere from nanoseconds to years, but it has units of time. The lower latency the better optimised is the program.

Throughput is the term for at which rate a series of tasks can be completed. It has units of work per unit time and the larger throughput the better. When referring to memory or communication transactions, a related term is *bandwidth* which refers to throughput rates that have a frequency-domain interpretation.

Throughput and latency are related so that an optimisation that improves throughput may increase the latency.

4.1.2 Speedup, Efficiency and Scalability

Two important metrics related to performance and parallelism are speedup and efficiency.

The speedup of a program is defined by the following formula

$$S_N = \frac{T_1}{T_N}, \tag{4.1}$$

where

- S is the speedup.
- T_1 is the latency of the program with one processor.
- T_N is the latency of the program running with N processors.

The ideal speedup is called linear and it occurs when $S = N$. Linear speedup is rare in practice, since there are extra work involved in distributing work to processors and coordinating them. When the speedup exceeds the number of processes we call it super-linear and when the number of processes exceeds the speedup we call it sub-linear.

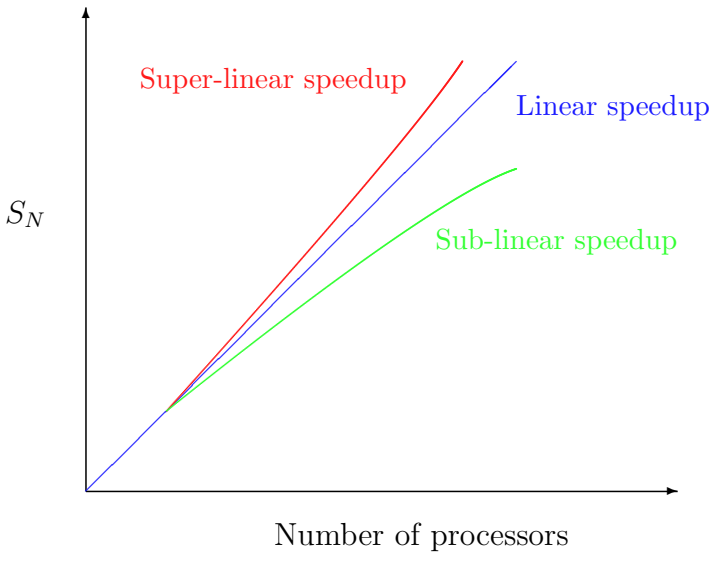


FIGURE 4.2

As communication among different processes only can be done through message passing, the overhead communication will often be a benchmark for where increasing the number of processes will not affect the runtime in the same manner as before.

The efficiency of a program is defined by

$$E = \frac{S_N}{N} = \frac{T_1}{NT_N}, \tag{4.2}$$

and it measures return on hardware investment. The ideal efficiency is 1, which corresponds to linear speedup.

4.1.3 Amdahl's Law [8]

Amdahl's law reflects how the program run faster with the same workload. It predicts the maximum expected improvement of a program using multiple processors as the speedup is limited by the sequential fraction of the program. Amdahl argued that the execution time, T, of a program falls into two categories

- Time spent doing non-parallelisable serial work, W_{serial}.
- Time spent doing parallelisable work, $W_{parallel}$.

These gives us the following times for sequential and parallel execution

$$T_1 = W_{serial} + W_{parallel}$$
$$T_N \geq W_{serial} + \frac{W_{parallel}}{N},$$

where N is the number of processors.

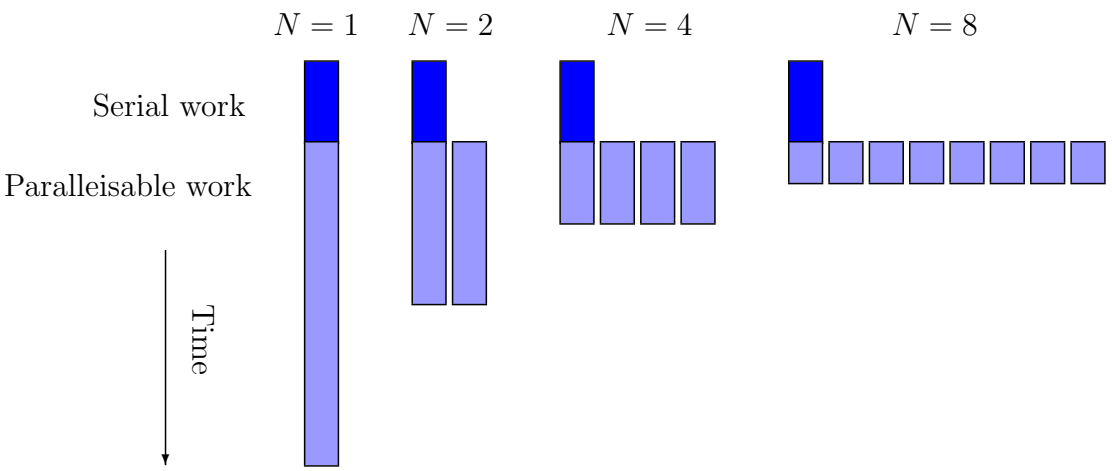


FIGURE 4.3: In Amdahl's law, speedup is limited by the non-parallelisable serial portion of the code.

If we let f denote the non-parallelisable serial fraction and p the parallelisable fraction of the total work, the following hold

$$W_{serial} = fT_1$$
$$W_{parallel} = pT_1.$$

Plugging these relations into the definition of speedup yields Amdahl's Law

$$S_N = \frac{1}{1 - p + \frac{p}{N}}.$$

If we let the number of parallel threads increase, we will get the limit of the speedup

$$\lim_{N \to \infty} S_N = \frac{1}{1-p} = \frac{1}{f} = S_\infty.$$

This implies that it is always sufficient to have a finite number of parallel threads to achieve good performance as the speedup is limited by the fraction of the work that is not parallelisable. If for example the parallelisable portion of the code is 0.90 the speedup can reach 10, while for algorithms with a parallelisable portion of 0.99 the speedup is limited by 100. It is therefore noteworthy that the sequential percentage of the algorithm have a severe impact on the speedup of the algorithm.

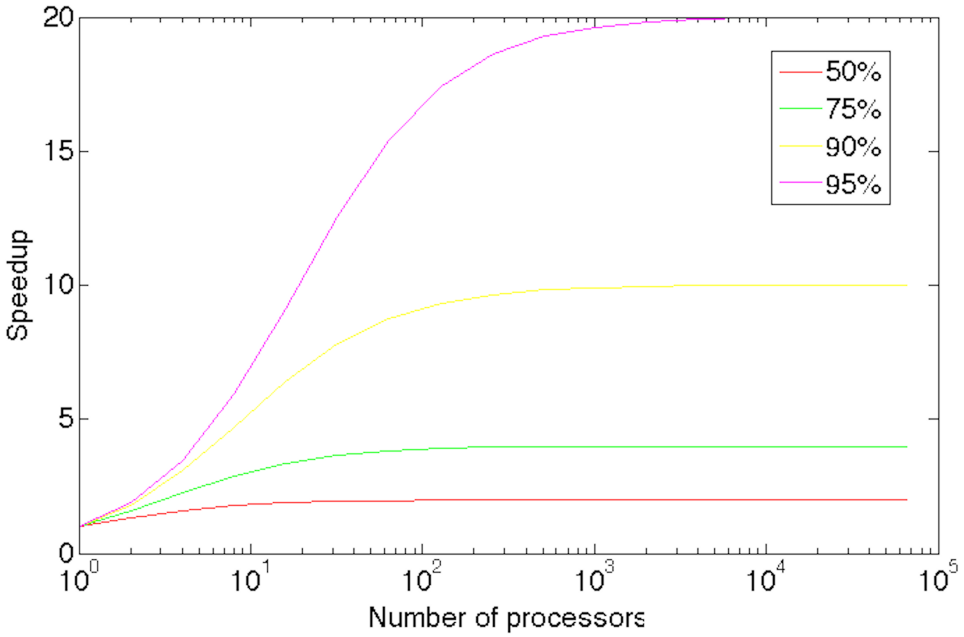


FIGURE 4.4: The figure shows the effect of multiple processors on speedup for different percentages of the parallel portion of the code.

The reality is even worse than predicted by Amdahl's law due to load balancing, scheduling, cost of communications and input/output.

4.1.4 Gustafson-Barsis' law [8]

John Gustafsons suggested that the speedup should be measured by scaling the problem to the number of processors rather than by fixing the problem size. Experience indicates that as computers get new capabilities, applications change to exploit these features which contradicts Amdahl's laws view on programs as fixed and the computer as changeable.

Gustafson-Barsis' law is based upon that as computers get more powerful the problem sizes grow and as the problem sizes grow, the work required for the parallel part of the problem frequently grows much faster than the serial part. This implies that as the problem size grows the serial fraction decreases and hence the speedup improves.

Gustafson-Barsis' law is given by

$$S_N = N - f(N-1).$$

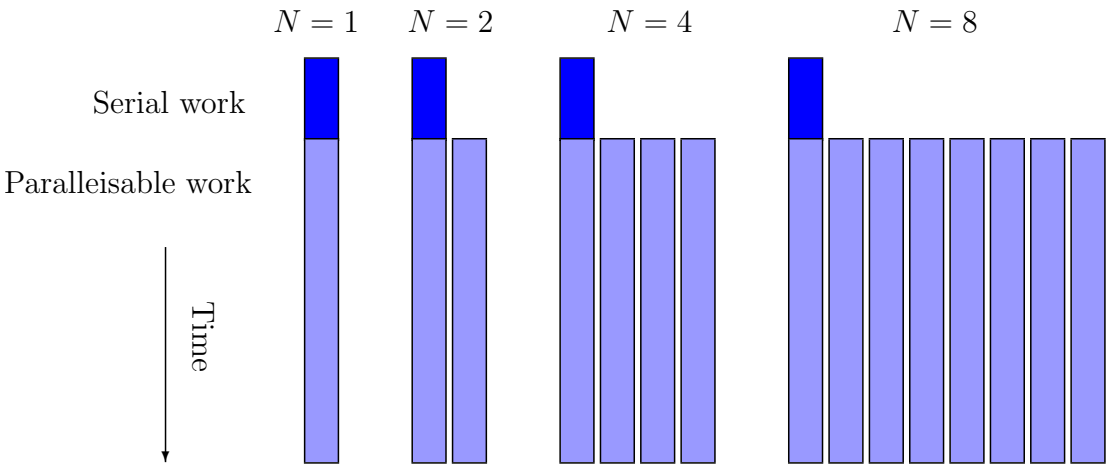
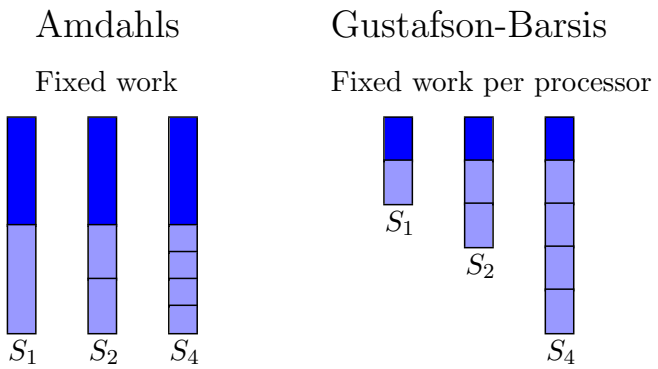


FIGURE 4.5: Gustafson-Barsis's law states that if the problem size increases with N while the serial portion grows slowly or remains fixed, speedup grows as processes are added.

