IT 18 048
Examensarbete 30 hp
November 2018

Spark for HPC: a comparison with MPI on compute-intensive applications using Monte Carlo method

Huijie Shen
Tingwei Huang

Institutionen för informationsteknologi
Department of Information Technology

Abstract

Spark for HPC: a comparison with MPI on compute-intensive applications using Monte Carlo method

Huijie Shen and Tingwei Huang

With the emergence of various big data platforms in recent years, Apache Spark - a distributed large-scale computing platform, is perceived as a potential substitute for Message Passing Interface (MPI) in High Performance Computing (HPC). Due to the limitations in fault-tolerance, dynamic resource handling and ease of use, MPI, as a dominant method to achieve parallel computing in HPC, is often associated with higher development time and costs in enterprises such as Scania IT. This thesis project aims to examine Apache Spark as an alternative to MPI on HPC clusters and compare their performance in various aspects. The test results are obtained by running a compute-intensive application on both platforms to solve a Bayesian inference problem of a extended Lotka-Volterra model using particle Markov chain Monte Carlo methods. As is confirmed by the tests, Spark is demonstrated to be superior in fault tolerance, dynamic resource handling and ease of use, whilst having its shortcomings in performance and resource consumption compared with MPI. Overall, Spark proves to be a promising alternative of MPI on HPC clusters. As a result, Scania IT continues to explore Spark on HPC clusters for use in different departments.

Handledare: Erik Lönroth
Ämnesgranskare: Sverker Holmgren
Examinator: Mats Daniels
IT 18 048
Tryckt av: Reprocentralen ITC

Acknowledgement

We would like to express gratitude to the following people for their involvement and contributions in this thesis project:

Erik Lönroth and Jimmy Hedman at Scania IT and Mohammadtaghi Rezaei at KTH, for their continuous guidance as supervisors of the project

Prof. Sverker Holmgren at Uppsala University for his valuable feedback on the report

Prof. Darren Wilkinson at Newcastle University, for providing use case on our application

Fredrik Hurtig, Lars Kihlmark, Daniel Sigwid, Mats Johansson at Scania IT, for their help in solving technical and process issues throughout the project

Justin Pearson and Edith Ngai at Uppsala University, for their help on practical issues in the thesis process

Acronym

HPC	High Performance Computing
LV	Lotka-Volterra
MCMC	Markov chain Monte Carlo method
MH	Metropolis-Hastings algorithm
MPI	Message Passing Interface
ODE	ordinary differential equation
PMMH	particle marginal Metropolis-Hastings algorithm
PDF	probability density function
RDD	Resilient Distributed Dataset

Contents

1. Introduction

1.1 Project Motivation

Message Passing Interface (MPI) is currently the dominant method of doing message-passing and data exchange within High Performance Computing (HPC) applications. While being efficient, MPI has long been known for its limitations in fault-tolerance and dynamic resource balancing. Besides, the semantics of the MPI designed to facilitate the work of both researchers and tool builders has, on the contrary, further complicated their work.

The development of Apache Spark as an emerging method of performing distributed computing within HPC has given rise to great interests in recent years. Not only that Spark is featured by automatic fault tolerance, some research work has also shown comparable performance between Spark and MPI for data-intensive applications [Jha et al., 2014]. Besides, a higher level cluster processing framework like Spark enables users to focus more on the problem itself. As a result, the solution delivered is likely to be simpler and more condensed than MPI, with much less work involved in case of slight adjustments to the program. Examples on how MPI and Spark differ in code complexity can be illustrated in a 1D diffusion equation problem [Dursi,].

At Scania IT, HPC clusters are provided to many departments for development use. Due to challenges in developing robust MPI applications and the high costs in case of crashing, alternative distributed computing platforms on the HPC clusters are in urgent need.

In this thesis project, we intend to explore Spark as an alternative to MPI on the HPC clusters. A well known compute-intensive algorithm - particle Markov Chain Monte Carlo (pMCMC) is selected as the method to solve the Bayesian inference problem of an extended Lotka-Volterra model. Study of data-intensive frameworks such as Spark shows the state of running classical HPC applications on this type of frameworks with its advanced features including fault tolerance, reactive and dynamic resource allocation, and domain specific languages. For the specific compute-intensive applications, Spark is compared against MPI in various aspects such as performance, fault tolerance, dynamic resource handling, resource utilization and ease of use. The test results aim to provide insights in the possibility of introducing Spark or the combination of Spark and other traditional platforms in HPC applications at Scania IT. We believe the thesis work would also benefit those who are interested in big data/hybrid solutions for typical HPC problems.

1.2 Scope

Particle Filter and Markov chain Monte Carlo (MCMC) algorithm provide computational methods for systematic inference and learning in complex dynamical systems, such as nonlinear and non-Gaussian state-space models. This project is built upon a compute-intensive application using combination of these two methods. The application is run on two different platforms: Apache Spark-a data-intensive computing platform, and MPI-a traditional HPC platform.

The use case selected is based on the Bayesian inference of original Lotka-Volterra model using particle marginal Metropolis-Hastings algorithm [Wilkinson, 2011b]. We extended the use case to solve Bayesian inference problem of an extended Lotka-Volterra model, and implemented it in Scala for Spark and C for MPI. The sequential implementation of the use case in Scala can be found on-line[1], by the author of the original application. Our work has adapted the base implementation into a parallel one with Spark and MPI.

The application will be tested on both Spark and MPI on HPC clusters. Apart from the performance and scalability of both platforms, we also look at the profiling information to better understand how Spark/MPI distributes workload. Tests are performed to check if Spark can handle node failure as well as dynamic resources allocation as it claims, in comparison to MPI. Furthermore, different parameters are tuned to gain additional insights about Spark - with respect to performance and resource utilization.

[1]`https://github.com/darrenjw/djwhacks/tree/master/scala/bayeskit`

2. Background

2.1 High Performance Computing

High-performance computing (HPC) is the use of supercomputers and parallel processing techniques for solving complex computational problems [tec, 2016]. It is a sub-discipline of computer science that focuses on developing an utilized architecture around supercomputer with respect to both software and hardware. The main idea is to achieve efficient infrastructure for parallel computing.

2.1.1 History of Supercomputer

A supercomputer is a computer or arrays of computers with high performance in processing computational intensive problems compared to a general personal computer or server. Supercomputers are mainly used to perform complex scientific calculation tasks, modeling simulation and processing 3D graphics etc. The speed of supercomputers is generally measured in "FLOPS" (FLoating point Operations Per Second), as compared to "MIPS" (Million Instructions Per Second) in the case of general-purpose computers [Wu, 1999].

Supercomputers were first introduced in 1960s, with the Atlas jointly designed by University of Manchester, Ferranti and Plessey and CDC 6600 by Seymour Cray with speed up to 3 megaFLOPS. During the next two decades, Cray kept delivering supercomputers: Cray 1 in 1970's and Cray 2 in 1980's reached speed up to 1.9 gigaFLOPS. Both of the Cray series were successful supercomputers at that time. In 1990's, processor, network and parallel architecture in both hardware and software of supercomputer were improved largely. Thousands of processors and high speed network were introduced in supercomputer, as well as MPI. For example, Hitachi SR2201 using 2048 processors achieved the performance of 600 gigaFLOPS. By 2000's, with the introduction of mechanics such as GPU, TPU and optimised chips, more and more customized supercomputers were invented such as the famous Deep Blue. Within those decades, the speed of supercomputer has increased from megaFLOPS in 1960's to dozens of PFLOPS today. As of June 2016, the Sunway TaihuLight, with a Linpack benchmark of 93 PFLOPS, was ranked 1st of the Top500 supercomputers.

2.1.2 Supercomputer architecture

Local parallelism was the main approach used to achieve high performance on the earliest supercomputers. Local paralleling computing such as pipelining and vector processing was suitable for early supercomputers as they only had a few closely packed processors.

As the number of processors increased on a single supercomputer, massive parallelism became the trend. There are two types of massive parallel computing. One is massive centralized parallel system which mainly uses supercomputers with a large number of processors located in close proximity. The other is massive distributed parallel computing. For example, in grid computing, the processing power of a large group of computers in distributed, diverse administrative domains is opportunistically used whenever a computer is available [Prodan and Fahringer, 2007]. Another example is BOINC: a volunteer-based, opportunistic grid system, whereby the grid provides computing power only on a best effort basis [Francisco and Erick, 2010].

2.1.3 Software and system management

The basic software stack on HPC cluster consists of operating systems and drivers, schedulers, file systems, cluster manager and HPC programming tools. In this section, we will focus on operating system and scheduler which are relevant to this project. MPI, as one of the programming standard on HPC and an important part of this project, will be introduced in a separated section below.

Operating system on HPC

In the early days of supercomputer, its architecture and hardware developed so rapidly that operating system had to be customized to adapt to its need. As a result, the fast changing architecture brought along various quality issues and high cost. In the 1980s, Cray spent as much on software development as on hardware. Gradually, in-house operating systems were replaced by generic software [MacKenzie, 1998]. The idea was to use a general operating system (e.g. Unix in old days and Linux in recent years), with rich additional functions to fit for supercomputers with various architectures.

As general operating system stabilizes, the size of the code has also increased, which made it hard for developers to use the system and maintain their code. To avoid this, people started to use a lightweight microkernels system which only included minimal functions to support the implementation of operating system. In general, microkernels operating systems only provide low-level address management, thread management and inter-process communication. Based on the development of microkernels operating system, a mixed type of operating system came into use. Microkernels operating system usually run on compute nodes, instead of the server or I/O nodes used in larger system such as Unix or Linux.

Figure 2.1 shows the operating system used on top 500 supercomputers between 1995 and 2015. Clearly, Linux had become the mainstream operating system for supercomputers over these years.

This project is based on a test cluster at Scania IT which uses Red Hat Linux 6.4 as operating system.

Resource management/Job scheduler on HPC

A job scheduler is an application to control the job execution with regards to dependencies. It is used for submission and execution of orders in a distributed network of computers. Most operating systems and some programs provide in-house job scheduling capabilities. It is usually necessary in an enterprise to control and manage jobs submitted from different business units with the help of job schedulers. In Scania IT, LSF(IBM's Platform Load Sharing Facility) is the job scheduler for the HPC clusters. LSF has the capability to schedule diverse jobs for HPC clusters. LSF is featured by monitoring state of resources, managing the job queue according to user-defined policies, and assigning the resources to the jobs properly. In this project, the applications implemented in Spark and MPI are both submitted to the LSF queuing system and scheduled by

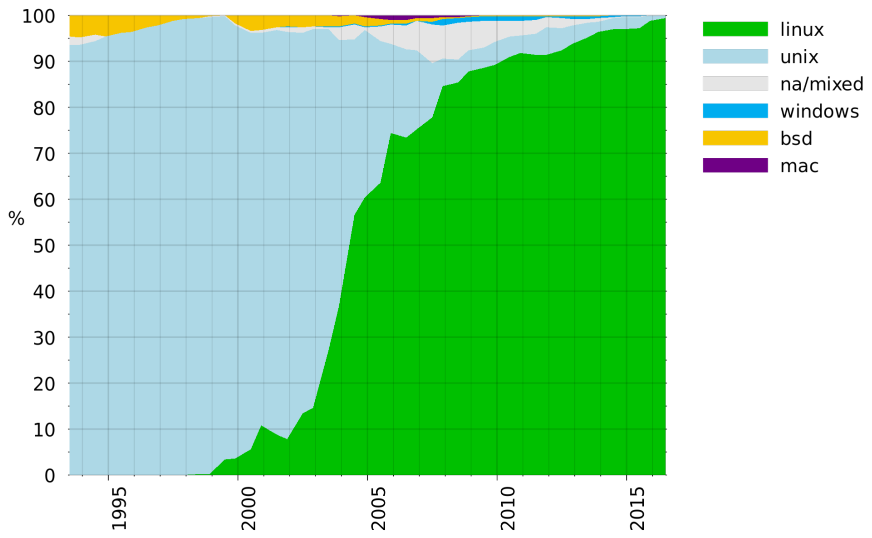


Figure 2.1. The operating system used on top 500 supercomputers

it. Besides LSF, there are some other popular job schedulers including Simple Linux Utility for Resource Mangement(SLURM) and OpenLava etc.

2.2 Message Passing Interface

MPI is a HPC programming standard coming with a core library to support scalable and portable parallel programming model on HPC platform. It is a programming interface for message-passing application, together with protocol and semantic specifications for how its features must behave in any implementation (such as a message buffering and message delivery progress requirement). MPI includes point-to-point message passing and collective (global) operations, all scoped to a user-specified group of processes [Gropp et al., 1996].

2.2.1 Parallel Computing Model

With the success of simulations on supercomputers, demands for supercomputing resources increased sharply in 1980s. On one hand, single computing was not sufficient to accomplish complex scientific simulation. On the other hand, the increased performance and capacity of both Ethernet network and wide-area network made it possible to exchange data efficiently. All the factors combined gave rise to parallel computing.

Examples of parallel computational models at that time were data parallelism, shared memory, message passing, remote memory operation, threads and some combined models [Gropp et al., 1999]. Using Flynn's taxonomy, these models can be categorized below based on multiple data streams: SIMD - when data parallelism makes the same instructions carried out simultaneously on multiple data items; MIMD - when task parallelism makes multiple instructions executed on multiple data streams; SPMD - when multiple processors execute same program on multiple data streams; MPMD - when multiple processors execute multiple programs on multiple data streams [Flynn, 1972]. SPMD is equivalent to MIMD since each MIMD program can be made SPMD (similarly for

SIMD, but not in practical sense). The message-passing model addressed by MPI is for MIMD/SPMD parallelism [Gropp, 1998].

2.2.2 Evolution of MPI

A traditional process consists of a program counter and its associated memory space. Multiple threads from the same process use respective memory space. To achieve parallel computing, MPI carries out the communication among them with synchronization and data transfer from one process memory to another by sending and receiving messages [Gropp, 1998].

The initial workshop in MPI was established in 1992. A working group comprised of 175 individuals from parallel computer vendors, software developers, academia and application scientists met in November, 1992. They presented the draft MPI in 1993. And later in 1994 the first version of MPI (MPI-1.0) was introduced. In 1998, MPI-2 came alive with more details in MPI specification. The MPI standard used in the thesis is MPI-3.0, which was released in 2012.

The primary goal of the MPI specification is to demonstrate that users do not need to compromise efficiency, portability, or functionality in order to write portable programs [Gropp et al., 1999]. It is the first universal, vendor independent standardized specification accomplished by a message passing library. The earliest MPI was designed for distributed memory architectures. As the HPC architecture changed, it started to support hybrid architecture combining distributed memory and shared memory systems. Today, MPI could run on any platform, no matter being distributed memory, shared memory or hybrid system.

2.2.3 Implementation of MPI

There are a lot of MPI implementations; the popular ones in recent years are MVAPICH MPI, OpenMPI and LAM MPI. Many efforts have been made on hardware-specific MPI implementations, such as Cray MPI, HP MPI, and IBM MPI, etc. Most of the implementations of MPI have native support for C, C++ and Fortran. There are some libraries that support developing MPI applications using other languages, such as Bryan Carpenter's mpiJava for Java, and pyMPI for Python, etc. In this project, we develop the application with C in OpenMPI implementation.

The MPI program written in C/Fortran is compiled by ordinary C/Fortran compiler and linked with the MPI library. In our case, the C compiler is gcc. The details of developing MPI application will be described in Section 4.4.2.

2.3 Apache Spark

Apache Spark is a large-scale distributed data processing engine. It provides high-level APIs for different programming languages (Scala, Java, Python and R for now) that allow users to efficiently execute data processing, streaming, SQL or machine learning workloads. It achieves fast and robust large-scale computing through data parallelism and built-in fault tolerance.

Spark project was first initiated around 2009 by Matei Zaharia during his Ph.D program at University of California Berkeley's AMPLab. It was then donated to and maintained by Apache Software Foundation.