The difference between these two laws lies in whether you want your program to run faster with the same workload or to run in the same time with a larger workload.

Example 4.2. *We compare Amdahl's law and Gustafson-Barsis's law for a code with a parallelisable fraction of 0.5.*



$$S_N = \frac{1}{1 - p + \frac{p}{N}} \qquad S_N = N - f(N-1)$$

$$S_2 = \frac{1}{1 - 0.5 + \frac{0.5}{2}} = 1.33 \qquad S_N = 2 - 0.5(2-1) = 1.5$$

$$S_4 = \frac{1}{1 - 0.5 + \frac{0.5}{4}} = 1.6 \qquad S_N = 4 - 0.5(4-1) = 2.5$$

FIGURE 4.6: Comparison of Amdahl's and Gustafson-Barsis's law

Concurrent programming languages, libraries, APIs, and parallel programming models have been created for programming parallel computers. These can generally be divided into classes based on the assumptions they make about the underlying memory architectures. We will now introduce the parallel programming techniques used to make the Monte Carlo algorithm more time efficient.

4.2 Memory

The main memory in a parallel computer is either shared memory or distributed memory and the largest and fastest computers in the world today exploit both shared and distributed memory architectures. When writing parallel programs the memory layout is extremely important and it is crucial to handle memory with care to achieve good performance.

The amount of memory required can be greater for parallel codes than serial codes, due to the need to replicate data and for overheads associated with parallel support libraries

and subsystems. Communication and synchronisation between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance. For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation. The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.

4.2.0.1 Data Structure Alignment

Data structure alignment is the way data is arranged and accessed in computer memory. When a computer performs a read or a write to a memory address, it will be done in chunks of size s. The system's performance can be increased by putting the data at a memory address equal to some multiple of s due to the way the CPU handles memory. This is what we refer to as *data alignment*. *Data structure padding* means inserting some meaningless bytes between the end of the last data structure and the start of the next in order to get the data aligned.

Example 4.3. *Suppose the computer's size is* 4 *bytes and we want to read a* 4 *byte message from memory. The data to be read should be at a memory address which is some multiple of* 4*. If the data starts at address* 10 *the computer has to read two chunks from memory. If the data had been correctly aligned, the data's address had started at* 12*, and only one chunk had to be read from memory.*

4.2.1 Shared Memory

In a shared memory model tasks share a common address space, which they asynchronously can read and write to.

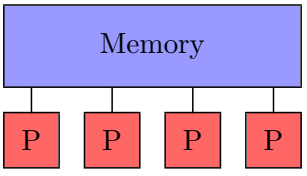


FIGURE 4.7: All processors have access to a pool of shared memory.

The shared memory component can be a shared memory machine and/or GPUs. A *thread*, short for *thread of execution* are a way for a program to divide itself into two or more simultaneously running tasks. In general a thread is contained inside a process and different threads within the same process share the same resources. Threads communicate with each other through global memory. This requires synchronisation constructs to ensure that shared data structures is only modified by one thread at a time. Two of the most widely used shared memory APIs are POSIX Threads and OpenMP.

4.2.1.1 OpenMP [10]

OpenMP is a set of compiler directives and library routines used for multithreaded parallel processing. It is a method of parallelising where a master thread forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

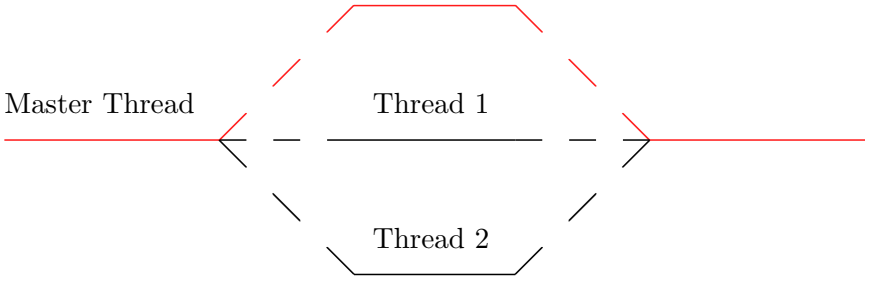


FIGURE 4.8: The master thread forks a specified number of threads. Each thread performs the task given before they assemble.

4.2.2 Distributed Memory

The distributed memory component is the network of multiple shared memory/GPU machines, in which each processor has its own private memory.

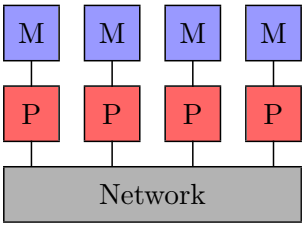


FIGURE 4.9: Memory is local to each processor.

Each processor works on its section of the problem and they can only operate on local data. If remote data is required, the processes must communicate to move data from one machine to another. The communication is often referred to as message passing over a network and the most widely used message-passing API is the Message Passing Interface (MPI).

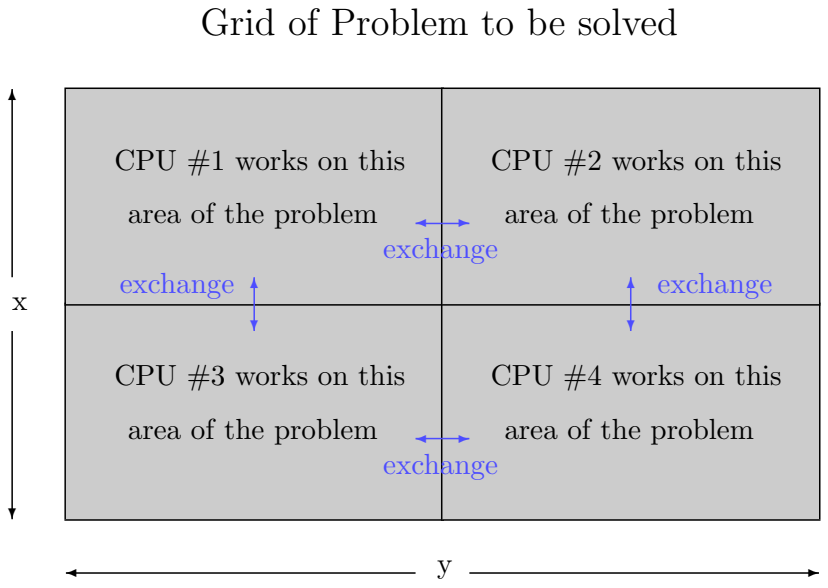


FIGURE 4.10: The figure shows how a grid of problem can be solved using 4 CPUs.

4.2.2.1 MPI [11]

MPI is a message passing paradigm of parallelism. You have one root process which spawns programs among all the processes within the same communicator. The root process is often referred to as the *master process*. All the threads in the system is independent and hence the only way of communication between them is through message

passing over the network. The network bandwidth and throughput is therefore one of the most crucial factor in MPI implementation's performance.

When setting up the MPI environment a communicator is formed around all of the processes that where spawned. The foundation of communication is built upon send and receive operations among processes. Each process has it's own *rank* and we often refer to the process with rank zero as the root or the master. A process may send a message to another process by providing the rank of the process and a unique tag to identify the message. The receiver can then post a receive for a message with a given tag, and then handle the data accordingly. The communication operations are divided into two blocks: collective and point-to-point communications. A collective communication involves communication among all processes in a process group (often all processes in the communicator) whereas a point-to-point communication involves communication between two specific processes.

When running a MPI program you can choose how many processes you want your program to run on and if your code is well optimised the runtime will decrease as the number of processes increase. Most of the computers and laptops contains 2 or 4 processors, which in many cases is not enough. The use of servers is therefore sometimes needed.

4.2.3 CUDA [12] [13]

For our GPU implementation we target NVIDIA GPUs with its programming platform named CUDA. CUDA is a Simultaneous multithreading paradigm of parallelism. It uses state of the art graphics processing unit (GPU) architecture to provide parallelism. A GPU contains blocks of set of cores working on same instruction in a lock-step fashion. Hence, if all the threads in a system do a lot of identical work, you can benefit performance wise by using CUDA.

In CUDA, the *host* refers to the CPU and its memory, while the *device* refers to the GPU and its memory. We are dealing with *heterogeneous computing* which refers to a system that uses more than one kind of processors. Parallel portions of an application are executed on the device as *kernels*. The kernel executes the same code on a large batch of parallel threads.

CUDA organizes a parallel computation using the abstractions of threads, blocks and grids. These can be defined as follows

Thread An execution of a kernel with a given index. Each thread uses its index to access elements in an array such that all of the threads cooperatively processes the entire data set. It allows programs to transparently scale to different GPUs.

Warp A warp is a group of 32 threads. There is a limit to the number of warps that can be achieved on each streaming multiprocessor. This limit depends on the device properties.

Block A block is a group of threads. The dimensions of the different blocks must be equal and coordination among the threads can be achieved by synchronisation which makes a thread stop at a certain point until all the other threads in the same block reach the same point.

Grid A grid is a group of blocks. It is not possible to obtain synchronisation among the different blocks.

When running a CUDA program you have to specify the grid and block dimensions for the kernels. It is difficult to determine which combination of dimensions that will give the best performance, and the best way to find it is by trial and error.

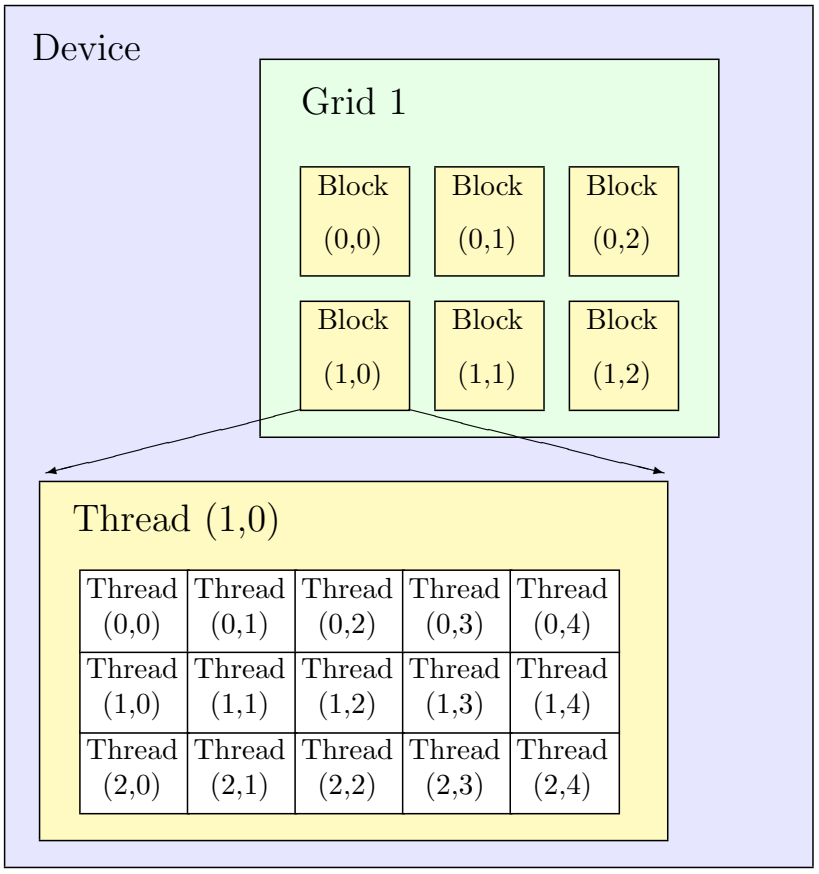


FIGURE 4.11: CUDA thread model. Each kernel is assigned a grid. Each grid contains a number of blocks. Each block contains threads (512 maximum per block).

When using CUDA it is extremely important to handle memory with care. There are several different CUDA memory types and the choice of which one to use will affect the performance of the program.

Global Memory The global memory is slow compared to the other memory types. It requires sequential and aligned (coalesced) read and writes to be fast. A coalesced memory transaction is one in which all of the threads in a half-warp access global memory at the same time. The correct way to do it is to have consecutive threads access consecutive memory addresses.

Texture Memory The texture memory is read only which means that the data in the memory cannot be edited directly, only read by the threads. Texture memory is written by the host, and read by the device.

Constant Memory Constant memory is used for data that will not change over the course of a kernel execution. In some situations, using constant memory rather than global memory will reduce the required memory bandwidth.

Shared Memory Shared Memory is fast, but it is subject to bank conflicts[19]. Shared memory arrays are subdivided into smaller subarrays called banks, where different banks can be accessed simultaneously. If two or more addresses of a memory request are in the same bank, the access is serialised and this is what is called a bank conflict. Threads within the same block have two main ways to communicate data with each other. The fastest way would be to use shared memory. If you only need the threads to share a small amount of data at any given time, using shared memory is by far the fastest and most convenient way to do it.

4.2.4 Hybrid

A combination of the different programming models is called a hybrid model. Two popular examples of a hybrid model is using MPI with CUDA and MPI with OpenMP.

4.3 Parallel Monte Carlo

If we want to compute some unknown quantity μ, a standard Monte Carlo would tell us to obtain some large number n of samples, $X_1, \ldots, X_n$, where each random variables have mean m and variance v. The next step will be to estimate μ by the estimator $E = \frac{1}{n}\sum_{i=1}^{n} X_i$. This estimator would have bias $m - \mu$, and variance $\frac{v}{n}$.

If we now have k processors available, the simplest idea is to run the same program as described in the previous section on each of the k processors. Each processor j then produces n samples, compute their average E_j, and reports the result back to the master processor. The master processor then average these k results to obtain the final result

$$\bar{E} = \frac{1}{k}\sum_{j=1}^{k} E_j.$$

This final result will have mean m and variance $\frac{v}{kn}$. The result is equivalent to the result we would have obtained by running our program on a single processor for k times as long and we have thus obtained a linear speed-up.

We want to make two parallel versions of the algorithm produced in Example 3.1, one using MPI and the other using CUDA.

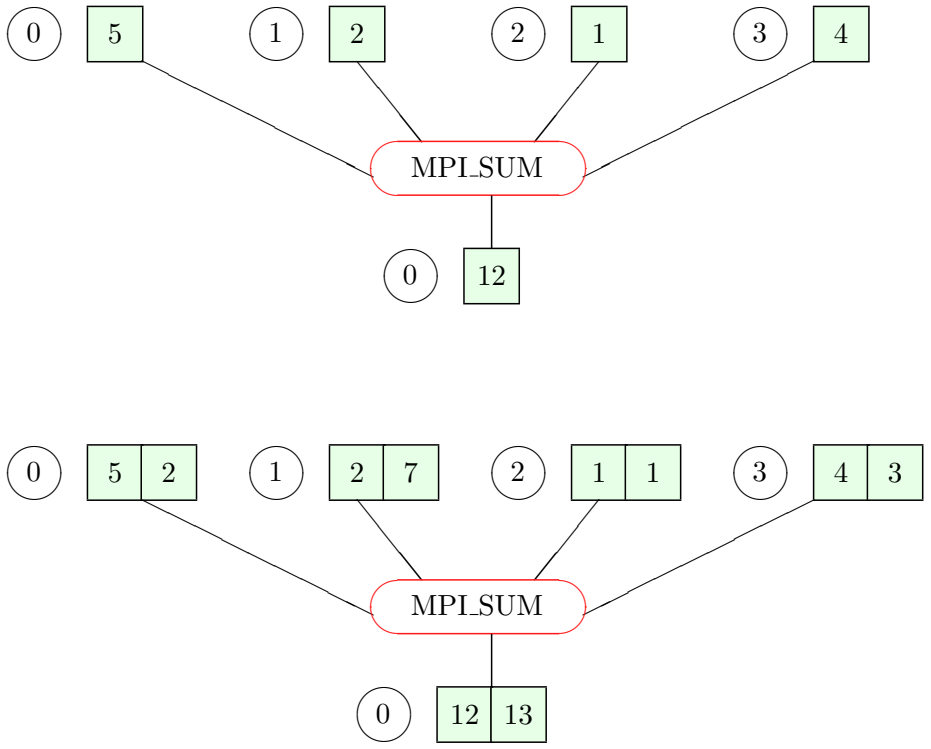
4.3.1 MPI

We let each process be responsible of propagating the same initial distribution over the same time interval n independent times. Each process finds the average of their results and sends it back to the master process by using the MPI library function MPI_Reduce. This function takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result. The prototype of the function looks like this:

```
MPI_Reduce(
        void* send_data,
        void* recv_data,
        int count,
        MPI_Datatype datatype,
        MPI_Op op,
        int root,
        MPI_Comm communicator)
```

The send_data parameter is an array of elements of type *datatype* that each process wants to reduce. The recv_data is only relevant on the process with a rank of *root* and contains the reduced result. The length of the array is *count*. The *op* parameter is the operation that you wish to apply to your data. Custom reduction operations can be defined, but MPI contains a set of communication reduction operations that can be used. Among these we find MPI_MAX, MPI_MIN and MPI_SUM.

Below is an illustration of the communication pattern of MPI_Reduce where we want to sum up an array of integers from each process.



We compare the run times of the two programs by letting the serial program propagate n independent distributions and by running the parallel program for different number of processors. The results are shown in Table 4.1.

Number of simulations	Serial (s)	$n = 2$ (s)	$n = 4$ (s)	$n = 8$ (s)	$n = 16$ (s)	$n = 32$ (s)
2000	3.934607	2.002540	1.032375	0.553108	0.516247	0.294940
4 000	7.874334	3.979395	2.024673	1.043744	1.018425	0.762433
8 000	15.666488	7.926271	3.974366	2.023714	1.315159	0.762604
16 000	31.395772	15.706708	7.914551	3.986790	3.067251	2.021594
32 000	62.192365	31.469908	15.797379	7.963596	7.733702	6.024957
40 000	82.526232	38.398987	20.975954	10.412540	7.636600	7.013166

TABLE 4.1: The runtime of the serial and the parallel versions of the code. The number of time steps used is 5000, $\alpha = 2$, $\beta = 0.05$, $\gamma = 1.5$ and the initial prey and predator populations are 50. Each combination of processors and number of independent simulations are run ten times and the results given are the average over the ten runs.

One of the biggest impact on MPI-programs performance is communication and the MC methods do not require a lot of communication. Hence the MC methods are well suited for the Message Passing Interface.