Spark consists of Spark core and other components such as APIs for steaming or machine learning jobs. It can be run on Apache Mesos, Hadoop YARN or Spark standalone cluster mode and access distributed data sources such HDFS (Hadoop Distributed File System), Cassandra, S4, HBase as well as non-distributed file systems. Users can develop Spark applications using its API and different libraries. Figure 2.2 shows Spark architecture and its basic modules.

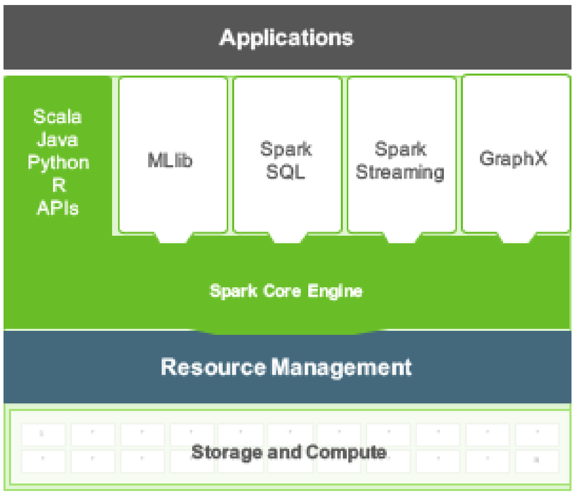


Figure 2.2. Aparch Spark Architecture

2.3.1 Spark Core

Spark core is the kernel of the whole Spark project. It is based on the abstraction of a Resilient Distributed Dataset(RDD), which is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost [Zaharia et al., 2010]. Using RDD allows users to explicitly store intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators [Zaharia et al., 2012]. This significantly helps to improve the performance of running iterative machine learning tasks and interactive data mining jobs. These jobs need to keep reloading data from disk, which is a challenge for other cluster parallel computing models such as MapReduce and Dryad. RDDs are instead loaded (and cached if required) into memory and can be reused by these interactive jobs efficiently.

RDDs can only be created through deterministic operations on either data in stable storage or other RDDs [Zaharia et al., 2012]. These operations are usually called transformations including *map*, *filter* and *join*. Another type of operations supported by RDDs are actions, such as *reduce*, *collect*, *foreach* etc. Actions usually return final results after a few RDD transformations. These general operations make it possible to perform many different operations on RDDs in parallel, and users are granted high-level control of their applications though these operations on RDDs.

A RDD has enough information (*lineage*) about how it was derived from other RDDs [Zaharia et al., 2012]. Spark is able to recover lost partitions of a RDD by recomputing the logged *lineage* information. This is basically how fault tolerance is achieved in Spark.

2.3.2 Spark Module

In addition to Spark core, Spark provides generality platform with modules of Spark SQL, Spark Streaming, MLlib (Machine Learning library) and GraphX (graph computing library) for users to develop functional applications efficiently. These modules are formatted as libraries and users could combine a few of them in one application.

2.3.3 Spark Scheduler

Spark scheduler is a cluster manager to schedule Spark jobs. Spark currently could work against three type of cluster managers: Spark Standalone, Apache Mesos and Hadoop YARN(Yet Another Resource Negotiator).

Apache Mesos is a cluster manager handling cluster resources such as CPU, memory and storage resources for all applications. It is a common resource manager with fine-grained, simple, scalable scheduling mechanism. It could schedule Spark job and other tasks as well. Although it is quite different with the LSF queuing system, they have similar feature with regards to resource management and task scheduling. Due to potential conflicts of the two platforms on the same cluster, we therefore consider the other two options.

Hadoop YARN is the prerequisite for Enterprise Hadoop, providing resource management and a central platform to deliver consistent operations, security, and data governance tools across Hadoop clusters [hor, 2011]. YARN is one of the two core-compenents of Hadoop 2.0+. Spark could also run on Hadoop using YARN as the scheduler. Hadoop framework is normally used to process large datasets, which is not necessary for the sake of this project. Hence, we decide to go with Spark and its native standalone cluster manager without using distributed file system.

Spark standalone is self-packaged by Spark. It uses master/slave framework. Spark master is responsible for distributing tasks and data to workers; the executors on each of the workers will execute received tasks. The cluster manager provides the capability to launch the Spark master and worker, run Spark applications, schedule the cluster resources, fault tolerance, monitor and log for jobs (with a Web UI). Users can tune the cluster through the configuration files, scripts, or parameters when submit a Spark application. Currently standalone cluster manager uses a simple FIFO queue to schedule across multiple applications.

2.3.4 How Spark works on a cluster

So far, we have introduced some key concepts of Spark such as RDD and cluster manager. In this section, we will put these components together and briefly explain how Spark works on a cluster. Some new concepts that have not been mentioned previously will be introduced as well.

Once Spark is deployed on a cluster and Spark standalone cluster manager (master) is started on one node while connected with some worker nodes, the user can submit a Spark application to the master from a computer that can connect with the cluster manager and workers.

Every Spark application must create a *SparkContext* so that Spark knows how to connect to the cluster. The program in which SparkContext is created is called *driver program*. Driver program negotiates with cluster manager for executors on worker nodes. One worker node can have one or more executors, depending on the configuration.

After the executors are acquired, the driver program is sent to the executors. The driver program contains the user code to perform transformations and actions on RDDs as well as some other functions. A complete series of transformations on RDDs that end with action or data saving is one single *job*. The transformations on RDDs in this job will be translated into a *DAG* (Direct

Acyclic Graph). There can be multiple *stage*s within a job, containing transformations that do not need data shuffling or re-partitioning. Each stage is divided into multiple parallel *task*s that operates on a single partition of the RDD. These tasks are sent to the executors by the driver program through cluster manager. Once executed, the results are returned to the driver program.

Figure 2.3 shows how Spark works in a cluster, with more details illustrated in Figure 2.4. Some sophisticated concepts in these figures are not covered in this section, as they are not used in the whole report. Readers may find relevant materials on the Internet or contact us if they are interested to learn more.

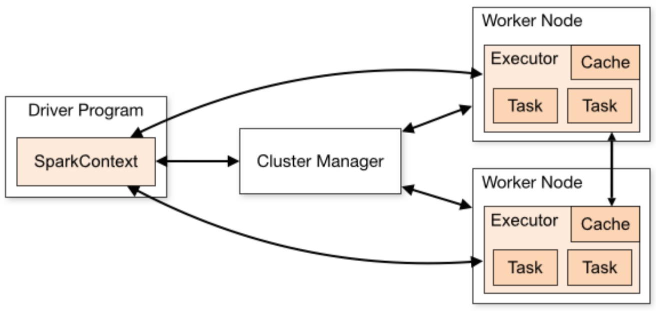
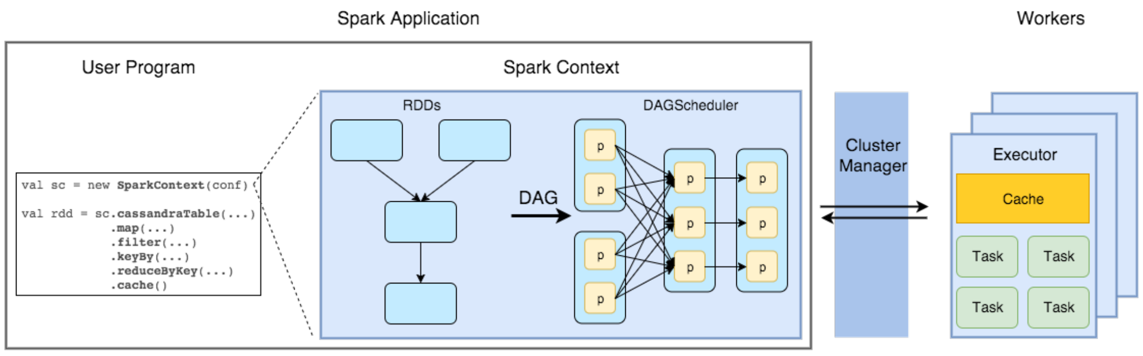


Figure 2.3. A basic view about how Spark works in a cluster. From http://spark.apache.org/docs/1.6.0/cluster-overview.html; accessed 2016-09-10



```
val sc = new SparkContext(conf)

val rdd = sc.cassandraTable(...)
             .map(...)
             .filter(...)
             .keyBy(...)
             .reduceByKey(...)
             .cache()
```

Figure 2.4. A more detailed view about how Spark works in a cluster. From http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/; accessed 2016-09-10

3. The application

In this project, we are interested in comparing Spark and MPI for a specific application. Since Spark has been widely known as a powerful platform for data-intensive applications, we instead selected a compute-intensive application that solves the Bayesian inference problem of an extended Lotka-Volterra model, using particle marginal Metropolis-Hastings algorithm. The details will be explained in this chapter.

3.1 Lotka-Volterra model

Lotka-Volterra (LV) model [Lotka, 1925, Volterra, 1926] is a model that describes the dynamics of two interacting species in a simplified biological system - the prey and the predator. The model can be used to predict the population of both species in a simple biological system, with broader applications in other areas such as economics.

3.1.1 Deterministic Lotka-Volterra model

Lotka-Volterra model have two interacting species - one prey and one predator. The dynamics of the population can be described by two ordinary differential equations (ODEs):

$$\frac{dx}{dt} = \alpha x - \beta xy \tag{3.1}$$

$$\frac{dy}{dt} = \delta xy - \gamma y \tag{3.2}$$

for α, β, δ, $\gamma > 0$. In Equation 3.1 and 3.2, x and y represent the population of prey and predator respectively, and t represents the time. α is the reproduction rate of the prey, and γ is the death rate of the predator. β and δ are the interaction rate between prey and predator.

Equation 3.1 describes the changing rate of the population for the prey. The population increases through reproduction and decreases through getting killed by the predators. Whereas equation 3.2 describes the balance of the predator population, which is impacted by consumption of preys and natural death. For simplicity, no other interaction is considered in this model.

The model described by Equation 3.1 and 3.2 is the deterministic form of the LV model. Figure 3.1 and 3.2 illustrates a deterministic LV model, where α = 1, β = 0.1, δ = 1.5 and γ = 0.08. Such deterministic form is periodic with perfectly repeating oscillations and carries on indefinitely. In contrast, stochastic form oscillates, but in a random, unpredictable way [Wilkinson, 2011a]. Therefore, a deterministic LV model can barely reflect the realistic dynamics of the biological system, in which randomness in many small parts of the system lead to a relatively high unpredictability in the evolution of the whole system. In this sense, a stochastic perspective of the model is closer to the reality, and we adopted a stochastic form of LV model in the application.

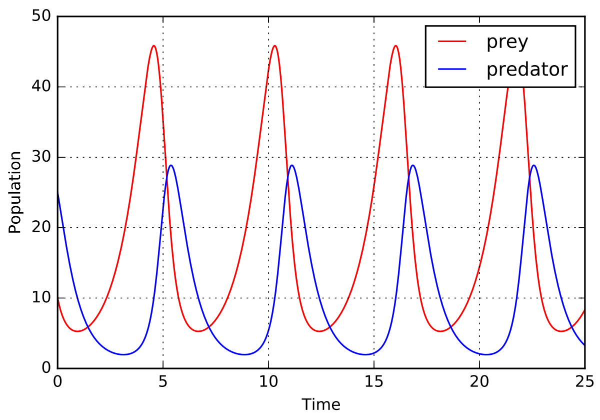


Figure 3.1. The dynamics of prey and predator in a Lotka-Volterra model

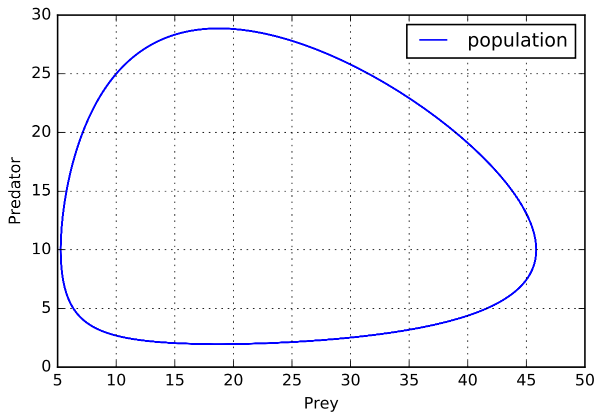


Figure 3.2. The population of prey and predator in a deterministic LV model is changing periodically and sustains dynamic equilibrium

3.1.2 Stochastic Lotka-Volterra model

The stochastic LV model can be represented by three stochastic chemical kinetics formulas [Golightly and Gillespie, 2

$$X_1 \xrightarrow{c_1} 2X_1 \tag{3.3}$$

$$X_1 + X_2 \xrightarrow{c_2} 2X_2 \tag{3.4}$$

$$X_2 \xrightarrow{c_3} \emptyset \tag{3.5}$$

where X_1 and X_2 are the population of prey and predator respectively. c_1, c_2 and c_3 are the rate parameters. Reaction 3.3 is the reproduction of the prey; reaction 3.4 is the interaction between prey and predator; reaction 3.5 is the death of predator. Each reaction has its own hazard:

$$h_1(x,\ c_1) = c_1 x_1 \tag{3.6}$$

$$h_2(x,\ c_2) = c_2 x_1 x_2 \tag{3.7}$$

$$h_3(x,\ c_3) = c_3 x_2 \tag{3.8}$$

The stoichiometry matrix of the model is

$$S = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \end{pmatrix} \tag{3.9}$$

By applying Gillespie's algorithm, we can derive the population of the prey and predator after any given time period, if we consider the dynamics as a stochastic process. Gillespie's algorithm can simulate the forward process of a stochastic model. It was popularized by Dan Gillespie in late 1970s [Gillespie, 1977], and was called "stochastic simulation algorithm" in that paper back then. Gillespie's algorithm has been well summarized as algorithm 1 by Golightly and Gillespie [Golightly and Gillespie, 2013]. Since stochastic chemical kinetics is based on biochemistry, the population originally represented the number of molecules of each species. That is why x_i represents the initial molecule number in the first step of algorithm 1.

Algorithm 1 Gillespie's algorithm

```
1: Set $t = 0$. Initialise the rate constants $c_1, \dots, c_v$ and the initial molecule numbers $x_1, \dots, x_u$.
2: Calculate $h_i(x, c_i),\ i = 1, ..., v$ based on the current state, $x$.
3: Calculate the *combined hazard* $h_0(x, c) = \sum_{i=1}^{v} h_i(x, c_i)$
4: Simulate the time to the next event, $t' \sim Exp(h_0(x, c))$ and put $t := t + t'$.
5: Simulate the reaction index, $j$, as a discrete random quantity with probabilities
   $h_i(x, c_i)/h_0(x, c),\ i = 1, \dots, v$.
6: Update $x$ according to reaction $j$. That is, put $x := x + S^j$.
7: Output $x$ and $t$. If $t < T_{max}$, return to step 2.
```

The dynamics of prey and predator in a 50-unit-long time range in a stochastic Lotka-Volterra model are shown in Figure 3.3 and 3.4. In this model, c_1 = 1, c_2 = 0.005, and the initial population of prey and predator is 80 and 50 units, respectively. Unlike deterministic LV model, stochastic LV model does not follow a periodic pattern, instead, the model will end up in the extinction of one or both species in finite time. The extinction cases of stochastic LV model can be found in the book by Ullah and Wolkenhauer [Ullah and Wolkenhauer, 2011, p. 136-137] and many other sources.