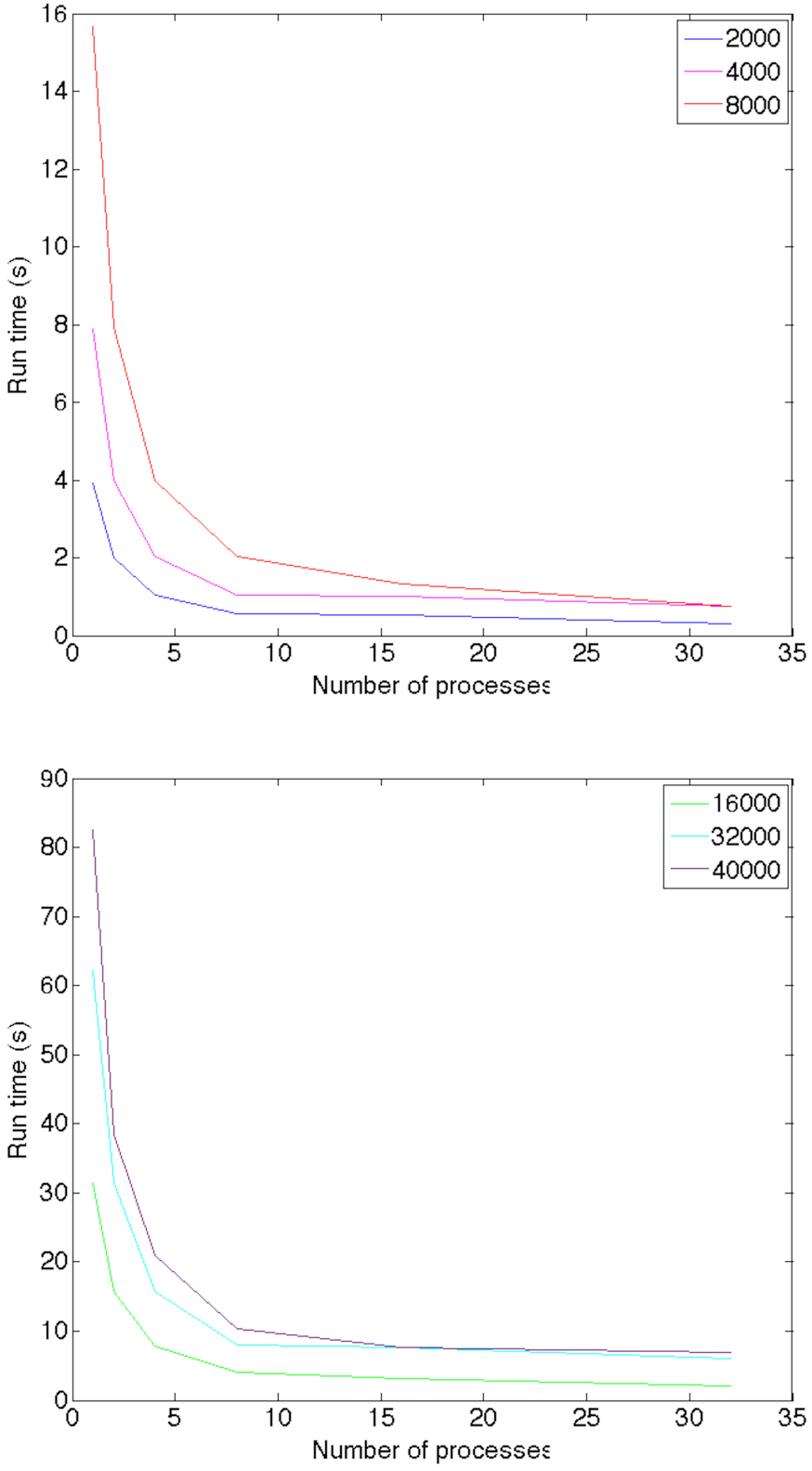


FIGURE 4.12: The two figures show the runtime of the program with different combinations of number of simulations and number of processes.

From Table 4.1, Figure 4.12 and Figure 4.13 we can conclude that we have almost achieved a linear speedup. The delay in the parallel program is due to the communication which has to be done between all the processes and the root process at the end of each run. The more processes we use and the larger number of simulations we have, the communication which has to be done increases. We can also notice that when the number of processes exceeds 8, the linearity vanishes and we do not get the same effect on the run time as we did for the smaller numbers of processes.

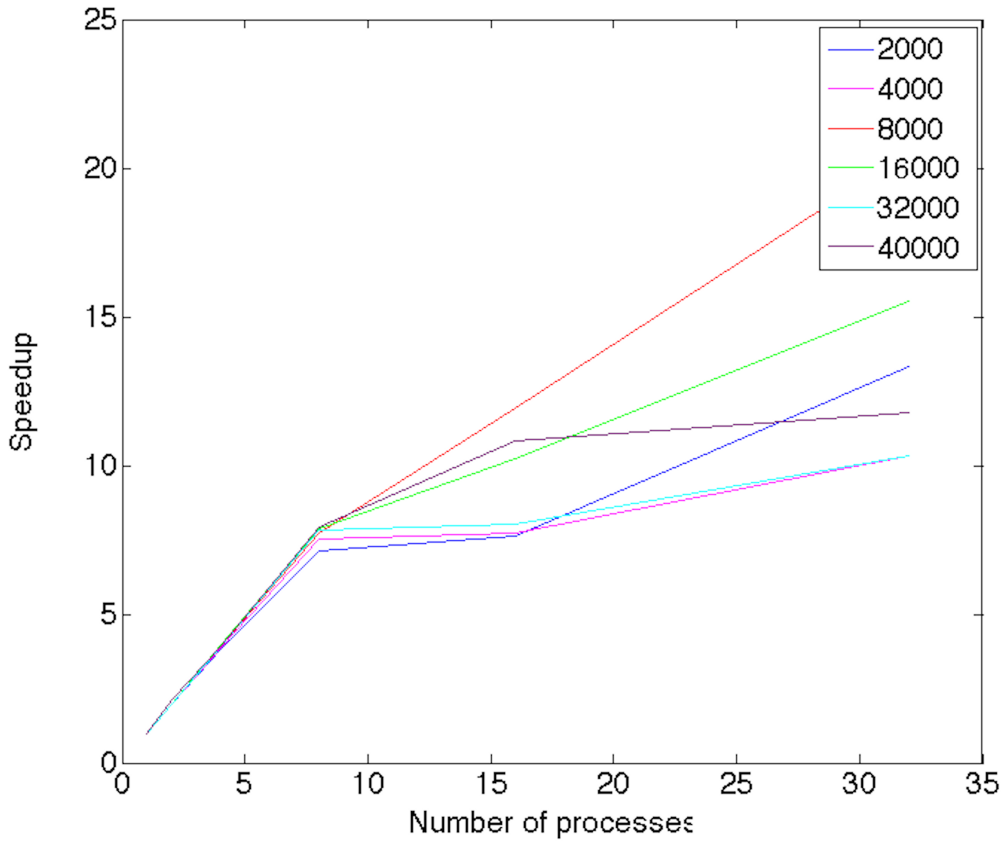


FIGURE 4.13: The figure shows the speedup of the program with different combinations of number of simulations and number of processes. The formula used to calculate the speedup is given by (4.1).

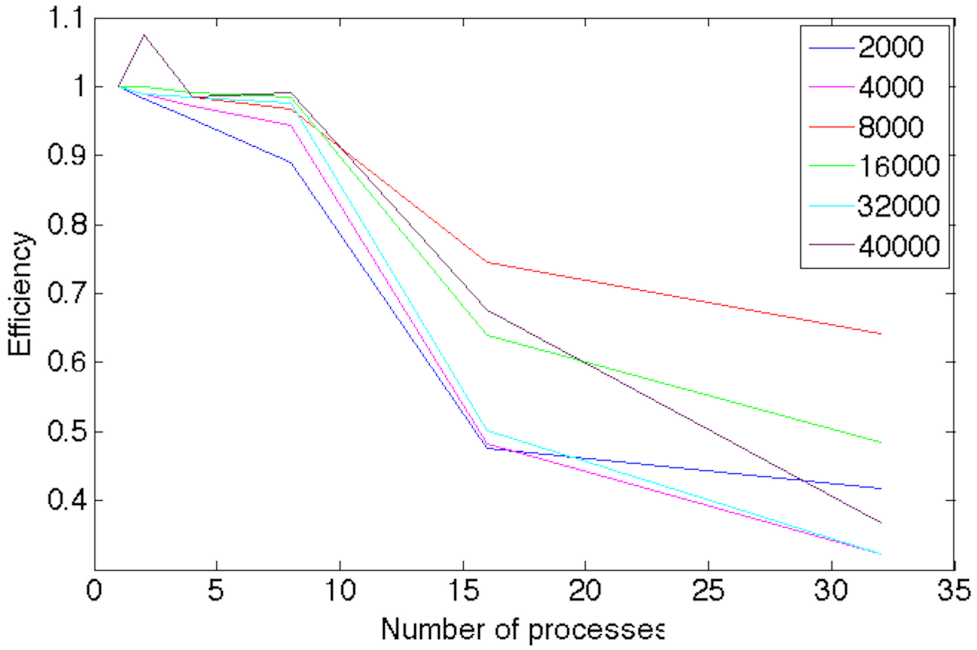


FIGURE 4.14: The figure shows the efficiency of the program with different combinations of problem number of simulations and number of processes. The formula used to calculate the efficiency is given by (4.2).

From Figure 4.14 we can see that the efficiency decreases as the number of processes increases. This is expected due to the extra work involved in distributing work to the different processes and the work coordinating them. We concluded from Figure 4.13 that the linearity in the speedup vanishes when we exceed 8 processes and this coincide with Figure 4.14 as the efficiency drops below 1 at this point.

4.3.2 CUDA

The next attempt is to make the code parallel by using CUDA. We let each thread do one propagation of the initial distribution and then we find the average of the distributions within each block before returning to the host. To do this we make use of shared memory which is faster than global memory. A profitable way of performing computation on the device is therefore by partitioning the data into subsets that fit into shared memory. Each thread block handles each data subset by loading the subset from global memory to shared memory using multiple threads to exploit memory-level parallelism. Each thread block then performs the necessary computations before copying the results back

to global memory. As we saw in Example 4.1.2 the amount of shared data will be a lot to handle for the GPU and will therefore affect the performance of the program.

We encountered some problems when trying to write this parallel code. The first and biggest problem is that the program runs out of shared memory. With the same number of time steps as used in the MPI version we run out of memory with only one thread in each block. One solution to this problem turned out to be removing the use of shared memory and using global memory instead. Global memory is normally slower than shared memory, but we concluded that this was the best way to fix the problem.

Number of threads	Number of blocks	Number of simulations	Run time (s)
32	16	512	0.117434
32	32	1 024	0.287328
64	64	4 096	1.106318
128	64	8 192	2.202157
256	128	32 768	13.570726
512	64	32 768	13.508272
1 024	32	32 768	13.269728
2 048	16	32 768	13.465909
4 096	8	32 768	13.113456
8 192	4	32 768	13.344003
16 384	2	32 768	13.433822

TABLE 4.2: The runtime of the CUDA versions of the code. The number of time steps used is 5000, $\alpha = 2$, $\beta = 0.05$, $\gamma = 1.5$ and the initial prey and predator populations are 50. Each combination of number of threads and blocks are run ten times and the results given are the average over the ten runs.

We ran out of memory when trying to run the code for a number of simulations bigger than 32768. We can conclude from Table 4.2 that the best runtime for the biggest number of simulations is achieved when we use the combination of 8 blocks with 4096 threads in each block.

4.4 Results

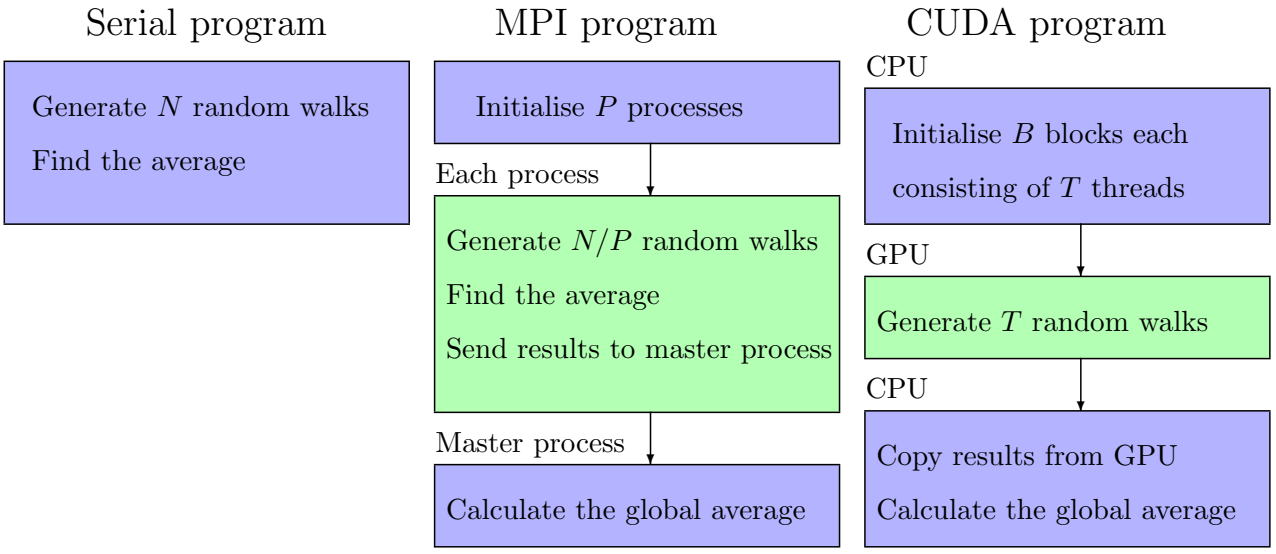


FIGURE 4.15: The program flow of the different programs. The blue boxes are serial work and the green boxes are work done in parallel.

4.4.1 CUDA VS MPI

It is not straightforward to compare CUDA and MPI as they are two different paradigm of parallelism. The best way of comparing the two methods is by comparing the best possible runtime for the two versions for the same number of simulations. When running a CUDA program the number of threads and number of blocks are normally set to be a power of two. Hence the runtimes of the CUDA version is for approximate the given number of simulation.

Number of simulations	MPI (s)	CUDA (s)
4 000	0.762433	1.106318
8 000	0.762604	2.202157
16 000	6.024957	13.113456

TABLE 4.3: Comparison of the runtime of the two parallel versions of the KMC algorithm.

We can conclude from Table 4.3 that the CUDA program is slower than the MPI program. The main reason for this is that in the MPI program only one averaging over all

the processes has to be preformed, whereas in the CUDA program the averaging is done on the CPU with no use of parallelism. The CUDA program therefore requires more memory space and more computations has to be performed.

The MPI program is run on NTNU's cluster Kongull, which is a CentOS 5.3 Linux cluster running Rocks. The cluster has 113 nodes consisting of 4 I/O and 108 compute nodes.

4.4.2 Accuracy

To determine the accuracy of the KMC method we return to the simplified LV model

$$\begin{aligned} \dot{V} &= \alpha V - \beta VU \\ \dot{U} &= \beta UV - \gamma U. \end{aligned} \tag{4.3}$$

Eliminating time from the two differential equations above yields

$$\frac{dU}{dV} = \frac{U(\beta V - \gamma)}{V(\alpha - \beta U)},$$

which leads to

$$\frac{dU}{U}(\alpha - \beta U) = \frac{dV}{V}(\beta V - \gamma),$$

whose solutions are closed curves. Integrating both sides of the above equation gives us the solution

$$C = -\beta V + \gamma \ln(V) - \beta U + \alpha \ln(U),$$

where C is a constant which is conserved on each curve. The constant depends only on initial conditions and not on time.

We use MPI to solve 4.3 and compare the results to the solution given above for different numbers of simulations. If we let $\tilde{U}$ and $\tilde{V}$ be the estimated population sizes of the predators and the preys, respectively, we define the error, e, to be

$$e = \max_{i \in [0,n]} |C - \big(- \beta \tilde{V}_i + \gamma \ln(\tilde{V}_i) - \beta \tilde{U}_i + \alpha \ln(\tilde{U}_i)\big)|.$$

The ideal situation would be that the error tends to zero as the number of simulations increases.

Number of simulations	Error
100	1.6566
1000	1.4067
5 000	1.3829
10 000	1.3555
20 000	1.3453
30 000	1.3304
40 000	1.3229
50 000	1.3197
60 000	1.3150
70 000	1.3149
80 000	1.3107
90 000	1.3105
100 000	1.3102

TABLE 4.4: The table shows the error for the KMC method for the given number of simulations. The initial populations are 50, $\alpha = 2$, $\beta = 0.05$, $\gamma = 1.5$ and the number of time steps is 5000.

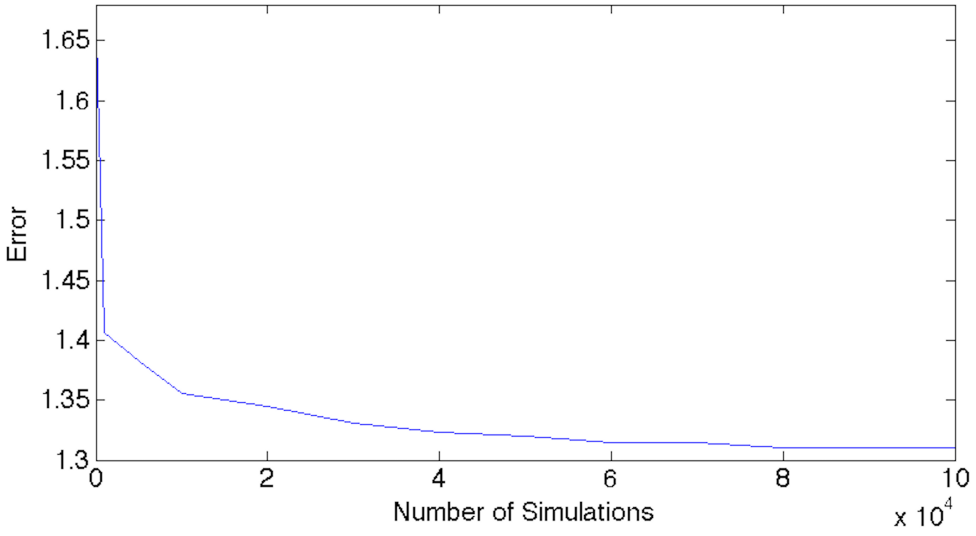


FIGURE 4.16: The figure shows the accuracy of the KMC method for the given number of simulations.

We can conclude from Table 4.4 and Figure 4.16 that the error decreases as the number of simulations increases. We would have hoped that the error tended more rapidly towards zero.

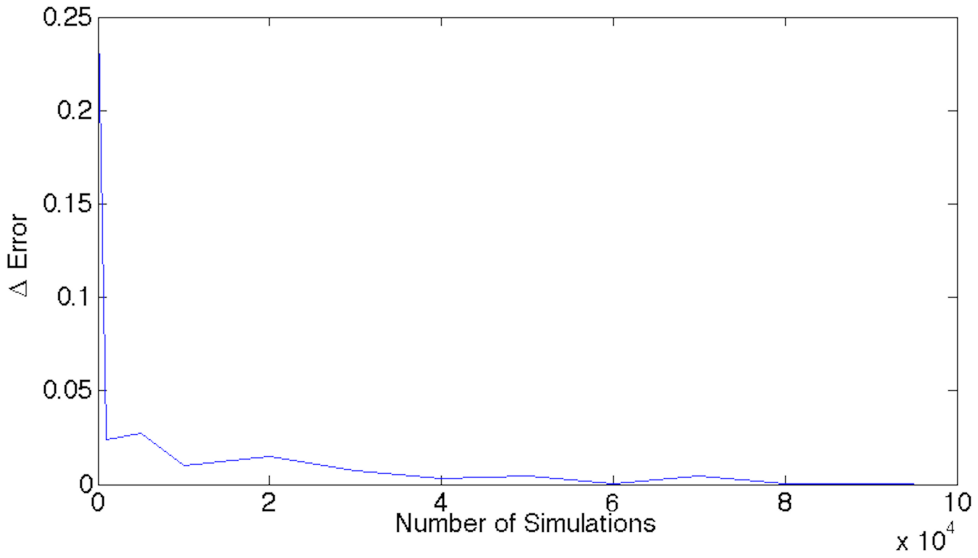


FIGURE 4.17: The figure shows how the difference in the accuracy decreases as the number of simulations increases.

We can see from Figure 4.17 that the difference in the accuracy tends towards zero as the number of simulations increases. The KMC method will therefore not achieve a significantly better accuracy by increasing the number of simulations above 100000. As we saw earlier, the run time for the serial KMC program was 82.526232 seconds for 40000 simulations. Hence, in order to achieve the best accuracy in a reasonable amount of time, the use of parallel programming is necessary.

In order to determine whether or not the KMC algorithm described in this report is a good option to the path integration method, the accuracy of the path integration method needs to be studied. The path integration method is described and an algorithm is given in [16] and [20]. As the algorithm given in these two master theses where not described in detail we where not able to get some proper accuracy results when running the code.

Chapter 5

Conclusion

The main purpose of this work was to develop a more time efficient solution to the Lotka-Volterra model. To compare runtimes, in addition to the serial version, the algorithm was implemented using the Message Passing Interface as well as on a GPU using CUDA. By comparing the best runtime for the three versions we concluded that the KMC method is best suited for the message passing interface. We have studied the accuracy of the KMC method and it turned out that there was a limit for which increasing the number of simulations did not affect the accuracy significantly. In order to reach this limit in a reasonable amount of time the use of parallel programming is essential.

5.1 Summary

5.1.1 Performance

In the MPI version we assigned an equal amount of work to each process which means that the program will be load balanced. The MC method used, turned out to be optimal for parallel programming as the computations performed are independent.

As is common for GPU implementation we assigned one random walk to each thread. Due to the problem we encountered while writing this code we believe that there is still considerable potential for optimising the GPU program.

5.2 Further Research

The topics of further research can be divided into two main categories; code optimisation and extending the model.

5.2.1 Code Optimisation

In the MPI code the efficiency can be improved by the use of MPI-IO. MPI-IO lets the processes access files all together at the same time which allows read and write operations to be performed in parallel. As the program works now, the writing to a text file is done by only one process and we can hence expect an increase in the efficiency by making this parallel.