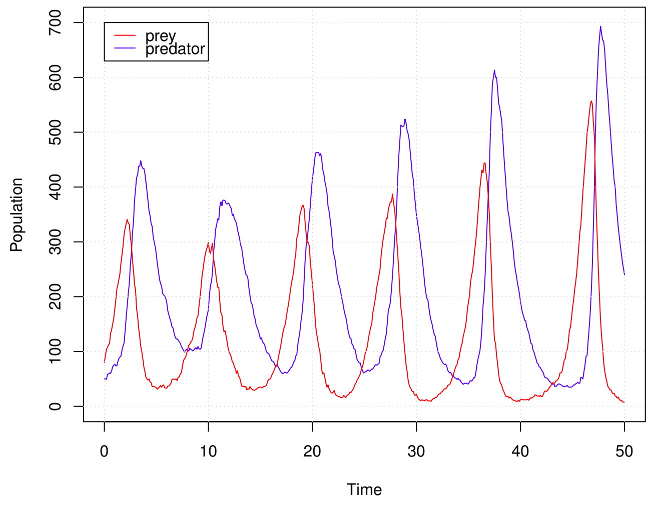


Figure 3.3. The dynamics of prey and predator in a stochastic Lotka-Volterra model

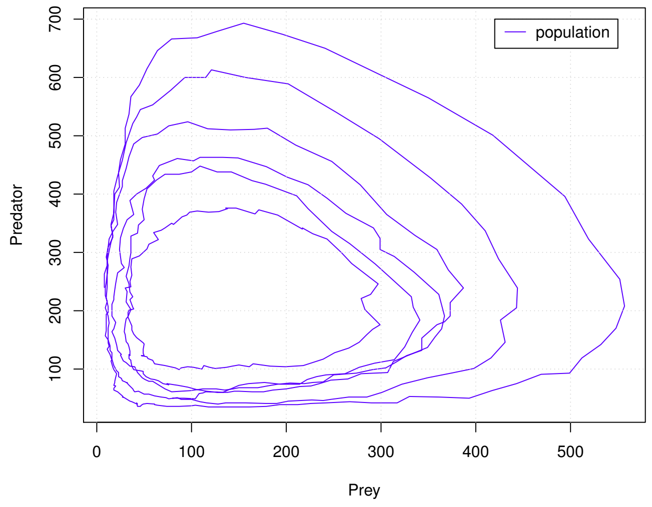


Figure 3.4. The population of prey and predator in a stochastic LV model does not change periodically and one species will extinct in finite time

3.1.3 An extended Lotka-Volterra model

Our project intends to compare Spark and MPI for compute-intensive applications. By design, the application shall take sufficiently long computing time for each parallel part. In our first attempt, we used the application that solves the Bayesian inference problem of a stochastic Lotka-Volterra model using particle marginal Metropolis-Hastings algorithm (see Section 3.2.1). However, as illustrated in Figure 3.5, the overhead of the computing job is much more time-consuming than the actual computing itself. Since the computing load is not significant enough, the benefit of parallel execution on HPC platform is minimal.

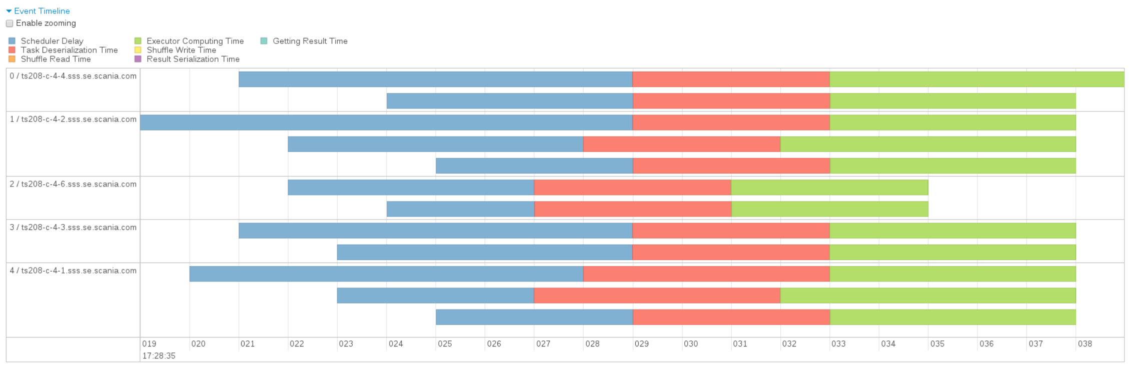


Figure 3.5. The time decomposition for 12 parallel computing jobs in Spark for a stochastic LV model. The blue, red and green bars show the time spent on scheduling, task deserialisation and computing, respectively.

To solve the problem, we extended the one prey-one predator LV model (LV-11 model) to one prey-two predators LV model (LV-12 model), based on the generalised LV-1n model [Ristic and Skvortsov, 2015]. The computation in LV-12 model has largely increased compared with LV-11 model, based on the test results.

LV-12 model adds one more predator into the LV-11 model. The two predators live on the same prey, but do not prey on each other. The deterministic LV-12 model can be described by three ODEs:

$$\frac{dx}{dt} = \alpha x - \beta_1 x y_1 \tag{3.10}$$

$$\frac{dy_1}{dt} = \delta_1 x y_1 - \gamma_1 y_1 \tag{3.11}$$

$$\frac{dy_2}{dt} = \delta_2 x y_2 - \gamma_2 y_2 \tag{3.12}$$

The definitions of variables in Equation 3.10, 3.11 and 3.12 are similar to those in Equation 3.1 and 3.2. β_i and δ_i represent the interaction rate between the prey and the i^{th} predator. γ_i is the death rate of the i^{th} predator for i = 1, 2.

For stochastic LV-12 model, the same assumption of prey and predators applies. The stochastic chemical kinetics formulas are

$$X_1 \xrightarrow{c_1} 2X_1 \tag{3.13}$$

$$X_1 + X_2 \xrightarrow{c_2} 2X_2 \tag{3.14}$$

$$X_1 + X_3 \xrightarrow{c_3} 2X_3 \tag{3.15}$$

$$X_2 \xrightarrow{c_4} \emptyset \tag{3.16}$$

$$X_3 \xrightarrow{c_5} \emptyset \tag{3.17}$$

where X_1, X_2 and X_3 are the population of prey and the two predators. c_i (i = 1,. . . ,5) are the reaction rates. Compared with the stochastic LV-11 model, two reactions are added in the stochastic LV-12 model: the interaction between the added predator (reaction 3.15) and the prey and the death of the added predator (reaction 3.17).

The hazards of each reaction are

$$h_1(x,\ c_1) = c_1x_1 \tag{3.18}$$

$$h_2(x,\ c_2) = c_2x_1x_2 \tag{3.19}$$

$$h_3(x,\ c_3) = c_3x_1x_3 \tag{3.20}$$

$$h_4(x,\ c_4) = c_4x_2 \tag{3.21}$$

$$h_5(x,\ c_5) = c_5x_3 \tag{3.22}$$

The stoichiometry matrix of the model is

$$S = \begin{pmatrix} 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 1 & -1 & -1 \end{pmatrix} \tag{3.23}$$

By applying Gillspie's algorithm (algorithm 1), the random process of the population change can be simulated for any particular time range, given the initial population of the prey and predators. With one more predator in the play, the randomness and complexity have increased tremendously. The population of all the three species at any specific moment differ a lot in different runs. Figure 3.6 shows one simulation run of the LV-12 model using Gillspie's algorithm.

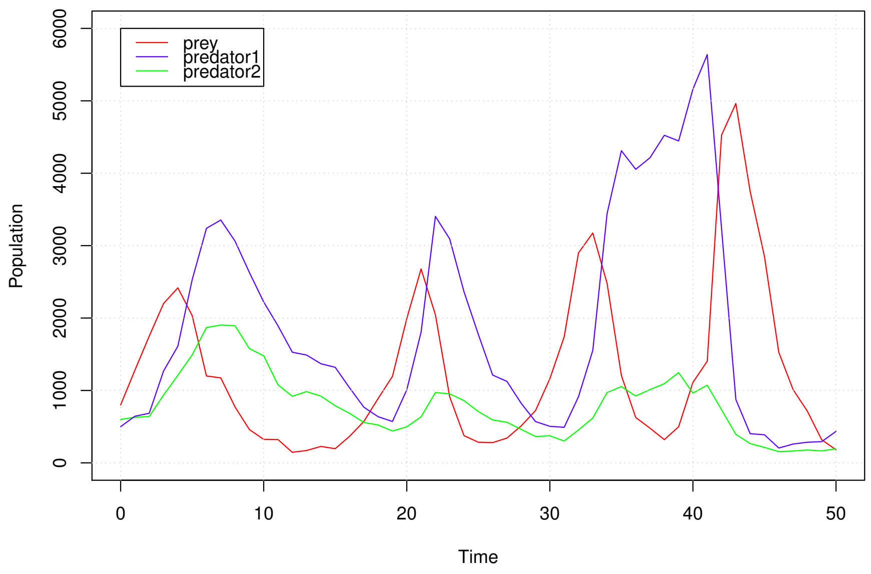


Figure 3.6. The dynamics of prey and two predators in a stochastic LV-12 model. The initial population of prey, first and second predators are 800, 500, and 600. Reaction rate c_i = 10.0, 0.005, 0.0025, 6.0, 3.0, for i = 1,. . . ,5. The population are derived in the time range between 0 and 50, with time interval of 1.

As we can see, LV-12 model shares a lot of basic concepts and equations with LV-11 model. By introducing one more predator into the model, there are more reactions and parameters in the stochastic chemical kinetics representation. Consequently, the model becomes more complex and the computation required for each forward step of the simulation using Gillespie's algorithm is much heavier than that of LV-11. For the same setting of the application on Spark, the total running time of the application using LV-12 model is at least 15 times higher. The large volume of computation for the Bayesian inference problem of the LV-12 model makes it suitable as the application for comparing Spark and MPI for the purpose of high performance computing.

3.2 Monte Carlo methods and Bayesian inference

3.2.1 Metropolis-Hastings algorithm

Metropolis-Hastings algorithm (MH) is one of the simplest and most common Markov chain Monte Carlo methods (MCMC). MH can generate a sequence of random samples from a probability distribution that is not fully known. The probability distribution of the generated samples, or the equilibrium distribution of the Markov chain, approximates the true probability.

Metropolis-Hastings algorithm is described in algorithm 2. The transition kernel $q(\theta'|\theta)$ can be an arbitrary one. The choice of the transition kernel does not affect the true probability distribution of the outcome [Wilkinson, 2011b, p. 265-266]. $\pi(\theta)$ is the probability density function (pdf). Although $\pi(\theta)$ is often the target density of using Metropolis-Hastings algorithm and therefore unknown beforehand, a density proportional to the target density will suffice in MH since only the ratio $\frac{\pi(\theta')}{\pi(\theta)}$ is used in calculating the acceptance ratio.

Algorithm 2 Metropolis-Hastings algorithm

```
1: Suppose the value of the current state is $\theta$.
2: Propose a new value $\theta'$ according to transition kernel $q(\theta'|\theta)$.
3: Set the value of the next state as $\theta'$ with acceptance ratio $\alpha(\theta,\theta') = min\{1, \frac{\pi(\theta')q(\theta|\theta')}{\pi(\theta)q(\theta'|\theta)}\}$. If
   not accepted, set the value as $\theta$.
4: Return to step 1.
```

As a special case of MH algorithm, Metropolis algorithm uses a symmetric proposal $q(\theta'|\theta)$, with $q(\theta'|\theta) = q(\theta|\theta')$. Therefore, the acceptance ratio $\alpha(\theta,\theta')$ simplifies to $min\{1, \frac{\pi(\theta')}{\pi(\theta)}\}$.

3.2.2 Sequential Monte Carlo method

Sequential Monte Carlo method, also known as particle filter, is a method to solve Hidden Markov Model and nonlinear, non-Gaussian filtering problems. In this application, the bootstrap particle filter is used to make Monte Carlo estimate of the marginal likelihood $\pi(y|\theta)$ (see next section for details). In principle, the mechanism of the bootstrap particle filter can be described below.

Each state in the sample space is updated to the next state in the Markov chain using a transition kernel (known beforehand). The updated states are re-sampled based on their weight, which is determined by how close each state is to the observed state. The complete bootstrap particle filter algorithm is well summarised by Doucet et al. [Doucet et al., 2001, Ch. 1] as algorithm 3.

Algorithm 3 bootstrap particle filter

```
1: Initialisation, $t = 0$.
    • For $i = 1,\dots,N$, sample $x_0^{(i)} \sim \pi(x_0)$ and set $t = 1$.
2: Importance sampling step
    • For $i = 1,\dots,N$, sample $\tilde{x}_t^{(i)} \sim \pi(x_t|x_{t-1}^{(i)})$ based on the Markov process model and set $\tilde{x}_{0:t}^{(i)} = (x_{0:t-1}^{(i)}, \tilde{x}_t^{(i)})$.
    • For $i = 1,\dots,N$, evaluate the importance weights $\tilde{w}_t^{(i)} = \pi(y_t|\tilde{x}_t^{(i)})$.
    • Normalise the importance weights.
3: Selection step
    • Resample with replacement N particles $(x_{0:t}^{(i)}; i = 1,\dots,N)$ from the set $(\tilde{x}_{0:t}^{(i)}; i = 1,\dots,N)$ according to the importance weights.
    • Set $t \leftarrow t+1$ and if $t \leq T$, go to step 2 (Importance sampling step).
```

By applying a bootstrap particle filter, we can get the unbiased estimate of $\pi(y_{1:T})$:

$$\hat{\pi}(y_{1:T}) = \hat{\pi}(y_1)\prod_{t=2}^{T}\hat{\pi}(y_t|y_{1:t-1}), \tag{3.24}$$

where

$$\hat{\pi}(y_t|y_{1:t-1}) = \frac{1}{N}\sum_{k=1}^{N}\tilde{w}_t^k. \tag{3.25}$$

The unbiasedness as $M \to \infty$ has been proven in various literatures, see [Del Moral, 2004, Pitt et al., 2010].

3.2.3 Bayesian inference

Bayesian inference is a fundamental method based on Bayes Theorem in Bayesian statistics for statistical inference. Bayesian inference updates the posterior probability based on the prior probability and some observed data.

Formally, Bayesian inference can be represented as

$$P(H_i|X=x) = \frac{P(X=x|H_i)P(H_i)}{\sum_{j=1}^{n} P(X=x|H_j)P(H_j)}, \quad i = 1,\dots,n, \tag{3.26}$$

where H_i is the hypothesis, X is the outcome, and x is an observation of the outcome [Wilkinson, 2011b, p. 249-250]. The LHS of the equation is the posterior probability. $P(H_i)$ is the prior probability of hypothesis H_i. $P(X=x|H_i)$ is the probability of X = x given H_i; it is called likelihood function. $\sum_{j=1}^{n} P(X=x|H_j)P(H_j)$ is the marginal likelihood, which is the probability of $X = x$ averaged across the all possible hypotheses. The marginal likelihood is the same for every hypothesis H_i.

For continuous hypothesis space (and discrete outcome X), Bayesian inference can be represented as

$$\pi(\theta|X=x) = \frac{\pi(\theta)P(X=x|\theta)}{\int_{\Theta} P(X=x|\theta')\pi(\theta')d\theta'}, \tag{3.27}$$

where the hypothesis is denoted as θ in the space Θ, and its prior and posterior probability become density functions, denoted as $\pi(\theta)$ and $\pi(\theta|X=x)$ [Wilkinson, 2011b, p. 250].