When writing the GPU code we found several issues. The main issue was that each thread block required too much shared data. This problem proved to be difficult to solve. The first attempt to solve this was to reduce the number of threads in each block. This attempt did not fix the problem. To fix the problem the use of global memory was introduced. Global memory is normally slower than shared memory so with respect to code efficiency we first suggest looking at returning to shared memory. To reduce the amount of shared memory required by the program one may consider a hybrid solution combining MPI and CUDA. In our code we are taking the average over all the results from the threads on the CPU. A more efficient way of doing this, would have been to divide this work among the threads. The averaging takes about 95% of the runtime so changing this will definitively have a positive effect on the performance. We have chosen to let each thread do one propagation of the initial distributions. By assigning multiple propagation to each thread and let each thread find the average over its own results before writing the results to global memory, we can reduce the amount of global memory needed and the program will hence be able to run for bigger problem sizes.

To determine whether or not the KMC method is a good alternative to the path integration method the difference in the accuracy between these two methods needs to be studied. Some further research of the accuracy of the path integration method is therefore needed to draw a conclusion to this question.

5.2.2 Extend the Model

There are several versions of the Lotka-Volterra model, some more difficult to solve than others. The code could have been fitted better to the real life situation by including more variables which are affecting the two species as well as including other species that they interact with. The more dimensions your problem has the harder it is to make the code parallel and the focus has therefore only been on the 2D problem. In conclusion there are many possibilities to expand the underlying mathematical model.

Appendix A

Source code

```
/*This is the source code used to perform the Lotka-Volterra
 simulation using the time algorithm of KMC .
 */

#include <stdlib.h>
#include <stdio.h>

#include <math.h>
#include <stdbool.h>

#include <time.h>
#include <sys/time.h>

// Print the total time used by the program
void print_time(struct timeval start, struct timeval end){
    long int ms = ((end.tv_sec * 1000000 + end.tv_usec) -
                   (start.tv_sec * 1000000 + start.tv_usec));
    double s = ms/1e6;
    printf("Time : %f s\n", s);
}

double uniform_distribution(int rangeLow, int rangeHigh)
{
    double myRand = rand()/(1.0 + RAND_MAX);
    return myRand;
}
```

```
int* find_geq(double* array, double condition, int size)
{
    int* output = (int*)malloc(sizeof(double)*size);
    int pointer = 0;
    for (int i = 0; i < size; i++)
    {
        output[i] = 0;
        if (array[i] >= condition)
        {
            output[pointer] = i;
            pointer ++;
        }
    }
    return output;
}

double* cumsum(double* k, int size)
{
    double* output = malloc(sizeof(double)*size);
    output[0] = k[0];
    for (int i = 1; i < size; i++)
    {
        output[i] = output[i-1]+k[i];
    }
    return output;
}

void KMC(int Nt, double alpha, double beta, double gamma, int prey_init,
         int predator_init, int runs)
{
    double* T = malloc(sizeof(double)*(Nt+1));
    double* prey = malloc(sizeof(double)*(Nt+1));
    double* predator = malloc(sizeof(double)*(Nt+1));

    for (int i = 0; i < Nt+1; i++){
        T[i] = 0;
        predator[i] = 0;
        prey[i] = 0;
    }

    double k[3];
```

```
double* s;
double u;
int* index;

int x,y;

for (int j=0; j<runs; j++)
{
    x = prey_init;
    y = predator_init;
    prey[0] = x;
    predator[0] = y;
    T[0]=0;

    double current_time = 0;

    for (int i = 0; i < Nt; i++) {
        k[0] = alpha*x;
        k[1] = beta*x*y;
        k[2] = gamma*y;
        s = cumsum(k, 3);
        u = uniform_distribution(0,1);
        index = find_geq(s, s[2]*u, 3);
        switch (index[0]) {
            case 0:
                x = x+1;
                break;
            case 1:
                if (x > 1)
                {
                    x = x-1;
                    y = y+1;
                }
                break;
            default:
                if (y>1)
                    y = y-1;
        }
        prey[i+1] += x;
        predator[i+1] += y;
        double dt = -log(1-u)/s[2];
```

```
            current_time += dt;
            T[i+1] = current_time;
        }
    }
    for (int i=0; i<Nt; i++){
        prey[i] = prey[i]/runs;
        predator[i] = predator[i]/runs;
        T[i] = T[i]/runs;
    }


    FILE *file = fopen("KMC.txt", "w");
    if (file == NULL)
    {
        printf("Error opening file!\n");
        exit(1);
    }
    for (unsigned int j = 0; j < Nt; j ++) {
        fprintf(file, "%f %f %f \n", T[j] , prey[j], predator[j]);
    }
    fclose(file);
}

int main(int argc, char **argv) {
    /*
     argv[1] = Number of time steps
     argv[2] = Alpha, the rate constant of prey reproduction
     argv[3] = Beta, the rate constant of prey death and predator
  reproduction
     argv[4] = Gamma, the rate constant of predator death
     argv[5] = Initial prey population
     argv[6] = Initial predator population
     argv[7] = Number of independent simulations
     */
    if (argc !=8)

        printf("Need 6 inputs: Nt,
               alpha,
               beta,
               gamma,
               initial prey and initial predator populations,
```

```
                number of simulations for each process \n");

    struct timeval start, end;

    gettimeofday(&start, NULL);

    KMC(atoi(argv[1]), atof(argv[2]), atof(argv[3]), atof(argv[4]), atoi(
   argv[5]),
        atoi(argv[6]), atoi(argv[7]));

    gettimeofday(&end, NULL);
    print_time(start, end);

    exit(0);
}
```

code/KMC_serial

```
/*
 This source code performs the Lotka-Volterra Predator-Prey simulation
 using the residence time algorithm of KMC and the Message Passing
   Interface is used
 to parallelise the code. The Lotka-Volterra model for a 2 species (prey-
   predator)
 system is given by

 dx/dt = alpha*x-beta*x*y-H(y)
 dy/dt = k*beta*x*y-m*y

 */

#include <stdlib.h>
#include <stdio.h>

#include <math.h>
#include <stdbool.h>

#include <time.h>
#include <sys/time.h>

#include <mpi.h>
```

```
// Global variables
int rank,                       // MPI rank
    size,                       // Number of MPI processes
    dims[2],                    // Dimensions of MPI grid
    coords[2],                  // Coordinate of this rank in MPI grid
    periods[2] = {0,0};         // Periodicity of grid

MPI_Comm cart_comm;             // Cartesian communicator

// MPI initialization, setting up cartesian communicator
void init_mpi(int argc, char** argv){

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Dims_create(size, 2, dims);
    MPI_Cart_create( MPI_COMM_WORLD, 2, dims, periods, 0, &cart_comm );
    MPI_Cart_coords( cart_comm, rank, 2, coords );
}


// Print the total time used by the program
void print_time(struct timeval start, struct timeval end){
    long int ms = ((end.tv_sec * 1000000 + end.tv_usec) -
                   (start.tv_sec * 1000000 + start.tv_usec));
    double s = ms/1e6;
    printf("Time : %f s\n", s);
}

// Returns a random numer from a uniform distribution
double uniform_distribution(int rangeLow, int rangeHigh)
{
    double myRand = rand()/(1.0 + RAND_MAX);
    return myRand;
}

/* Returns the linear indicies corresponding to the entries of the array
   which
    satisfies the given condition*/
int* find_geq(double* array, double condition, int size)
```

```
{
    int* output = (int*)malloc(sizeof(double)*size);
    int pointer = 0;
    for (int i = 0; i < size; i++)
    {
        output[i] = 0;
        if (array[i] >= condition)
        {
            output[pointer] = i;
            pointer ++;
        }
    }
    return output;
}

// Returns a vector containing the cumulative sum of the elements of the
   array k
double* cumsum(double* k, int size)
{
    double* output = malloc(sizeof(double)*size);
    output[0] = k[0];
    for (int i = 1; i < size; i++)
    {
        output[i] = output[i-1]+k[i];
    }
    return output;
}

/* Harvesting based upon a gradual continous increase in the harvesting
   rate as a
   function of the population size */
double H(double y)
{
    double T = 1;
    double h = 1;
    return (h*(y-T)/(h+y-T));
}

/* Perform the Lotka-Volterra Prey-Predator simulation using the residence
    time
   algorithm of KMC */
```

```
void KMC(int Nt, double alpha, double beta, double gamma, double k, double
    m,
          int prey_init, int predator_init, int runs)
{
    //Pre-allocating arrays for time steps and the populations of
   predators and preys
    double* T = malloc(sizeof(double)*(Nt+1));
    double* prey = malloc(sizeof(double)*(Nt+1));
    double* predator = malloc(sizeof(double)*(Nt+1));

    double* T_final = malloc(sizeof(double)*(Nt+1));
    double* prey_final = malloc(sizeof(double)*(Nt+1));
    double* predator_final = malloc(sizeof(double)*(Nt+1));

    // Initializing the arrays
    for (int i=0; i<Nt; i++){
        prey[i] = 0;
        predator[i] = 0;
        T[i] = 0;
    }

    double rates[4];
    double* s;
    double u;
    int* index;

    int x,y;

    for(int j=0; j<runs; j++)
    {
        x = prey_init;
        y = predator_init;
        T[0]=0;
        prey[0] += x;
        predator[0] += y;

        double current_time = 0;

        for (int i = 0; i < Nt; i++) {
            // Calculating the transition to perform
            rates[0] = alpha*x;
```

```
            rates[1] = beta*x*y+gamma*x*x;
            rates[2] = k*beta*x*y;
            rates[3] = m*y+H(y); // H is the harvesting of predators
            s = cumsum(rates, 4);
            u = uniform_distribution(0,1);
            index = find_geq(s, s[3]*u, 4);

            switch (index[0]) {
                case 0:
                    x = x+1;
                    break;
                case 1:
                    if (x > 1)
                    {
                        x = x-1;
                    }
                    break;
                case 2:
                    y=y+1;
                default:
                    if (y>1)
                        y = y-1;
            }
            // Updating the populations
            prey[i+1] += x;
            predator[i+1] += y;

            // Calculating the time step
            double dt = -log(1-u)/s[2];
            current_time += dt;
            T[i+1] = current_time;
        }
    }
    for (int i=0; i<Nt; i++){
        prey[i] = prey[i]/(size*runs);
        predator[i] = predator[i]/(size*runs);
        T[i] = T[i]/(size*runs);
    }

    // Sum up all the runs performed by each process
    MPI_Reduce(prey, prey_final, Nt, MPI_DOUBLE, MPI_SUM, 0, cart_comm);
```

```
    MPI_Reduce(predator, predator_final, Nt, MPI_DOUBLE, MPI_SUM, 0,
  cart_comm);
    MPI_Reduce(T, T_final, Nt, MPI_DOUBLE, MPI_SUM, 0, cart_comm);

    // Write the results to a file which will be used to create plots
  using MATLAB
    if (rank==0)
    {
        FILE *file = fopen("KMC_mpi.txt", "w");
        if (file == NULL)
        {
            printf("Error opening file!\n");
            exit(1);
        }
        for (unsigned int j = 0; j < Nt; j ++) {
            fprintf(file, "%f %f %f \n", T_final[j] , prey_final[j],
  predator_final[j]);
        }
        fclose(file);
    }
}



int main(int argc, char **argv) {
    /*
     argv[1] = Number of time steps
     argv[2] = Alpha, the rate constant of prey reproduction
     argv[3] = Beta, the rate constant of prey death and predator
  reproduction
     argv[4] = Gamma, the rate constant of predator death
     argv[5] = k, predator population growth rate due to predation
     argv[6] = m, rate of predator decline in absence of prey
     argv[7] = Initial prey population
     argv[8] = Initial predator population
     argv[9] = Number of independent simulations for each process
     */

    if (argc !=10){
```

```
        printf("Need 9 inputs: Nt, alpha, beta, gamma, k, m, initial prey
    and initial"
               "predator populations, number of simulations for each
    process \n");
        exit(0);
    }

    init_mpi(argc, argv);

    struct timeval start, end;

    if (rank == 0)
        gettimeofday(&start, NULL);

    KMC(atoi(argv[1]), atof(argv[2]), atof(argv[3]), atof(argv[4]), atof(
    argv[5]),
        atof(argv[6]), atoi(argv[7]), atoi(argv[8]), atoi(argv[9]));

    if (rank == 0){
        gettimeofday(&end, NULL);
        print_time(start, end);
    }

    MPI_Finalize();

    exit(0);
}
```

code/KMC_mpi.c

```
/*This is the source code used to perform the Lotka-Volterra simulation
   using
the time algorithm of KMC.
 */

#include <cuda.h>

#include <stdlib.h>
#include <stdio.h>

#include <math.h>
#include <stdbool.h>
```

```
#include <time.h>
#include <sys/time.h>

#include <curand.h>
#include <curand_kernel.h>

#define DIM 8 // Total number of runs is DIM*runs
#define Nt 5000
#define Dt 40000 // DIM*Nt
#define runs 4




// Print the total time used by the program
void print_time(struct timeval start, struct timeval end){
    long int ms = ((end.tv_sec * 1000000 + end.tv_usec) -
                   (start.tv_sec * 1000000 + start.tv_usec));
    double s = ms/1e6;
    printf("Time : %f s\n", s);
}

// Print the average total time used by the program
void print_time_average(struct timeval start, struct timeval end, int
   number_of_runs){
    long int ms = ((end.tv_sec * 1000000 + end.tv_usec) -
                   (start.tv_sec * 1000000 + start.tv_usec));
    double s = ms/(1e6*number_of_runs);
    printf("Time : %f s\n", s);
}


/* Returns the indices corresponding to the entries of the array which
   satisfies the
given condition */
__device__ int* find_geq(double* array, double condition)
{
    int* output = (int*)malloc(sizeof(int)*3);
    int pointer = 0;
    for (int i = 0; i < 3; i++)
```

```
    {
        output[i] = 0;
        if (array[i] >= condition)
        {
            output[pointer] = i;
            pointer ++;
        }
    }
    return output;
}

// Returns a vector containing the cumulative sum of the elements of the
   array k
__device__ double* cumsum(double* k)
{
    double* output = (double*)malloc(sizeof(double)*3);
    output[0] = k[0];
    for (int i = 1; i < 3; i++)
    {
        output[i] = output[i-1]+k[i];
    }
    return output;
}


//Declare the CUDA kernel
__global__ void KMC(double *T, double *prey, double *predator, double
   alpha,
double beta, double gamma, int prey_init, int predator_init, curandState *
   states,
unsigned long seed)
{
    __shared__ double T_shared[runs*Nt];
    __shared__ double prey_shared[runs*Nt];
    __shared__ double predator_shared[runs*Nt];

    double k[3];
    double* s;
    float u;
    int* index;
```

```
int id = threadIdx.x;

curand_init(seed, id, 0, &states[id]);
curandState localState = states[id];

int x,y;

x = prey_init;
y = predator_init;
T_shared[id*Nt]=0;
prey_shared[id*Nt] = x;
predator_shared[id*Nt] = y;

double current_time = 0;

for (int i = 0; i < Nt; i++) {
    k[0] = alpha*x;
    k[1] = beta*x*y;
    k[2] = gamma*y;
    s = cumsum(k);
    u = curand_uniform(&localState);
    index = find_geq(s, s[2]*u);
    switch (index[0]) {
        case 0:
            x = x+1;
            break;
        case 1:
            if (x > 1)
            {
                x = x-1;
                y = y+1;
            }
            break;
        default:
            if (y>1)
                y = y-1;
    }
    prey_shared[id*(Nt)+i+1] = x;
    predator_shared[id*(Nt)+i+1] = y;
    double dt = -log(1-u)/s[2];
    current_time += dt;
```

```
            T_shared[id*(Nt)+i+1] = current_time;
        }

        __syncthreads();

        if (id ==0){
            int current_run = blockIdx.y+blockIdx.x*gridDim.x;
            for (int i=0; i<Nt; i++){
                prey[current_run *(Nt)+i] = 0;
                predator[current_run *(Nt)+i] = 0;
                T[current_run *(Nt)+i] = 0;

                for (int j=0; j<runs; j++){
                    prey[current_run *(Nt)+i] += prey_shared[(Nt)*j+i];
                    predator[current_run *(Nt)+i] += predator_shared[(Nt)*j+i
    ];
                    T[current_run *(Nt)+i] += T_shared[(Nt)*j+i];
                }
                prey[current_run *(Nt)+i] = prey[current_run*(Nt)+i]/runs;
                predator[current_run *(Nt)+i] = predator[current_run*(Nt)+i]/
    runs;
                T[current_run *(Nt)+i] = T[current_run*(Nt)+i]/runs;
            }
        }

}

int main(int argc, char **argv) {

    double alpha = 2; // The rate constant of prey reproduction
    double beta = 0.05; // The rate constant of prey death and predator
   reproduction
    double gamma= 1.5; // The rate constant of predator death
    int prey_init= 50; // Initial prey population
    int predator_init= 50; // Initial predator population

    struct timeval start, end;


    // Allocate arrays on host
    double* T_host = (double*)malloc(sizeof(double)*Dt);
```

```
    double* prey_host = (double*)malloc(sizeof(double)*Dt);
    double* predator_host = (double*)malloc(sizeof(double)*Dt);


    // Allocate array on device
    double *T, *prey, *predator;

    cudaError_t err = cudaMalloc((void **) &T, Dt*sizeof(double));
    printf("CUDA malloc T: %s\n",cudaGetErrorString(err));

    err = cudaMalloc((void **) &prey, Dt*sizeof(double));
    printf("CUDA malloc prey: %s\n",cudaGetErrorString(err));

    err = cudaMalloc((void **) &predator, Dt*sizeof(double));
    printf("CUDA malloc predator: %s\n",cudaGetErrorString(err));

    /*for (int i=0; i<Dt; i++){
        prey_host[i] = 0;
        predator_host[i] = 0;
        T_host[i] = 0;
    }*/

    // Copy arrays to CUDA device
    err = cudaMemcpy(T, T_host, Dt*sizeof(double), cudaMemcpyHostToDevice)
;
    printf("Copy T to device: %s\n",cudaGetErrorString(err));

    err = cudaMemcpy(predator, predator_host, Dt*sizeof(double),
cudaMemcpyHostToDevice);
    printf("Copy predator to device: %s\n",cudaGetErrorString(err));

    err = cudaMemcpy(prey, prey_host, Dt*sizeof(double),
cudaMemcpyHostToDevice);
    printf("Copy prey to device: %s\n",cudaGetErrorString(err));

    curandState *devStates;
    cudaMalloc(&devStates, DIM*sizeof(curandState));
    gettimeofday(&start, NULL);

    dim3 threadsPerBlock(atoi(argv[6]),1);
    dim3 grid(DIM,1);
```

```
 KMC<<<1,10>>>(T, prey, predator, alpha, beta, gamma, prey_init,
predator_init,
                devStates, time(NULL));

  err = cudaThreadSynchronize();
  printf("Run kernel: %s\n", cudaGetErrorString(err));

  // Retrieve result from device and store it in host array
  err = cudaMemcpy(T_host, T, Dt*sizeof(double), cudaMemcpyDeviceToHost)
;
  printf("Copy T off of device: %s\n",cudaGetErrorString(err));

  err = cudaMemcpy(predator_host, predator, Dt*sizeof(double),
cudaMemcpyDeviceToHost);
  printf("Copy predator off of device: %s\n",cudaGetErrorString(err));

  err = cudaMemcpy(prey_host, prey, Dt*sizeof(double),
cudaMemcpyDeviceToHost);
  printf("Copy prey off of device: %s\n",cudaGetErrorString(err));

  gettimeofday(&end, NULL);
  print_time(start,end);

  for (int i=0; i<Nt; i++){
      for (int j=1; j<DIM; j++){
          T_host[i] += T_host[i +j*Nt];
          prey_host[i] += prey_host[i +j*Nt];
          predator_host[i] += predator_host[i +j*Nt];
      }
      T_host[i] = T_host[i]/(DIM);
      prey_host[i] = prey_host[i]/(DIM);
      predator_host[i] = predator_host[i]/(DIM);
  }

  FILE *file = fopen("KMC_mpi.txt", "w");
  if (file == NULL)
  {
      printf("Error opening file!\n");
      exit(1);
  }
```

```
    for (unsigned int j = 0; j < Nt; j ++) {
        fprintf(file, "%f %f %f \n", T_host[j] , prey_host[j],
  predator_host[j]);
    }
    fclose(file);

    // Cleanup
    cudaFree(T);
    cudaFree(prey);
    cudaFree(predator);

    exit(0);
}
```