Consider the joint probability distribution $\pi(\theta,x) = \pi(\theta)\pi(x|\theta)$, where θ is the parameter and x is the data, a common situation is that x cannot be obtained directly (therefore called latent variable), while y is the data observed that implies x. The joint distribution of θ, x and y is

$$\pi(\theta,x,y) = \pi(\theta)\pi(x|\theta)\pi(y|x,\theta). \tag{3.28}$$

Given observed data y, we can get marginal posterior

$$\pi(\theta|y) = \frac{\pi(\theta)\pi(y|\theta)}{\int_{\Theta} \pi(y|\theta')\pi(\theta')d\theta'}. \tag{3.29}$$

Obviously $\pi(\theta|y) \propto \pi(\theta)\pi(y|\theta)$ since $\int_{\Theta} \pi(y|\theta')\pi(\theta')d\theta'$ is a constant value. To get the marginal posterior $\pi(\theta|y)$, Metropolis-Hastings algorithm (see Section 3.2.1) can be used. The acceptance ratio $\alpha(\theta,\theta')$ in this case should be $min\{1, \frac{\pi(\theta'|y)q(\theta|\theta')}{\pi(\theta|y)q(\theta'|\theta)}\}$, and since $\pi(\theta|y) \propto \pi(\theta)\pi(y|\theta)$, the acceptance ratio becomes

$$min\{1, \frac{\pi(\theta')\pi(y|\theta')q(\theta|\theta')}{\pi(\theta)\pi(y|\theta)q(\theta'|\theta)}\}, \tag{3.30}$$

where $\pi(y|\theta) = \int_X \pi(y|x,\theta)\pi(x|\theta)dx$. The difficulty in this method is that $\pi(y|\theta)$ is hardly possible to derive [Wilkinson, 2011b, p. 271-272]. Instead, we can use a Monte Carlo estimate $\hat{\pi}(y|\theta)$ of $\pi(y|\theta)$. $\hat{\pi}(y|\theta)$ is called an unbiased estimate of $\pi(y|\theta)$ if $E(\hat{\pi}(y|\theta)) = \pi(y|\theta)$. By replacing $\pi(y|\theta)$ with its unbiased estimate $\hat{\pi}(y|\theta)$, the acceptance ratio becomes

$$\alpha(\theta,\theta') = min\{1, \frac{\pi(\theta')\hat{\pi}(y|\theta')q(\theta|\theta')}{\pi(\theta)\hat{\pi}(y|\theta)q(\theta'|\theta)}\}, \tag{3.31}$$

and the target achieved is the exact approximation of the correct marginal posterior $\pi(\theta|y)$ [Wilkinson, 2011b, p. 273-274].

To infer the marginal posterior $\pi(\theta|y_{1:T})$ based on a series of observed data in discrete time $y_{1:T}$, we can use the method above plus a bootstrap particle filter introduced in Section 3.2.2, which will give an unbiased estimate of $\pi(y_{1:T}|\theta)$. At first glance, it seems that the bootstrap particle filter does not give $\hat{\pi}(y_{1:T}|\theta)$ but $\hat{\pi}(y_{1:T})$. But they are equivalent if we look at the bootstrap particle filter carefully. The prerequisite of the bootstrap particle filter is that we already have the parameter θ, and the calculation of $\hat{\pi}(y_{1:T})$ is conditional on the value of θ. Therefore, the bootstrap particle filter gives $\hat{\pi}(y_{1:T}|\theta)$ as a result and the combination of the bootstrap particle filter and Metropolis-Hastings algorithm works perfectly in solving the Bayesian inference problem of marginal posterior $\pi(\theta|y_{1:T})$ based on a series of observed data in discrete time $y_{1:T}$.

3.3 Solving the inference problem of LV-12 model using PMMH algorithm

The use case to compare Spark and MPI is the Bayesian inference problem of the LV-12 model (5 reaction rates as θ introduced in Section 3.1.3), given the observations $y_{1:T}$ of the population of the prey and the two predators in a time period from 1 to T. Since the observations are subject to measurement error, this can be seen as a Bayesian inference problem with latent variable. As discussed in Section 3.2.3, this problem can be solved using particle marginal Metropolis-Hastings (PMMH) algorithm - a Metropolis-Hastings algorithm targeting the marginal posterior with bootstrap particle filter embedded. Algorithm 4 is the application of PMMH to get the marginal posterior $\pi(\theta|y_{1:T})$ of LV-12 model parameters θ based on the observations $y_{1:T}$. The solution can be found in Wilkinson's work [Wilkinson, 2011b, ch. 10]. Most of the algorithm is explained in the subsections above. However, there are some decisions made for the specific use case and they will be justified below.

Algorithm 4 PMMH algorithm to infer LV-12 parameters

```
 1: function PMMH-LV12($observations$)
 2:     $log\_likelihood \leftarrow -\infty$
 3:     $\theta \leftarrow [10.0, 0.005, 0.0025, 6.0, 3.0]$                ▷ Initial guess
 4:     $tune \leftarrow 0.014$                           ▷ Sigma in the transition kernel
 5:     $sample\_size \leftarrow 1000$
 6:     for $i = 0$ to $1000$ do
 7:         $\theta' \leftarrow \theta$ map $\{x \rightarrow x * exp(Gaussian(0, tune).draw())\}$
                                                              ▷ Transition kernel
 8:         $prop\_log\_likeli \leftarrow particle\_filter(\theta', observations, sample\_size)$
 9:         $ran \leftarrow log(Random(0,1).draw())$
10:         if $ran < prop\_log\_likeli - log\_likelihood$ then
11:             $\theta \leftarrow \theta'$
12:             $log\_likelihood \leftarrow prop\_log\_likeli$
13:         end if
14:     end for
15: end function
```

First, marginal log-likelihood (line 2, 8, 12, 22, 35, 38 in Algorithm 4) is used instead of marginal likelihood. This is to improve performance and prevent overflow/underflow in likelihood multiplication [Bellot, 2010].

Second, the initial guess of the parameter θ (line 3 in Algorithm 4) is the parameter used to generate the observations. That means the initial guess is the one we expect to obtain from the algorithm. Practically, it is undoubtedly hard to make the right guess beforehand. We use the expected results as the initial guess because it is easier to observe how the Markov chain (of parameter θ) fluctuates around the initial guess. Moreover, it saves the burn-in time that is a common issue in MCMC algorithm.

Third, we made specific choices of transition kernel and tuning parameter (line 4 and 7 in Algorithm 4). As described in Section 3.2.1, the transition kernel can be an arbitrary one without affecting the true probability distribution of the outcome in Metropolis-Hastings algorithm. By choosing a symmetric transition kernel $\theta' = \theta * exp(Gaussian(0, tune).draw())$ that guarantees $q(\theta'|\theta) = q(\theta|\theta')$, the Metropolis-Hastings algorithm becomes the Metropolis algorithm, and the acceptance rate simplifies to $min\{1, \frac{\pi(\theta')\pi(y|\theta')}{\pi(\theta)\pi(y|\theta)}\}$. Here no prior ratio $\frac{\pi(\theta')}{\pi(\theta)}$ is provided, making the final acceptance rate $min\{1, \frac{\pi(y|\theta')}{\pi(y|\theta)}\}$. The results are still within our expectation though the prior ratio is omitted. The tuning parameter in the transition kernel (sigma in the Gaussian distribution)

```
16: function PARTICLE_FILTER($\theta'$, $observations$, $sample\_size$)
17:     $first\_observ \leftarrow observations.first()$
18:     $prey \leftarrow Possion(first\_observ.get(0)).sample(sample\_size)$
                                                    ▷ Initial samples of the prey
19:     $predator1 \leftarrow Possion(first\_observ.get(1)).sample(sample\_size)$
                                                    ▷ Initial samples of predator1
20:     $predator2 \leftarrow Possion(first\_observ.get(2)).sample(sample\_size)$
                                                    ▷ Initial samples of predator2
21:     $samples \leftarrow Matrix(prey, predator1, predator2).transpose()$
                  ▷ Inital samples of the population; a $sample\_size \times 3$ matrix
22:     $log\_likeli \leftarrow 0$
23:     $prev\_observ\_time \leftarrow first\_observ.time$
24:     for $i$ in $observations$ do
25:         $\Delta t \leftarrow i.time - prev\_observ\_time$
26:         if $\Delta t = 0$ then
27:             $next\_state \leftarrow samples$
28:         else
29:             $next\_state \leftarrow samples$ map $\{x \rightarrow Gillespie\_LV12(x, \Delta t, \theta)\}$
30:         end if
31:         $importance\_w \leftarrow next\_state$ map $\{x \rightarrow likeli(x, i.populations)\}$
32:         $mean \leftarrow mean(importance\_w)$
33:         $importance\_w \leftarrow normalize(importance\_w)$
34:         $samples \leftarrow next\_state.discrete\_sample(n, importance\_w)$
                                                    ▷ Re-sampling
35:         $log\_likeli \leftarrow log\_likeli + log(mean)$
36:         $prev\_observ\_time \leftarrow i.time$
37:     end for
38:     return $log\_likeli$
39: end function
```

```
40: function LIKELI($sample$, $observation$)
41:     $result \leftarrow 0.0$
42:     $sigma \leftarrow 100$
43:     for $i = 0$ to $sample.size$ do
44:         $result \leftarrow result + Gaussian(sample.get(i), sigma).pdf(observation.get(i))$
45:     end for
46:     return $result$
47: end function
```

is set to 0.014 to reach a acceptance rate (of state update in Markov chain) close to 0.234 - a value which has been proved theoretically and practically to optimize the efficiency of Metropolis algorithm [Roberts et al., 1997, Roberts et al., 2001, Christensen et al., 2005, Neal et al., 2006].

Fourth, the sample size (line 5 in Algorithm 4) is set to 1000 to achieve better estimate of the marginal likelihood. It is sufficiently big, yet not too so to cause much unnecessary computation that does not help much in generating the Markov chain of the parameter. A few papers [Pitt et al., 2012, Doucet et al., 2015, Sherlock et al., 2015] have given support to a common rule that the variance of the marginal log-liklihood should be around 1 in particle MCMC methods, whereas some others have cast doubts [Wilkinson, 2014]. In this use case, the variance of the marginal log-likelihood is relatively close to 1, usually in the range of 1.0 - 1.5.

3.4 Input and Outcome

Since it is generally hard to find a biological system which follows a perfect LV-12 model, we generated the input data by simulating a stochastic LV-12 model using Gillespie's algorithm, with the same parameters as the model in Figure 3.6. The initial population of prey, predator1 and predator2 are 800, 500, and 600, and reaction rate c_i is set to 10.0, 0.005, 0.0025, 6.0, 3.0, for $i = 1,\dots,5$.

At time interval of 2, we pass the current population of the three species and Δt of 2 to the Gillespie's algorithm for LV-12 model. This will give us their population after Δt of 2. Since there is much randomness in a stochastic model when the system evolve, we run the Gillespie's algorithm 1000 times for each even time and take the average of the population for each species in these 1000 runs. However, the state of the system is only recorded at time $4n$ for $n = 0,\dots,5$, because longer Δt will lead to longer computation time in Gillespie's algorithm which will be more suitable for parallelism. The generated observation input is listed in Table 3.1.

Table 3.1. *Input data of Bayesian inference problem of LV-12 model*

Time	Prey population	Predator1 population	Predator2 population
0	800	500	600
4	2399	767	728
8	1003	3085	1447
12	403	1132	875
16	1545	642	656
20	1231	2892	1390

Each time we run the PMMH algorithm using the input data as *observations*, we get five Markov chains of the five reaction rates c_i, $i = 1,\dots,5$, as well as the marginal log-likelihood of each state in the Markov chain. A good run of the PMMH algorithm, as mentioned in Section 3.3, should reach a acceptance rate (of state update in Markov chain) close to 0.234, and the variance of the marginal log-likelihood of all states should be relatively close to 1. Figure 3.7 shows the Markov chains of the five reaction rates in such a good run (acceptance rate = 0.207 and variance of marginal log-likelihood is about 1.37). The values on each Markov chain fluctuate around the expected value (value used to generate the input data) (see Figure 3.8). The means and standard deviations of the five reactions rates over their Markov chains are listed in Table 3.2; all the means of the five reactions rates are fairly close to the expected value.

To summarise, this section has explained in necessary details how the Bayesian inference problem of marginal posterior in LV-12 model is solved using particle marginal Metropolis-Hastings algorithm. The method has been proved by our tests to solve the problem to a large extent. However,

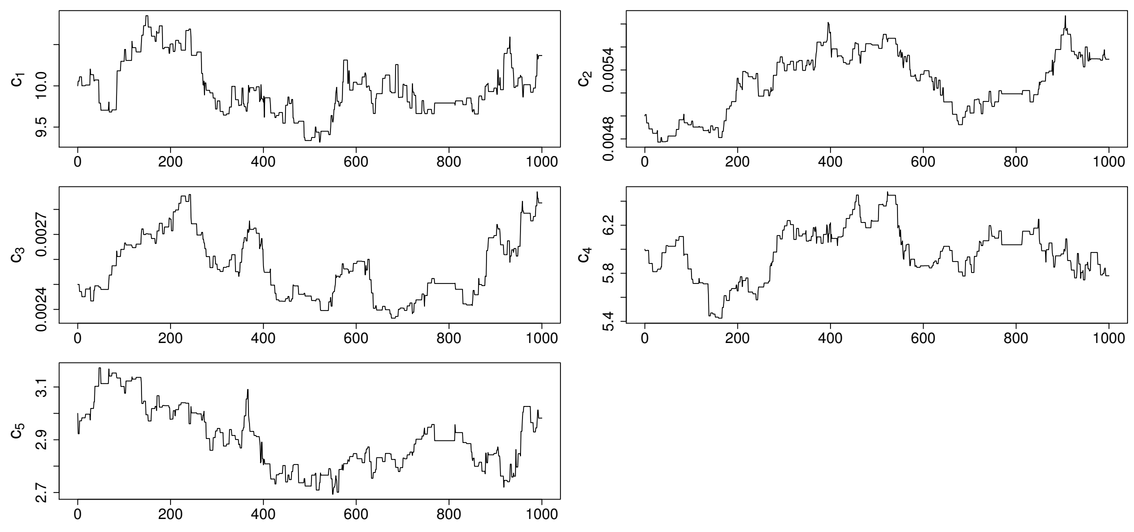


Figure 3.7. The value of the five reaction rates over their Markov chains in a good run

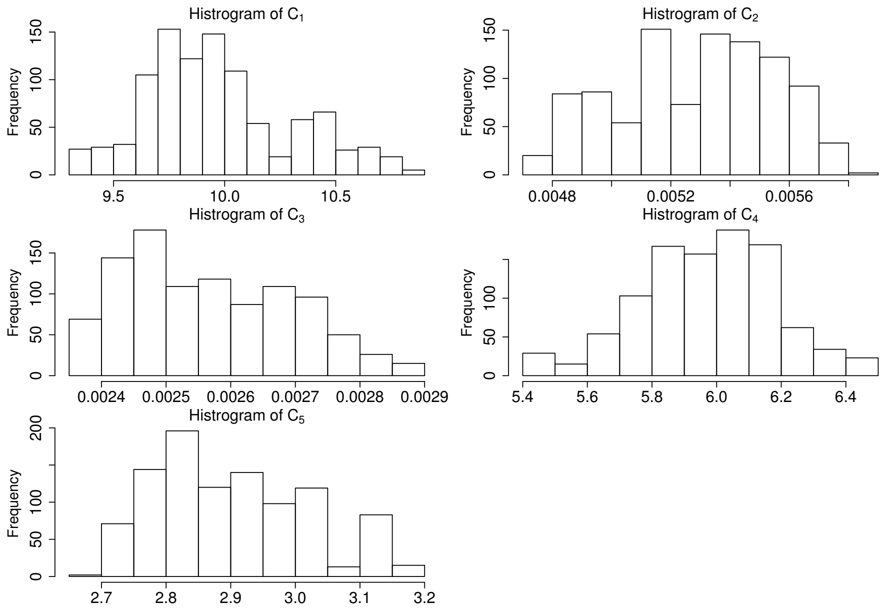


Figure 3.8. The distributions of the five reaction rates over their Markov chains in a good run

Table 3.2. *Means and standard deviations of the five reaction rates*

	c_1	c_2	c_3	c_4	c_5
Mean	9.81426	0.00507	0.00240	6.08547	2.87993
Standard Deviation	0.31405	0.00023	0.00009	0.25211	0.09004
Expected value	10	0.005	0.0025	6	3

this is still a sequential solution despite being a compute-intensive application. To be used widely for research or other practical purposes, the computation time needs to be decreased. In the next section, we will illustrate its parallel implementation in Spark and MPI, as well as how Spark and MPI is set up in the HPC cluster.

4. Implementation

As is proven in our implementation in the sequential program, particle marginal Metropolis-Hastings algorithm can be compute-intensive. To accelerate the computation, computing resources on multiple computers should be utilised. To enable such parallel computing in a cluster, Spark or MPI comes into play.