code/KMC_cuda_shared.cu

```
/*This is the source code used to perform the Lotka-Volterra simulation
  using the time algorithm of KMC .
 */
#include <cuda.h>


#include <stdlib.h>
#include <stdio.h>

#include <math.h>
#include <stdbool.h>

#include <time.h>
#include <sys/time.h>

#include <curand.h>
#include <curand_kernel.h>

#define DIM 16384 // Number of threads
#define Nt 5000 // Time steps
#define runs 2 // Number of Blocks


// Print the total time used by the program
void print_time(struct timeval start, struct timeval end){
    long int ms = ((end.tv_sec * 1000000 + end.tv_usec) -
```

```
                    (start.tv_sec * 1000000 + start.tv_usec));
    double s = ms/1e6;
    printf("Time : %f s\n", s);
}

// Print the average total time used by the program
void print_time_average(struct timeval start, struct timeval end, int
   number_of_runs){
    long int ms = ((end.tv_sec * 1000000 + end.tv_usec) -
                    (start.tv_sec * 1000000 + start.tv_usec));
    double s = ms/(1e6*number_of_runs);
    printf("Time : %f s\n", s);
}


/* Returns the indices corresponding to the entries of the array which
   satisfies the given condition */
__device__ int* find_geq(double* array, double condition)
{
    int* output = (int*)malloc(sizeof(int)*3);
    int pointer = 0;
    for (int i = 0; i < 3; i++)
    {
        output[i] = 0;
        if (array[i] >= condition)
        {
            output[pointer] = i;
            pointer ++;
        }
    }
    return output;
}

// Returns a vector containing the cumulative sum of the elements of the
   array k
__device__ double* cumsum(double* k)
{
    double* output = (double*)malloc(sizeof(double)*3);
    output[0] = k[0];
    for (int i = 1; i < 3; i++)
    {
```

```
        output[i] = output[i-1]+k[i];
    }
    return output;
}


//Declare the CUDA kernel
__global__ void KMC(double *T, double *prey, double *predator, double
   alpha, double beta, double gamma, int prey_init, int predator_init,
   curandState *states, unsigned long seed)
{

    double k[3];
    double s[3];
    float u;
    int index[3];

    int id = blockIdx.x *blockDim.x + threadIdx.x;

    curand_init(seed, id, 0, &states[id]);
    curandState localState = states[id];

    int x,y, pointer;

    x = prey_init;
    y = predator_init;
    T[id*Nt]=0;
    prey[id*Nt] = x;
    predator[id*Nt] = y;

    double current_time = 0;

    for (int i = 0; i < Nt; i++) {
        k[0] = alpha*x;
        k[1] = beta*x*y;
        k[2] = gamma*y;
        s[0] = k[0];
        for (int j = 1; j < 3; j++)
            s[j] = s[j-1]+k[j];

        u = curand_uniform(&localState);
```

```
            pointer = 0;
            for (int j = 0; j < 3; j++){
                index[j] = 0;
                if (s[j] >= s[2]*u){
                    index[pointer] = j;
                    pointer ++;
                }
            }
            //index = find_geq(s, s[2]*u);
            switch (index[0]) {
                case 0:
                    x = x+1;
                    break;
                case 1:
                    if (x > 1)
                    {
                        x = x-1;
                        y = y+1;
                    }
                    break;
                default:
                    if (y>1)
                        y = y-1;
            }
            prey[id*(Nt)+i+1] = x;
            predator[id*(Nt)+i+1] = y;
            double dt = -log(1-u)/s[2];
            current_time += dt;
            T[id*(Nt)+i+1] = current_time;
        }
        __syncthreads();
}


int main(int argc, char **argv) {

    double alpha = 2; // The rate constant of prey reproduction
    double beta = 0.05; // The rate constant of prey death and predator
  reproduction
    double gamma= 1.5; // The rate constant of predator death
```

```
  int prey_init= 50; // Initial prey population
  int predator_init= 50; // Initial predator population

  struct timeval start, end;


  // Allocate arrays on host
  double* T_host = (double*)malloc(sizeof(double)*Nt*DIM*runs);
  double* prey_host = (double*)malloc(sizeof(double)*Nt*DIM*runs);
  double* predator_host = (double*)malloc(sizeof(double)*Nt*DIM*runs);


  // Allocate array on device
  double *T, *prey, *predator;

  cudaError_t err = cudaMalloc((void **) &T, Nt*DIM*runs*sizeof(double))
;
  printf("CUDA malloc T: %s\n",cudaGetErrorString(err));

  err = cudaMalloc((void **) &prey, Nt*DIM*runs*sizeof(double));
  printf("CUDA malloc prey: %s\n",cudaGetErrorString(err));

  err = cudaMalloc((void **) &predator, Nt*DIM*runs*sizeof(double));
  printf("CUDA malloc predator: %s\n",cudaGetErrorString(err));

  /*for (int i=0; i<Nt*DIM*runs; i++){
      prey_host[i] = 0;
      predator_host[i] = 0;
      T_host[i] = 0;
  }*/

  // Copy arrays to CUDA device
  err = cudaMemcpy(T, T_host, Nt*DIM*runs*sizeof(double),
cudaMemcpyHostToDevice);
  printf("Copy T to device: %s\n",cudaGetErrorString(err));

  err = cudaMemcpy(predator, predator_host, Nt*DIM*runs*sizeof(double),
cudaMemcpyHostToDevice);
  printf("Copy predator to device: %s\n",cudaGetErrorString(err));
```

```
  err = cudaMemcpy(prey, prey_host, Nt*DIM*runs*sizeof(double),
cudaMemcpyHostToDevice);
  printf("Copy prey to device: %s\n",cudaGetErrorString(err));

  curandState *devStates;
  cudaMalloc(&devStates, DIM*sizeof(curandState));
  gettimeofday(&start, NULL);

  //dim3 threadsPerBlock(atoi(argv[6]),1);
  //dim3 grid(DIM,1);

  int threadsPerBlock = DIM;
  int block = runs;

 KMC<<<block,threadsPerBlock>>>(T, prey, predator, alpha, beta, gamma,
prey_init, predator_init, devStates, time(NULL));

  err = cudaThreadSynchronize();
  printf("Run kernel: %s\n", cudaGetErrorString(err));

  // Retrieve result from device and store it in host array
  err = cudaMemcpy(T_host, T, Nt*DIM*runs*sizeof(double),
cudaMemcpyDeviceToHost);
  printf("Copy T off of device: %s\n",cudaGetErrorString(err));

  err = cudaMemcpy(predator_host, predator, Nt*DIM*runs*sizeof(double),
cudaMemcpyDeviceToHost);
  printf("Copy predator off of device: %s\n",cudaGetErrorString(err));

  err = cudaMemcpy(prey_host, prey, Nt*DIM*runs*sizeof(double),
cudaMemcpyDeviceToHost);
  printf("Copy prey off of device: %s\n",cudaGetErrorString(err));

  for (int i=0; i<Nt; i++){
      for (int j=1; j<DIM*runs; j++){
          T_host[i] += T_host[i +j*Nt];
          prey_host[i] += prey_host[i +j*Nt];
          predator_host[i] += predator_host[i +j*Nt];
      }
      T_host[i] = T_host[i]/(DIM*runs);
      prey_host[i] = prey_host[i]/(DIM*runs);
```

```
        predator_host[i] = predator_host[i]/(DIM*runs);
    }
    gettimeofday(&end, NULL);
    print_time(start,end);

    FILE *file = fopen("KMC_mpi.txt", "w");
    if (file == NULL)
    {
        printf("Error opening file!\n");
        exit(1);
    }
    for (unsigned int j = 0; j < Nt; j ++) {
        fprintf(file, "%f %f %f \n", T_host[j] , prey_host[j],
  predator_host[j]);
    }
    fclose(file);

    // Cleanup
    cudaFree(T);
    cudaFree(prey);
    cudaFree(predator);

    exit(0);
}
```

code/KMC_cuda_global.cu

Bibliography

[1] Bertrand Delgutte. Random variables and probability density functions, 2000. URL http://web.mit.edu/~gari/teaching/6.555/lectures/ch_pdf_sw.pdf.

[2] Karl Sigman. Discrete-time markov chains, 2009. URL http://www.columbia.edu/~ks20/stochastic-I/stochastic-I-MCI.pdf.

[3] Petere Mörters and Yuval Peres. Brownian motion. *Draft Version*, http://www.stat.berkeley.edu/ peres/bmbook.pdf, May 2008.

[4] Steven P. Lalley. Stochastic differential equations, May 2012. URL http://galton.uchicago.edu/~lalley/Courses/385/SDE.pdf.

[5] Utz von Wagner Wolfram Martens. On the solution of high dimensional fokker planck equations using orthogonal polynomial expansion, 2010. URL http://onlinelibrary.wiley.com/store/10.1002/pamm.201010121/asset/257_ftp.pdf;jsessionid=B74C331A9356E98D89EFB8CAB0B54B04.f04t04?v=1&t=ianlfr20&s=311b3bf08345b2d09536220314f3e60fdc7b59ee.

[6] J. Kenneth Shultis William L. Dunn. *Exploring Monte Carlo Mathods*. Academpic Press, 2012.

[7] Arthur F. Voter. Introduction to the kinetic monte carlo method, 2005. URL http://www.fml.t.u-tokyo.ac.jp/~izumi/CMS/MC/Introduction_kMC.pdf.

[8] Amdahl's law vs. gustafson-barsis' law, october 2013. URL http://www.drdobbs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980.

[9] Haizhen Wu. Parallel computing using gpus. Master's thesis, School of Engineering and Computer Science., http://ecs.victoria.ac.nz/foswiki/pub/EResearch/EcsTeslaResource2010/Parallel.Computing.Usin March 2011.

[10] Blaise Barney. Openmp, 2015 May. URL https://computing.llnl.gov/tutorials/openMP/.

[11] Blaise Barney. Message passing interface (mpi), May 2015. URL https://computing.llnl.gov/tutorials/mpi/.

[12] nvidia developer. Cuda c programming guide, march 2015. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3YmH7CfR0.

[13] Cyril Zeller. Cuda c/c++ basics, 2011. URL http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf.

[14] Karl Sigman. Introduction to stochastic integration. http://www.columbia.edu/ ks20/FE-Notes/4700-07-Notes-Ito.pdf, 2007.

[15] Sebastian Jaimungal. Stochastic calculus main results. http://www.utstat.utoronto.ca/sjaimung/protected/mmf1952/notes/StochasticCalculus.pdf, 2006.

[16] Gaute Halvorsen. Numerical solution of stochastic differential equations by use of path integration. Master's thesis, NTNU, October 2011.

[17] F. James. Monte carlo theory an practice, 1980. URL http://gruppo3.ca.infn.it/defalco/fisica/james-mc.pdf.

[18] Marc R. Roussel. Stability analysis for odes, 2005. URL http://people.uleth.ca/~roussel/nld/stability.pdf.

[19] Christopher Cooper. Gpu computing with cuda. Pdf, Boston University, http://www.bu.edu/pasi/files/2011/07/Lecture31.pdf, August 2011.

[20] Agnes Koi Alexandersen. Numerical solution of stochastic differential equations. Master's thesis, NTNU, 2014.