4.1 System environment

The HPC cluster we deployed Spark and MPI on is a production-ready cluster consisting of 12 nodes (10 for general use). The CPU and memory specification of each node is listed in Table 4.1. Excluding the two special nodes that are reserved for other use, there are in total 216 CPUs and 755.35 GB memory available in the HPC cluster for testing.

Table 4.1. *Specification of each node in the cluster used in the project*

Node	# of CPUs	CPU Frequency (GHz)	Memory (GB)
Node 0	32	2.60	126.15
Node 1	16	2.60	31.46
Node 2	16	2.60	31.46
Node 3	16	2.60	31.46
Node 4	16	2.60	31.46
Node 5	40	2.80	63.01
Node 6	16	3.30	63.01
Node 9	16	3.19	125.78
Node 10	16	3.19	125.78
Node 11	32	3.19	125.78
Special node 1	12	2.00	63.02
Special node 2	16	3.30	63.01

In most test cases, Spark as well as MPI uses Node 1, 2, 3, 4, and 6. We tried to use the same nodes for Spark and MPI in the HPC cluster to eliminate impact from using different hardware. However, in the unavoidable case when the node is utilised by other users, we have to switch to other available nodes during our test.

The HPC cluster uses IBM LSF workload management platform to manage the computing resources of all nodes as well as the job queue. LSF decides when to start a job and which job to start depending on the resource allocation policies, the state of resources and the user request. The version of LSF platform on the test cluster is 9.1.3.

4.2 Set up Spark on HPC cluster

In section 2.3.3, we have introduced three cluster managers supported by Spark and have explained our decision to use Spark Standalone mode on the HPC cluster. To run Spark standalone on a HPC cluster, resources (nodes in this case) need to be allocated by a workload management platform (in our case, LSF) before Spark can be started.

More specifically, we need to specify the needed resources and parameters to start the job in a script, for instance node list, number of cores etc. Then the job is submitted by `BSUB` command to LSF batch scheduling queue. To ensure exclusive execution of Spark on the cluster, we use `BSUB -x` command. The `BSUB` command and its various options can be found in the official reference [1].

Once the requested resources are allocated, the Spark master will be firstly launched. The job process stays idle for 5 seconds to ensure that Spark master is up and running. Then the Spark slaves will be put into operation on the allocated work nodes. The spark-URL of the master (e.g. spark://1.2.3.4:7077) is also be addressed so that the slaves can connect to the master.

The key to launch Spark is the `start-master/slave.sh` script in the Spark package which will call configuration setting batch, environment setting batch, set cluster port and launch the cluster etc. In this project, the Spark package is built from source code using sbt since all the available released packages include Hadoop File System which is the thing that we want to avoid. This package is put on a shared folder accessible by every node in the cluster.

After the cluster up and running, we can below command below to submit a Spark application to the cluster from any computer that can connect to the Spark master

```
spark-submit --class <main-class> --master <master-url>
<application-jar>.
```

The progress of running a Spark application is visiualized through Saprk WebUI http://<driver-node>:4040, so the user can see the real time resources allocation, job stages and job status etc from the webUI. By enabling `eventLog`, the user can also track the whole execution processes through http://<master-url>:8080.

As the Spark installation and launch processes are hidden from the user, it brings no difference to them whether the Spark application is run on HPC cluster or ordinary cluster.

4.3 Set up MPI on HPC cluster

As mentioned in Section 2.2.3, we use C to develop the MPI application with OpenMPI standard. We can compile MPI application with C in similar way as a normal C application. For example, in Linux environment, the general command to compile C application

```
gcc my_mpi_application.c -o my_mpi_application
```

is modified to below for MPI application

```
mpicc my_mpi_application.c -o my_mpi_application.
```

Here we use MPI "wrapper" compiler, which is *mpicc* in the case of C language. What "wrapper" compiler does is adding in all the relevant compiler parameters and then invoking the actual compiler gcc. The command to run MPI application in OpenMPI is *mpirun* or *mpiexec*, for example,

```
mpirun np 4 my_parallel_application,
```

the number 4 here is the number of processor/slot this application requested. To make use of shared-memory model to share data among nodes, the command below can be used

[1]`https://www.ibm.com/support/knowledgecenter/SSETD4_9.1.2/lsf_command_ref/bsub.1.html`, accessed 2016-09-10

```
mpirun -mca btl sm self.
```

To schedule MPI jobs in HPC cluster, we again come to use LSF. LSF provides a flexible, scalable and extensible distributed application integration framework called *blaunch*, with which LSF integrates most popular MPI implementations including Open MPI [Ding, 2013]. *BSUB* is the command to submit a job to LSF platform as we mentioned earlier. Here is a script to be submitted to LSF, `openmpiapp.lsf`.

```
#!/bin/sh

#BSUB -n request_slot_number

#Add openmpi binary and library path to system path

export OpenMPI binary PATH to SYSTEM PATH

export OpenMPI library PATH to SYSTEM library PATH

mpirun my_openmpi
```

This LSF script specifies the resources needed for this MPI application, and exports the path of OpenMPI to system environment variables. This step is the prerequisite of running OpenMPI parallel in a cluster. Then we use BSUB to submit this script:

```
bsub < openmpiapp.lsf.
```

LSF will use *standarderr* and *standout* to print the output of the application. User could add relevant parameters or adjust LSF configurations to adapt to different scenarios.

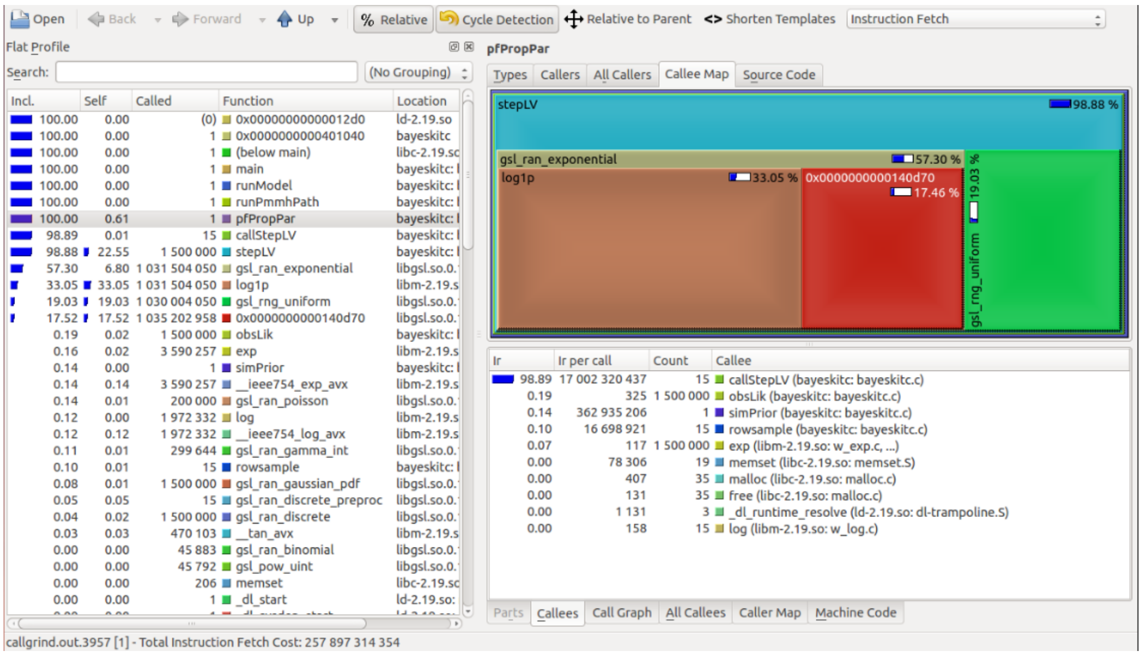
4.4 Parallelise the implementation

This section focuses on how the sequential application is parallelised in Spark and MPI, using two different mechanisms - parallel operations on RDD and message passing. The code for both Spark and MPI implementation is available in our GitHub repository [2]. Readers are welcome to download and run it.

By examining Algorithm 4 in Section 3.3, we found that the iterations in the Metropolis-Hastings algorithm could not be parallelised due to sequential dependence. A common technique in MCMC algorithm to achieve parallelism is to create multiple chains and compute them in parallel. We did not adopt this technique but turned to parallelise the particle filter part embedded in the Metropolis-Hastings algorithm (line 8 in Algorithm 4). This decision was made based on the discovery that the particle filter part is the most compute-intensive in the whole application. As shown in Figure 4.1, stepLV function (i.e. the `Gillespie_LV12` function at line 29) takes 98.88% of the whole computing time. This profiling data was obtained by running Callgrind (a profiling tool) on the sequential implementation written in C. Similar results are shown in the sequential implementation in Scala, too.

The `Gillespie_LV12` function is performed on each of the 1000 samples in the iterations of the particle filter. The `Gillespie_LV12` function on one sample is independent of the states of other samples, and hence can be parallelised. In the next two sections, we will introduce how the parallelism is implemented in Spark and MPI.

[2]`https://github.com/scania/scania-pmcmc`, accessed 2016-09-10



Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x00000000000012d0	ld-2.19.so
100.00	0.00	1	0x0000000000401040	bayeskitc
100.00	0.00	1	(below main)	libc-2.19.sc
100.00	0.00	1	main	bayeskitc: l
100.00	0.00	1	runModel	bayeskitc: l
100.00	0.00	1	runPmmhPath	bayeskitc: l
100.00	0.61	1	pfPropPar	bayeskitc: l
98.89	0.01	15	callStepLV	bayeskitc: l
98.88	22.55	1 500 000	stepLV	bayeskitc: l
57.30	6.80	1 031 504 050	gsl_ran_exponential	libgsl.so.0.
33.05	33.05	1 031 504 050	log1p	libm-2.19.s
19.03	19.03	1 030 004 050	gsl_rng_uniform	libgsl.so.0.
17.52	17.52	1 035 202 958	0x0000000000140d70	libgsl.so.0.
0.19	0.02	1 500 000	obsLik	bayeskitc: l
0.16	0.02	3 590 257	exp	libm-2.19.s
0.14	0.00	1	simPrior	bayeskitc: l
0.14	0.14	3 590 257	__ieee754_exp_avx	libm-2.19.s
0.14	0.01	200 000	gsl_ran_poisson	libgsl.so.0.
0.12	0.00	1 972 332	log	libm-2.19.s
0.12	0.12	1 972 332	__ieee754_log_avx	libm-2.19.s
0.11	0.01	299 644	gsl_ran_gamma_int	libgsl.so.0.
0.10	0.01	15	rowsample	bayeskitc: l
0.08	0.01	1 500 000	gsl_ran_gaussian_pdf	libgsl.so.0.
0.05	0.05	15	gsl_ran_discrete_preproc	libgsl.so.0.
0.04	0.02	1 500 000	gsl_ran_discrete	libgsl.so.0.
0.03	0.03	470 103	__tan_avx	libm-2.19.s
0.00	0.00	45 883	gsl_ran_binomial	libgsl.so.0.
0.00	0.00	45 792	gsl_pow_uint	libgsl.so.0.
0.00	0.00	206	memset	libc-2.19.sc
0.00	0.00	1	_dl_start	ld-2.19.so:

Ir	Ir per call	Count	Callee
98.89	17 002 320 437	15	callStepLV (bayeskitc: bayeskitc.c)
0.19	325	1 500 000	obsLik (bayeskitc: bayeskitc.c)
0.14	362 935 206	1	simPrior (bayeskitc: bayeskitc.c)
0.10	16 698 921	15	rowsample (bayeskitc: bayeskitc.c)
0.07	117	1 500 000	exp (libm-2.19.so: w_exp.c, ...)
0.00	78 306	19	memset (libc-2.19.so: memset.S)
0.00	407	35	malloc (libc-2.19.so: malloc.c)
0.00	131	35	free (libc-2.19.so: malloc.c)
0.00	1 131	3	_dl_runtime_resolve (ld-2.19.so: dl-trampoline.S)
0.00	158	15	log (libm-2.19.so: w_log.c)

callgrind.out.3957 [1] - Total Instruction Fetch Cost: 257 897 314 354

Figure 4.1. Profiling results on the sequential implementation

4.4.1 Parallelise the implementation in Spark

A Spark program starts a `SparkContext` object, which is created with a `SparkConf` as its parameter. `SparkConf` configures various settings of the application, such as application properties, runtime environment, shuffle behavior, etc. A complete list of configurations can be found in the official reference [3]. Some configurations can be tuned by users and the effect will be covered in Section 5.6.

Once a `SparkContext` object *sc* is created, a RDD of the samples can be created by parallelising the `samples` object, using the function below:

$$samples\_rdd \leftarrow sc.parallelize(samples,\ number\_of\_partitions),$$

where *number_of_partitions* specifies how many partitions the dataset should be divided into. Spark runs one task for each partition of the RDD. The official guide suggests 2-4 partitions for each CPU used and we adopted 3 partitions in this project. Once a RDD is created, its partitions can be operated in parallel. In our use case, a `map` operation is performed on *samples_rdd* , passing each sample through `Gillespie_LV12` function. A new RDD with the next states of all the samples is returned afterwards:

$$next\_state\_rdd \leftarrow samples\_rdd.map(x \rightarrow Gillespie\_LV12(x,\ \Delta t,\ \theta))\ .$$

Although some other parallel RDD operations (line 31 to 33 in Algorithm 4) can be performed on *next_state_rdd* and the results returned, our tests showed that it took take longer time in the parallel operations than their sequential versions due to the limited computation load. Therefore, *next_state_rdd* is transformed into an ordinary collection before any operation is performed on it:

$$next\_state \leftarrow next\_state\_rdd.collect().$$

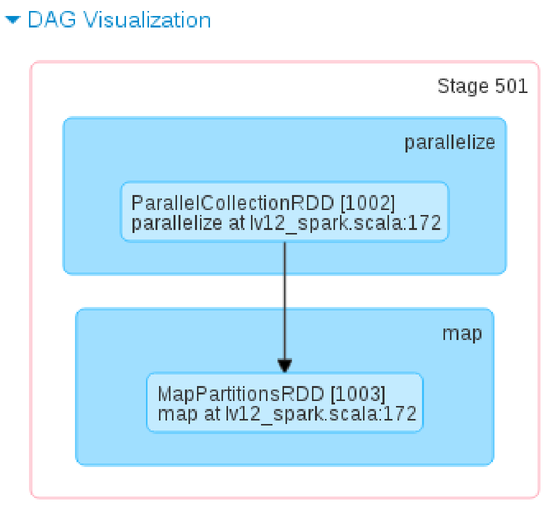
`collect` operation returns all the elements of the RDD as an array, and the array can be transformed into other type of collection by the users.

[3]`http://spark.apache.org/docs/1.6.0/configuration.html`, accessed 2016-09-10

The parallelism achieved using Spark can be summarised as replacing line 29 in Algorithm 4 as the following code piece:

$$\begin{gathered} \textit{samples\_rdd} \leftarrow \textit{sc.parallelize(samples, number\_of\_partitions)} \\ \textit{next\_state\_rdd} \leftarrow \textit{samples\_rdd.map}(x \rightarrow \textit{Gillespie\_LV12}(x,\ \Delta t,\ \theta)) \\ \textit{next\_state} \leftarrow \textit{next\_state\_rdd.collect()}. \end{gathered}$$

All the Spark jobs in this use case are the same and each job only have one stage, therefore the DAG graph shown in Figure 4.2 is the same for the single stage of all jobs.



DAG Visualization
Stage 501
parallelize
ParallelCollectionRDD [1002]
parallelize at lv12_spark.scala:172
map
MapPartitionsRDD [1003]
map at lv12_spark.scala:172

Figure 4.2. DAG visualization of one stage

As can be seen from the DAG graph, the Spark implementation is fairly simple and straightforward, due to the code simplification and the concept of RDD. Our work to parallelise and optimise of the use case shows success, as is confirmed by Figure 4.3) and Figure 4.4. More specifically, we can see that the parallel part (blue bars) takes up most of the computation time while the sequential part (the interval between blue bars) takes up very small proportion.

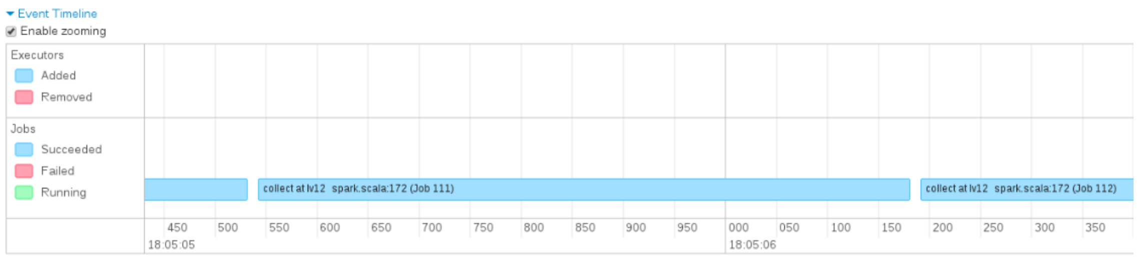


Figure 4.3. A particular job and its intervals between its previous and next jobs

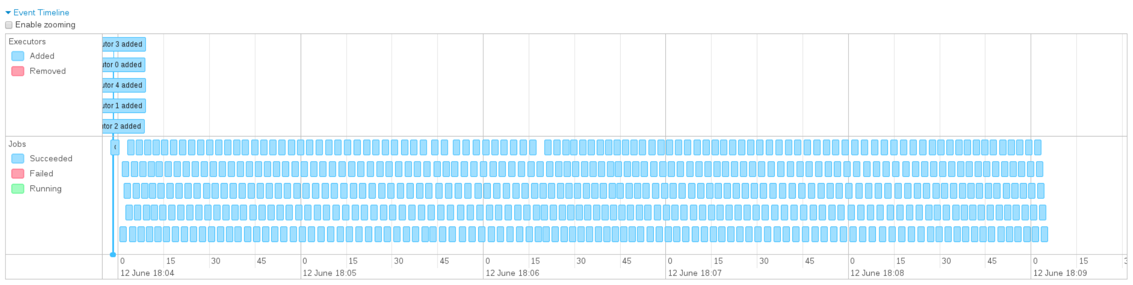


Figure 4.4. The event timeline of a test run of the Spark application

4.4.2 Parallelise the implementation in MPI

As is touched upon in Section 2.2, MPI utilises message passing model to achieve parallel computing. There are two ways to communicate: point-to-point message passing and collective operations. Point-to-point communication is message exchanging between two processes, while broadcast and collective operations will involve all processes in a communicator. When developing the MPI application, the former is the main method we use to pass messages between root and workers, whereas the latter helps to broadcast and synchronize some global variables.

A MPI application should always start with including MPI file, e.g. `mpi.h`. Then a MPI execution environment needs to be initialised by `MPI_Init` in the `main` function. During the initialisation, communicators are constucted. A communicator can be thought as a handle of a group of processes, which is necessary for most MPI operations. The communicators constructed in the `main()` function can be passed as parameter to the other functions. After that, users usually call `MPI_Comm_size(MPI_COMM_WORLD, &procnum)` to get the number of processes and `MPI_Comm_rank (MPI_COMM_WORLD, &rank)` to assign each process a unique integer as their rank/taskID. Function `MPI_Finalize()` is used to terminate the MPI execution environment.

There is no operation to mark the start and end points of parallel processing in a MPI application. Each process will execute the program at the very beginning of `main()` function. To execute sequential code, the user needs to specify one process to run the code using its `rank`. We assign all sequential code to the root process - the one with rank 0. The root always works as a worker as well.

Now we are going to explain how to parallelise the use case in MPI. As explained previously, only `Gillespie_LV12` in Particle Filter method is parallelised. We added the MPI code to parallelise line 29 in Algorithm 4 with the traditional loop in C. The parameters needed for `Gillespie_LV12(x,` Δt`,` θ`)` function are the current population of prey and predator for each sample, the time interval Δt and the model parameters θ, respectively. The last two parameters are the same for all samples, so they are broadcast to each process by operation `MPI_Bcast()`. Through point-to-point message passing (`MPI_Send()`), the root sends a portion of samples to each process, and the results are obtained via a receive operation `MPI_Receive()`. We divide the samples into (process number - 1) portions, which means each worker except the root will get the same number of samples. The remaining samples that have not been sent will be distributed to the root. The parallelism is achieved by having different workers calculating the new population of prey and predator for different samples simultaneously. To summarise, line 29 in Algorithm 4 is parallelised in MPI as below:

```
MPI_Bcast(model parameters $\theta$, each worker);

MPI_Bcast(time interval $\Delta t$, each worker);

IF ROOT

    MPI_Send(SAMPLENUM/(WORKERNUM-1) samples' value, each worker)

    FOR(i $\leftarrow$ (SAMPLENUM/(WORKERNUM-1))*WORKERNUM to SAMPLENUM )

        Gillespie_LV12(prey[i], predator[i], $\Delta t$, $\theta$)

    MPI_Receive(SAMPLENUM/(WORKERNUM-1) samples with new calculated preys and
predators' value, each worker)

ELSE IF WORKER
```

```
    MPI_Receive(SAMPLENUM/(WORKERNUM-1) number of samples' value, ROOT)

    FOR(i ← 0 to SAMPLENUM/(WORKERNUM-1)))

        Gillespie_LV12(prey[i], predator[i], $\Delta t$, $\theta$)

    MPI_Send(SAMPLENUM/(WORKERNUM-1) samples with preys and predators' value,
ROOT)
```

5. Results and discussion

With parallel implementation, we tested the application on Spark and MPI in HPC cluster. Our main focus is to compare the two platforms in aspects of performance, scalability, fault tolerance, dynamic resource handling, resource consumption, etc. In particular, we intend to verify automatic fault tolerance and dynamic resource handling features of Spark, as compared with MPI.

Additional specifications of the Spark and MPI applications used in our test are listed in Table 5.1. These factors may also affect the test results.

Table 5.1. *Information of the platforms used in the project*

Spark application	Platform	Spark 1.6.0
	Programming language	Scala 2.10.6
	Build tool	sbt 0.13.9
	Java	jdk 1.8.0_60
MPI application	Platform/Compiler	OpenMPI 1.6.2
	Programming language	C
	Compiler	gcc 4.4.6

To improve the robustness of the test results, we run the tests run multiple times and calculate the average of the results excluding outliers. In addition, whenever possible, we run Spark and MPI on the same nodes.

5.1 Performance and scalability

When testing the Spark and MPI applications, we used the same parameters for both, that is, 1000 samples in the particle filter, and 1000 iterations in the Metropolis-Hastings algorithm.

Figure 5.1 illustrates the computing time in relation to number of cores. MPI application has a clear advantage over Spark when less than 64 cores are used. However, there is a trend breaker at 64 cores when performance of the MPI application deteriorates drastically and therefore overtaken by Spark.

On the other hand, the computing time of the Spark application keeps dropping, and shows a converging trend from 48 cores. Between 64 and 96 cores, the Spark application is a bit faster than the MPI application.

Overall, performance of Spark application is positively correlated with number of cores, although the marginal improvement decreases significantly. We can draw the conclusion that MPI clearly outperforms Spark as it uses half of the time with only 48 cores compared to Spark with 96 cores. A possible reason for this is the performance advantage of C over Java platform (Spark is written in Scala and runs on JVM). Besides, there is only one `map` operation on the RDD of the samples before the result is fetched from memory and delivered to the driver node. This means in-memory computing of Spark is not fully exploited in the application.

As discussed before, when over 48 cores are used, adding more cores to MPI application only slows it down, and we typically call this issue "parallel slowdown". A possible explanation is

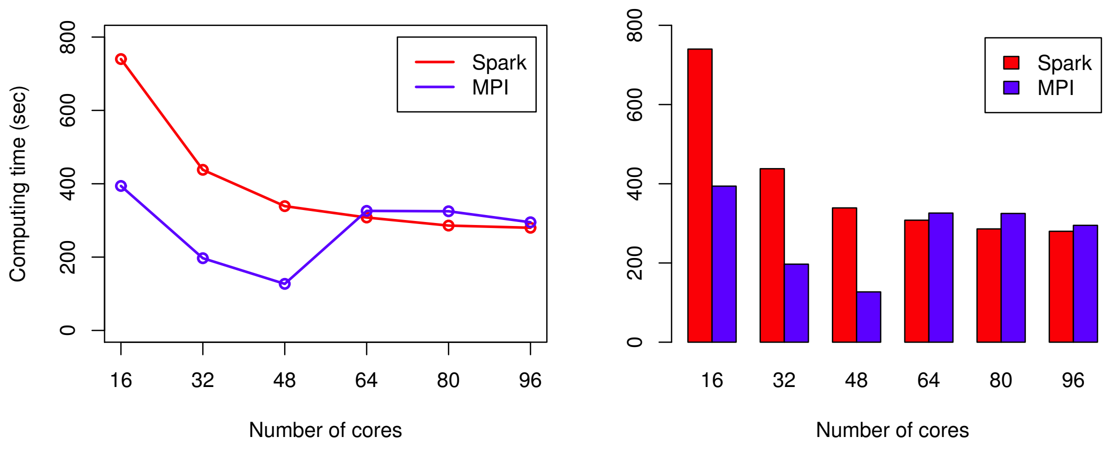


Figure 5.1. Scalability and computing time of the Spark and MPI applications, with 1000 samples used in the particle filter

that after certain point, the overhead of the parallelism outweighs the incremental acceleration by adding more computing resources. Due to absence of built-in monitoring tool for MPI, the exact cause of the slowdown cannot be verified. Whereas in the case of Spark, its web interface allows us to track the amount of time spent on each job and stage. Therefore, we picked two random stages to collect statistics regarding the breakdown of total running time, one run by 80 cores and the other by 96. The results are listed in Table 5.2. As we can see, the average computing time drops by only $1 - \frac{65.83}{74.93} \approx 12.14\%$ when number of cores increase by 20% ($\frac{96}{80} - 1 = 20\%$). Besides, there is an increase of scheduling delay by $30.80 - 27.03 = 3.77$ *ms* on average. As a result, in total 20% (16) more cores only gives reduction of 5.2% (5.33 ms) in average task time. Similar results are obtained with other stages. Although the Spark application has not shown uptick in Figure 5.1, we expect there to be a trigger point (after 96 cores) for a parallel slowdown.

Table 5.2. *Statistics on breakdown of total running time for two random Spark stages*

	80 cores	96 cores
Number of tasks	240	288
Total computing time (ms)	17982	18960
Total scheduler delay (ms)	6487	8871
Average computing time (ms)	74.93	65.83
Average scheduler delay (ms)	27.03	30.80
Average computing time + scheduler delay (ms)	101.96	96.63

Although figure 5.1 indicates limited scalability of both applications when 1000 samples are used in the particle filter. As sample size increases to a certain level (10000 and 20000 samples), we do observe linear speedup from 16 to 96 cores (see the right subplot of Figure 5.2). Therefore, with suficient computation distributed over the cluster, good scalability can be achieved.

5.2 Fault tolerance

Fault tolerance has become increasingly important for large-scale computing platforms. Some applications can take up hundreds even thousands of CPUs and other resources for running compute-intensive jobs. During the execution period, if some jobs cannot recover or continue in case of failure, the application will abort and has to be started over again. Very often, there could also be

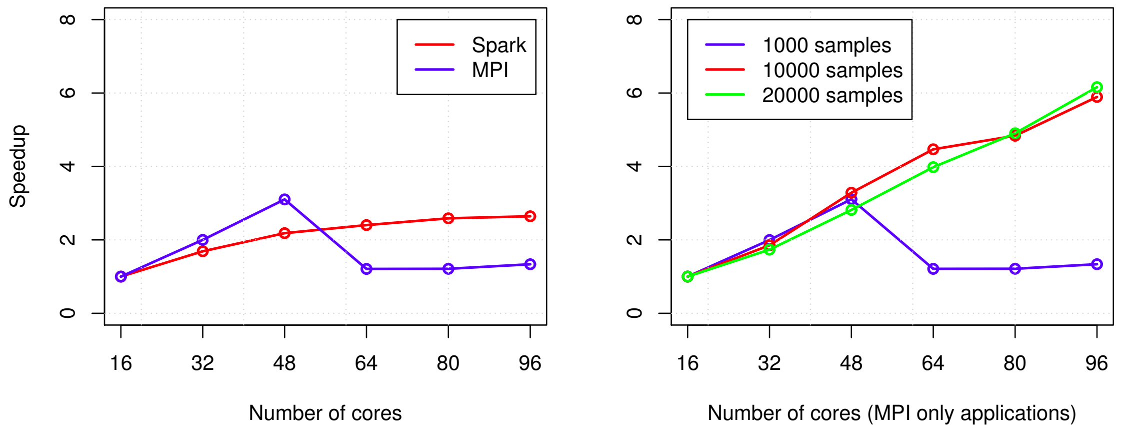


Figure 5.2. To the left: speedup of the Spark and MPI applications, with 1000 samples used. To the right: speedup of the MPI application based on different sample sizes.

resource hanging issues which need to be fixed manually. Therefore, such incidents can be very costly to enterprises both in monetary terms and beyond.

There are a few fault tolerance methods supported by Open MPI based on technologies such as algorithm-based fault tolerance, replication, message logging, checkpoint/restart etc. A few examples are CiFTS FTB (Coordinated infrastructure Fault Tolerance System, Fault Tolerance Backplane), BLCR (Berkeley Lab Checkpoint/Restart), OPAL_CRS (Open PAL MCA Checkpoint/Restart Service (CRS)) etc. CiFTS FTB is a project providing fault tolerance capabilities for different software components in HPC environment. It is a MPI independent library that suits for different platforms. The first FTB-enabled Open MPI is release 1.5.2. CiFTS FTB requires separate installation, which is a time consuming process on computer servers in large Enterprise, therefore we decided to skip this option. BLCR is another MPI fault tolerance implementation which is also supported by Open MPI. It is developed by Berkeley Lab's Future Technologies Group (FTG). However, it is out of functionality in the latest Open MPI releases. Another option is to use OPAL_CRS. It is a user-transparent checkpoint/restart service (CRS) for Open MPI. It ships two types of CRS: self CRS Component and BLCR CRS Component. Unfortunately, this method cannot be utilised in our test as the Open MPI installation used for the thesis work has build error with checkpoint component. Therefore, there are various technical challenges in applying the fault tolerance methods supported by Open MPI.

On the other hand, Spark comes with a functionality of automatic fault tolerance. As is claimed, all the storage levels provide full fault tolerance by recomputing lost data. This indicates that if a worker node is down or loses connection with its Spark master, the tasks that have not completed on that node can be passed on to other nodes, enabling the application to keep running without inaccuracy. As this feature is embedded in Spark, there is no extra work needed on the users' end.

In order to validate the claim, we set up a test to see how MPI and Spark react to node failure. For Spark, we run an application on two nodes - Node 6 and Node 9, each with 16 cores. This is to simplify the graphs and history in the webUI. To simulate a node failure, we removed Node 6 by killing the Spark slave process on it when Job 202 was running on both nodes. This incident can be seen in the event timeline in Figure 5.3 (Executor 15 is Node 6). Figure 5.4 gives more details of the event. The webUI makes it easier for the users/cluster administrators to take effective actions in time, given the details of the incident. More importantly, the application continued to run after the incident. By digging into the task history of Job 202 (see Figure 5.5), we found out that the 16 unfinished tasks on Node 6 were moved over to Node 9; each task started to run as soon as there was core available in Node 9. As there was only Node 9 (16 cores) available, the time spent on the next job (Job 203) was about twice as much as that on Job 200/201 when both Node 6 and 9 were available (32 cores), as shown in Figure 5.3.

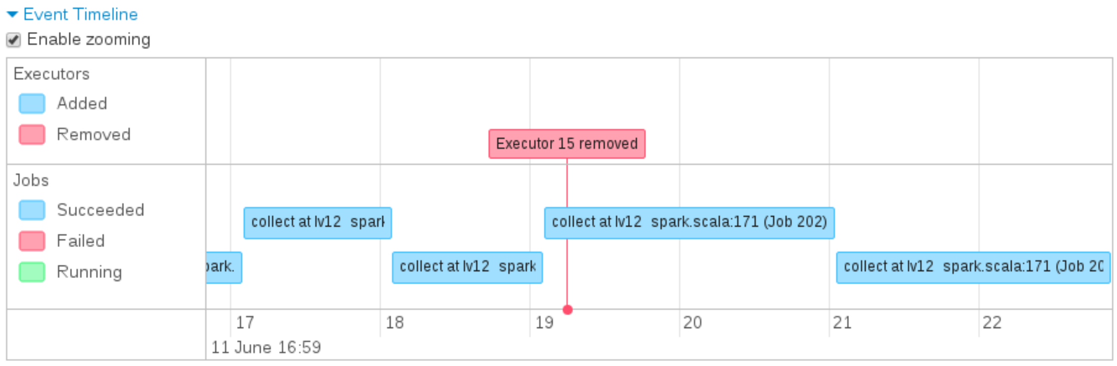
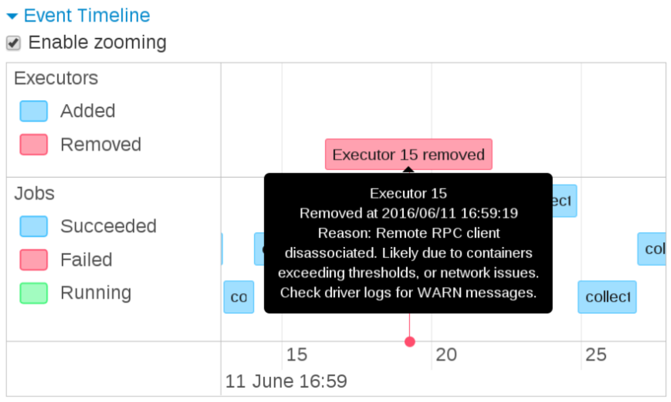


Figure 5.3. A node removal is shown on the event timeline in Spark's web interface



Executor 15
Removed at 2016/06/11 16:59:19
Reason: Remote RPC client disassociated. Likely due to containers exceeding thresholds, or network issues. Check driver logs for WARN messages.

Figure 5.4. Details of the event when hovering over the highlighted area in Spark web interface

Whereas for MPI, after we manually killed the running MPI process on one of the nodes, the whole MPI application crashed (see Figure 5.6).

From the tests, we confirmed that Spark's automatic fault tolerance feature is well functioning, at least for the application in our project. More specifically, the application can keep running if some of the worker nodes are down. Lost partitions can be recovered by recomputing the logged lineage information, and unfinished tasks on the failed nodes can be re-distributed to other available worker nodes to get re-executed. Fault tolerance in MPI could be achieved through independent library or dedicated checkpoint component. However, independent fault tolerance library requires additional installation and configuration, while self-supported checkpoint is in an experimental phase which involves extra user code. Furthermore, self-supported methods may go non-functional due to lack of maintenance resources. We conclude that the fine-grained fault tolerance feature of Spark is superior to that of MPI.

5.3 Dynamic resource handling

Earlier we discussed situations when certain nodes are down/removed. To the contrary, nodes being recovered/added in the cluster is another scenario that often happen in an enterprise. If a large compute-intensive application can make use of recovered/added nodes, total running time can be shortened by making better utilisation of the available resources. We call this feature dynamic resource handling, which is embeded in Spark as it claims.

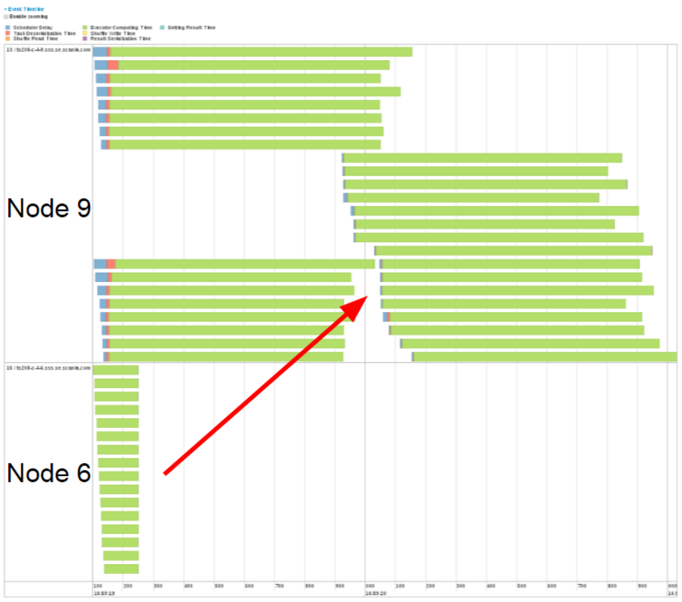
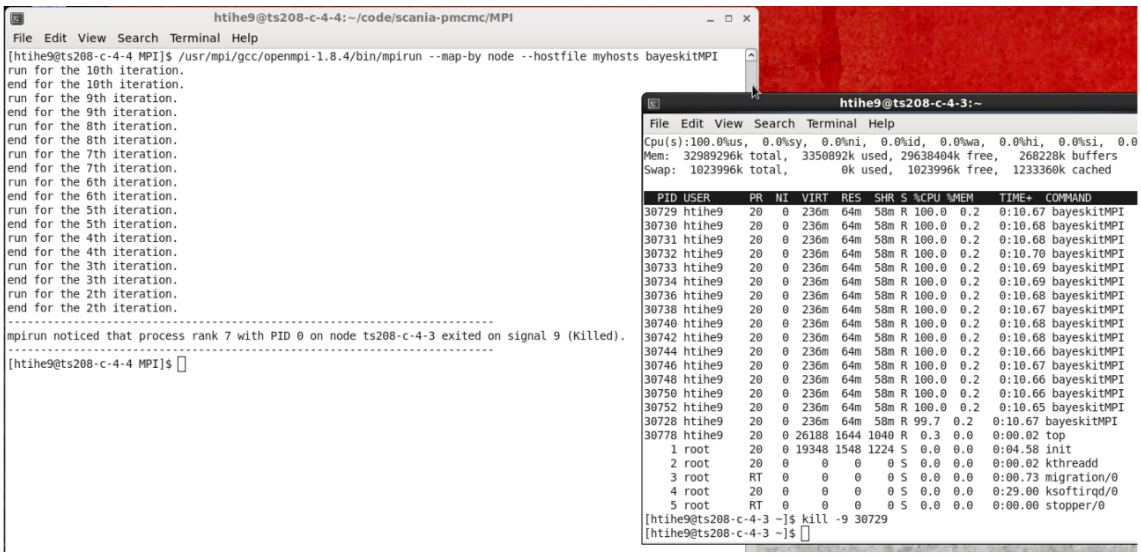


Figure 5.5. Unfinished tasks in a disconnected node were re-executed on the other available node


htihe9@ts208-c-4-4:~/code/scania-pmcmc/MPI

```
[htihe9@ts208-c-4-4 MPI]$ /usr/mpi/gcc/openmpi-1.8.4/bin/mpirun --map-by node --hostfile myhosts bayeskitMPI
run for the 10th iteration.
end for the 10th iteration.
run for the 9th iteration.
end for the 9th iteration.
run for the 8th iteration.
end for the 8th iteration.
run for the 7th iteration.
end for the 7th iteration.
run for the 6th iteration.
end for the 6th iteration.
run for the 5th iteration.
end for the 5th iteration.
run for the 4th iteration.
end for the 4th iteration.
run for the 3th iteration.
end for the 3th iteration.
run for the 2th iteration.
end for the 2th iteration.
--------------------------------------------------------------------------
mpirun noticed that process rank 7 with PID 0 on node ts208-c-4-3 exited on signal 9 (Killed).
--------------------------------------------------------------------------
[htihe9@ts208-c-4-4 MPI]$ 
```

htihe9@ts208-c-4-3:~

```
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0
Mem:  32989296k total,  3350892k used, 29638404k free,   268228k buffers
Swap:  1023996k total,        0k used,  1023996k free,  1233360k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
30729 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.67 bayeskitMPI
30730 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.68 bayeskitMPI
30731 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.68 bayeskitMPI
30732 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.70 bayeskitMPI
30733 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.69 bayeskitMPI
30734 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.69 bayeskitMPI
30736 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.68 bayeskitMPI
30738 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.67 bayeskitMPI
30740 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.68 bayeskitMPI
30742 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.68 bayeskitMPI
30744 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.66 bayeskitMPI
30746 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.67 bayeskitMPI
30748 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.66 bayeskitMPI
30750 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.66 bayeskitMPI
30752 htihe9    20   0  236m  64m  58m R 100.0  0.2   0:10.65 bayeskitMPI
30728 htihe9    20   0  236m  64m  58m R 99.7  0.2   0:10.67 bayeskitMPI
30778 htihe9    20   0 26188 1644 1040 R  0.3  0.0   0:00.02 top
    1 root      20   0 19348 1548 1224 S  0.0  0.0   0:04.58 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.02 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.73 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:29.00 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 stopper/0
[htihe9@ts208-c-4-3 ~]$ kill -9 30729
[htihe9@ts208-c-4-3 ~]$ 
```

Figure 5.6. The MPI application crashed after the node failure

For MPI, we learned from literature review a method called dpm (dynamic process management) which provide means to dynamically increase processes for large scaled job. It is introduced since MPI-2, using spawn technology to create child processes from the main processes (parent processes). It is initiated by calling MPI_Comm_spawn in the parent processes, number of identical copies of the MPI job will be started and the spawned processes is called as children. The children have their own MPI_COMM_WORLD, and intercommunicator results will be returned when MPI_Init is called in the children. So MPI_Comm_spawn in the parents and MPI_Init in the children form the command pair to start the spawn and collect the result. Figure 5.7 shows the work flow of spawn process. Due to time constraints, we did not cover this method in our tests but prioritised investigation on Spark in this section.

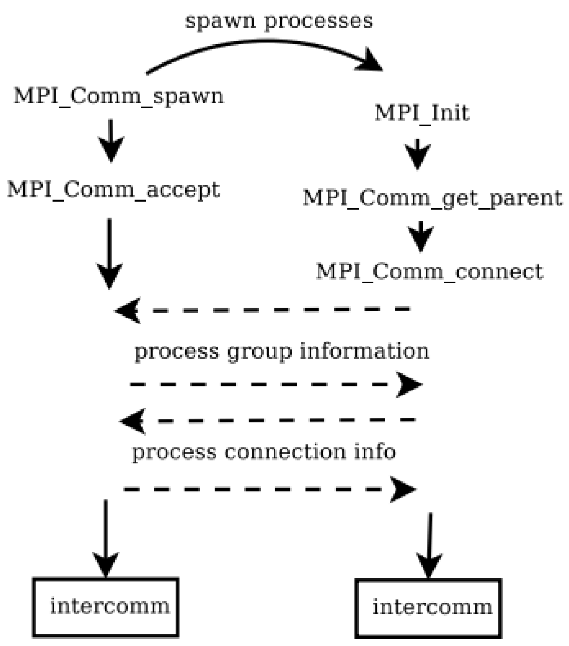


Figure 5.7. Work flow of MPI spawn process

Continuing from the test session in Section 5.2, we connected Node 1 (Executor 17) to the Spark master a few seconds after Node 6 (Executor 15) was removed. When Job 210 was just started, 16 tasks was assigned to Node 9, the only connected slave at that time. Right after Node 1 was added, the remaining 16 tasks of Job 210 were assigned to it. The timeline of all the 32 tasks is shown in Figure 5.9.

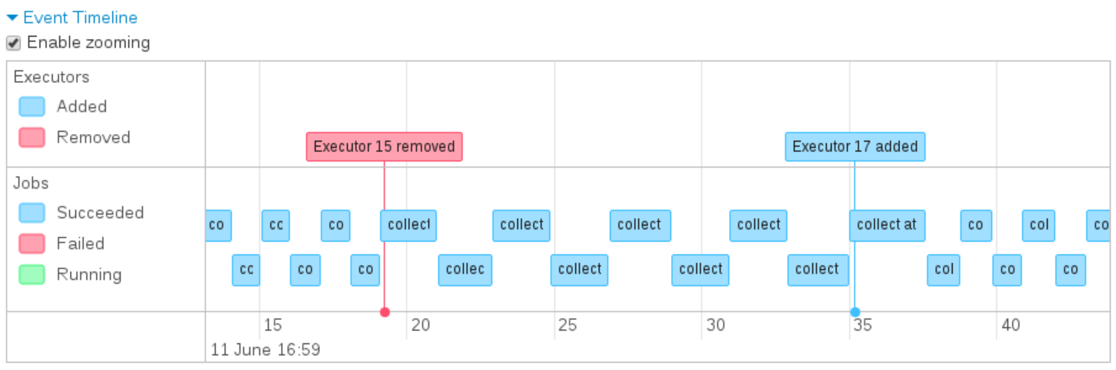


Figure 5.8. A node being added is shown on the event timeline in Spark's webUI

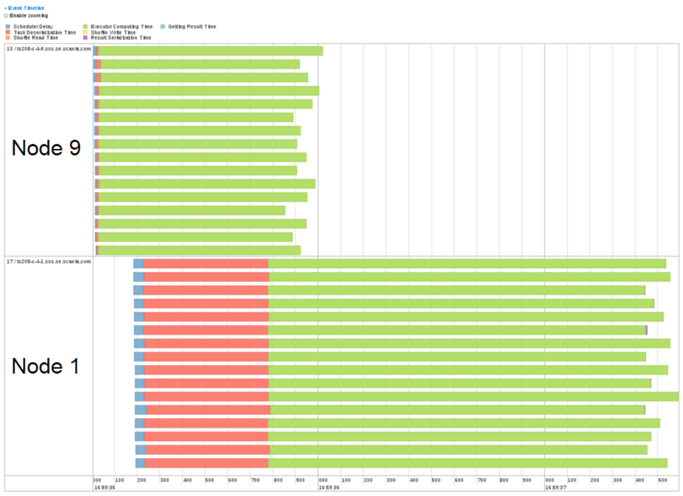


Figure 5.9. Timeline of the 32 tasks in Job 210 on Spark worker nodes

Interestingly, scheduler delay (blue bar), task deserialization time (red bar) and executor computing time (greenbar) of all tasks on Node 1 were much longer than those on Node 9. But for the next jobs, we found no big difference in task time between Node 1 and 9. Figure 5.10 shows the task timeline of one such job. We can draw similar conclusion from Figure 5.8: after Job 210, each job finished roughly in the same execution time as the jobs when Node 6 and 9 were both active. Detailed statistics in the web interface also confirmed this phenomenon.

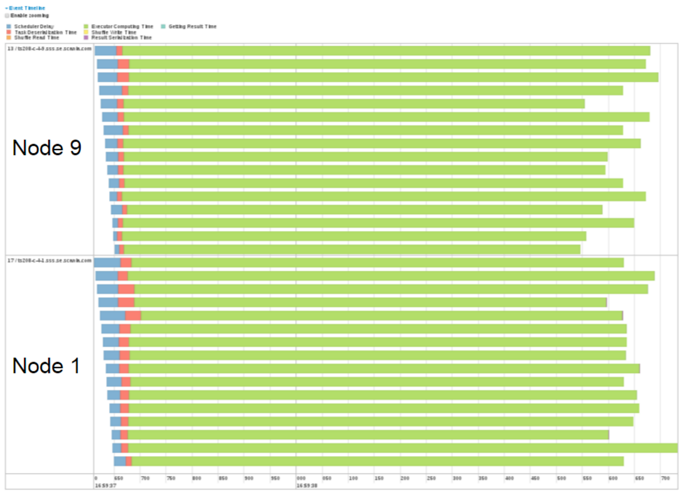


Figure 5.10. Timeline of a job after Job 210 on Spark worker nodes

To summarise, as is confirmed in our tests, Spark can detect newly added nodes and use them immediately in running applications. Although its processing time is longer than normal when the new node is first added. It runs smoothly and stably in the subsequent jobs. This can be seen as the node has "blended" into the Spark platform.

We continued to perform the test by removing and adding nodes while the Spark application was running. The complete timeline of the whole test is shown in Figure 5.11. We can see that neither the application nor the Spark platform crashed during the intensive test. And the Markov chain produced by the application also met our expectation. This gives us promising evidence that Spark can handle dynamic resources quite well.

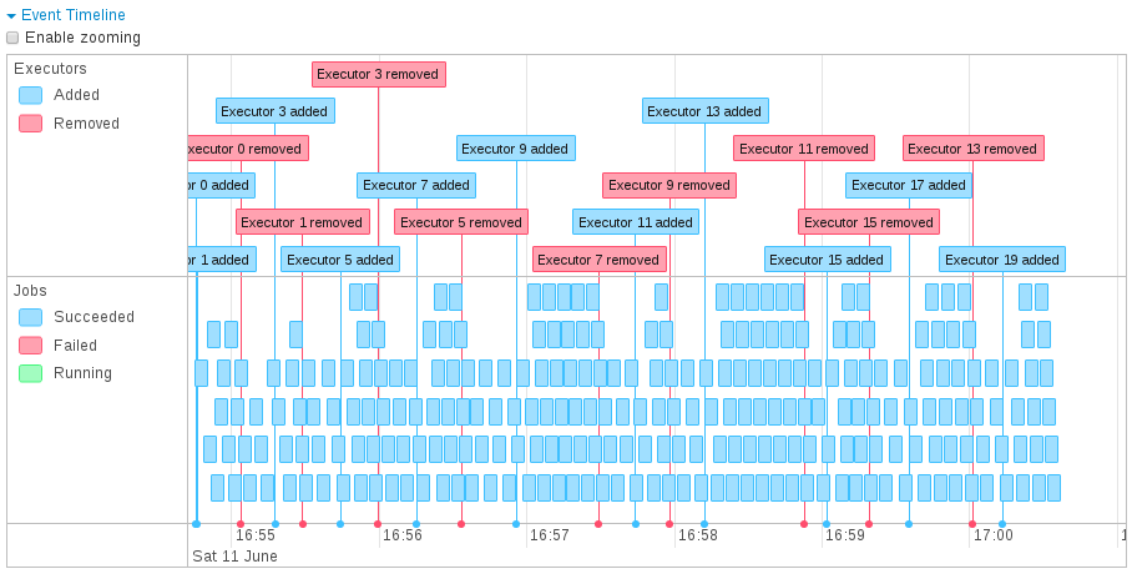


Figure 5.11. The Spark application was successfully completed, nodes removing and adding were happening continuously during its execution

5.4 Resource consumption

For computing platform, resource consumption is a key factor of concern in enterprises. In this project, one of our main interest area is to compare the resource consumption of Spark and MPI as computing platform, using a compute-intensive application.

We use Ganglia - a cluster monitoring tool to log the resource consumption for both of the applications. Each test session was exclusive to the application in order to minimise the impact from other running programs on the test results.

5.4.1 CPU consumption

We run the Spark application multiple times among 5 nodes (80 cores in total): Node 1 to 4 and Node 6, where Node 1 served as both master and worker node. The results are given in Figure 5.12. Except for the last four sessions, we used 1000 samples in the particle filter of the application. As we can observe, in the test sessions with 1000 samples, the maximum CPU usage was about 55-66% among all the nodes, and there was no big difference between Node 1 (master + worker node) and pure worker nodes. The task timeline shown in Figure 5.13 indicates why the remaining part of CPU was not used: a portion of the time was dedicated to scheduling and task deserialization, and the CPUs that finished tasks early with no new tasks assigned had to stay idle until the last task in the stage was done.

We then increased sample size to 500000 and immediately observed higher CPU usage, as represented by the peaks in the last few sessions in Figure 5.12: 83-87% in two sessions and 97-100% in one. The eventline of one stage in such a session is shown in Figure 5.14. The scheduling and task deserialization overhead is negligible compared with the long computing time of each task. Similarly, there was waiting time for CPUs that finished early. In short, CPU usage can be raised by prolonging the computing time of each parallel tasks.

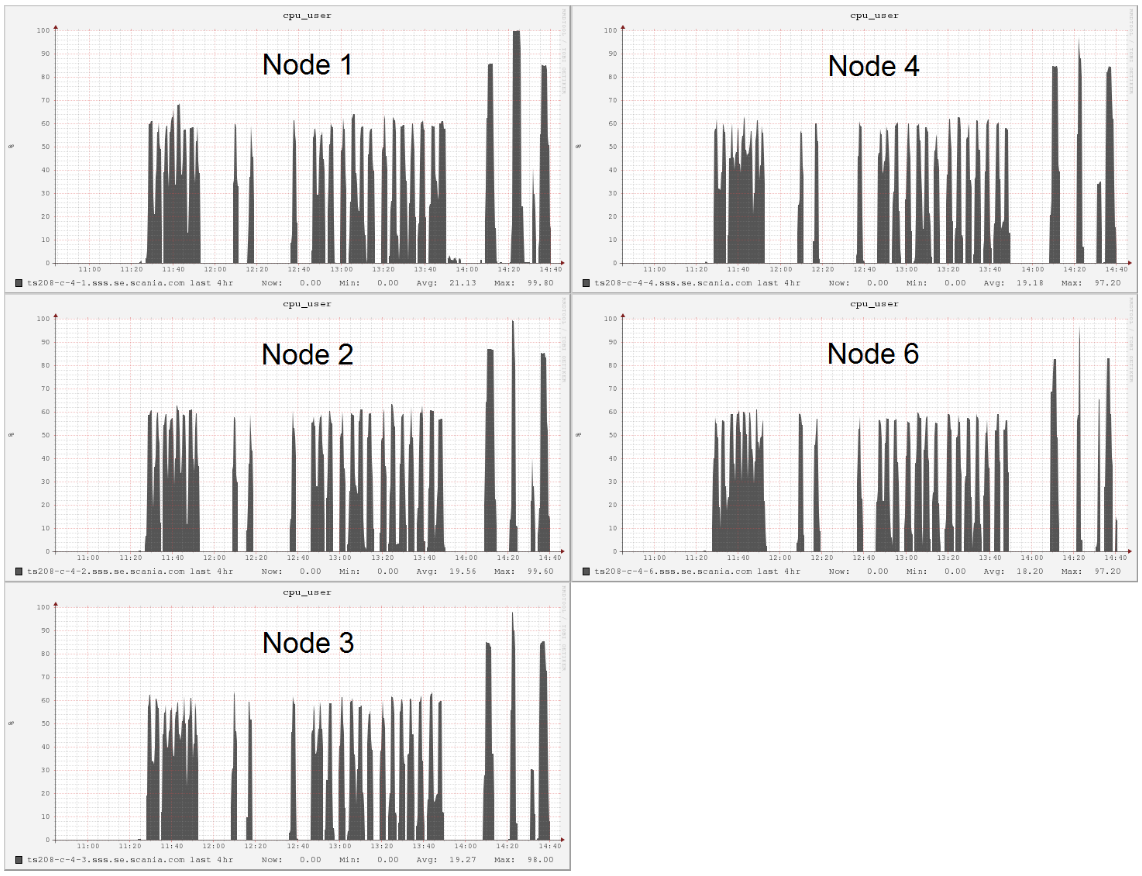


Figure 5.12. CPU usage of the Spark application when all 80 cores of 5 nodes were used, first with sample size of 1000 and then 500000 in the particle filter

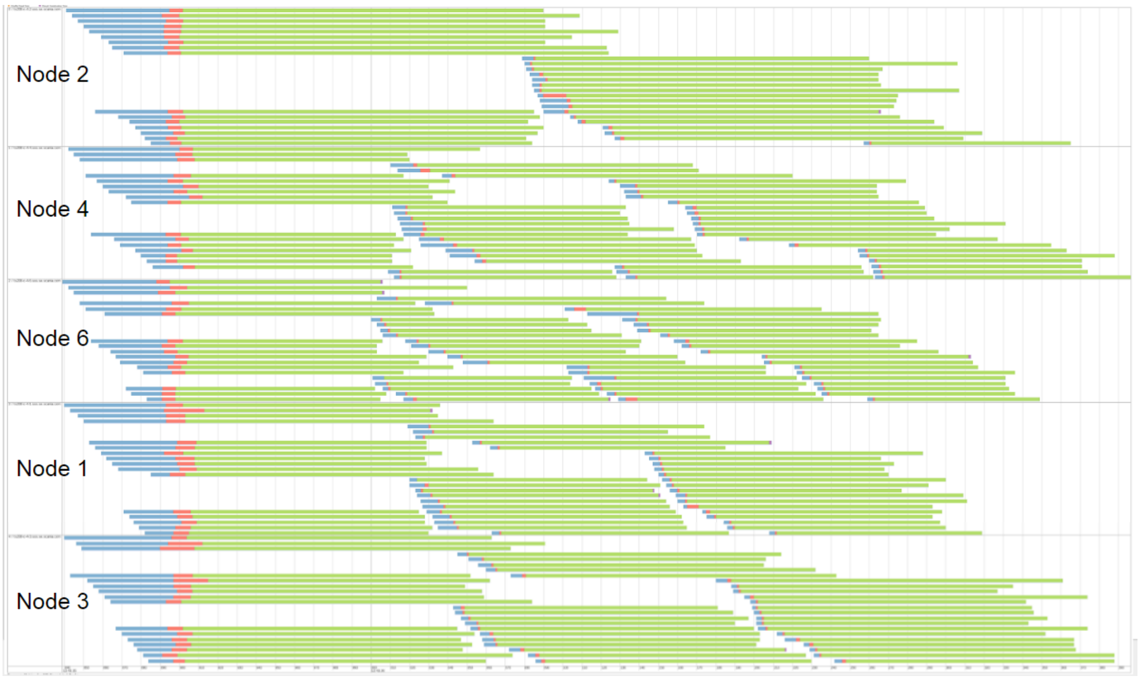


Figure 5.13. Task timeline of one stage when 1000 samples were used in the Spark application; the CPU usage was between 55-66% with this setting

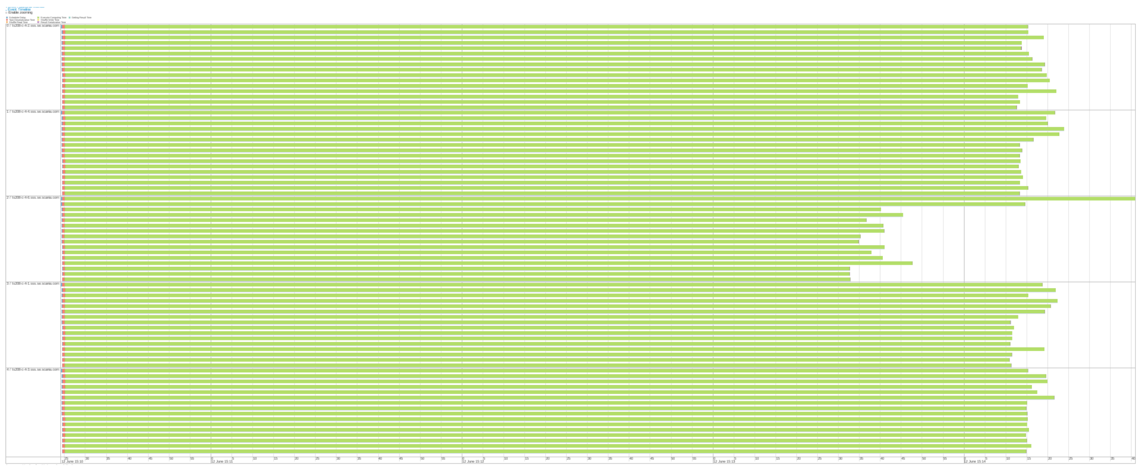


Figure 5.14. Task timeline of one stage when 500000 samples were used in the Spark application; the CPU usage was between 83-100% with this setting

Our observations are also validated by the test results of the MPI application in Figure 5.15. With two to three nodes used, the MPI application consumes more CPU resources as sample size increases from 1000 to 200000.

With the same setting (1000 samples and 5 nodes) as Spark, the MPI application reached up to 100% CPU usage, while the Spark application consumed only 55-66%. In this case, the MPI application has better CPU resources usage than Spark, which may explain its superior performance (see Section 5.1).

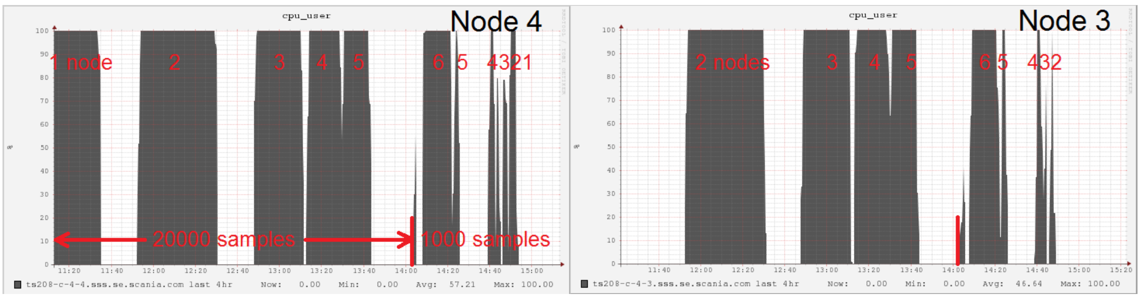


Figure 5.15. CPU usage of the MPI application when different number of cores and sample size were used. The usage of other worker nodes was similar to Node 3 when all their cores were used, and hence was not included in the graph.

5.4.2 Memory consumption

Following previous section on CPU consumption, we also looked into the memory consumption of both applications. For the MPI application, both Node 4 (root + worker) and Node 3 (pure worker) used about 0.5 GB memory in most of the sessions. The memory usage was only slightly impacted by test settings (different combination of cores and samples), as indicated by Figure 5.16. Considering one core was dedicated to root process and the other 15 cores to worker processes in Node 4, the root process used about $0.5 \div 16 = 0.03125$ GB = 32 MB memory resource. We believe this part is dedicated for root-process-only computations and operations (e.g. message passing).

As for Spark, similarly we monitored the memory usage in the first eight sessions in Figure 5.12, with 1000 samples and 5 nodes (80 cores) used. Gradually, we increased the memory on each

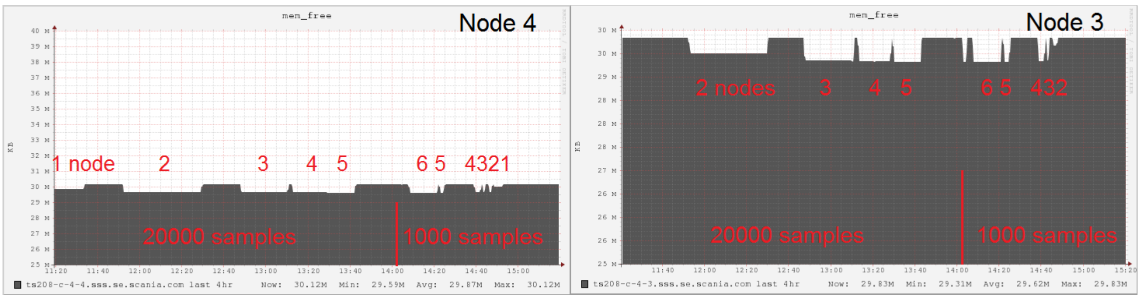


Figure 5.16. Memory usage of the MPI application when different number of cores and sample size were used. Not all worker nodes are included in this graph, but their memory consumption was similar to Node 3

node from 1 GB to 4 GB, each time by 1 GB. This was achieved by setting the parameter manually `spark.executor.memory`. The results are shown in Figure 5.17 and Table 5.3. And we found three interesting points in the results.

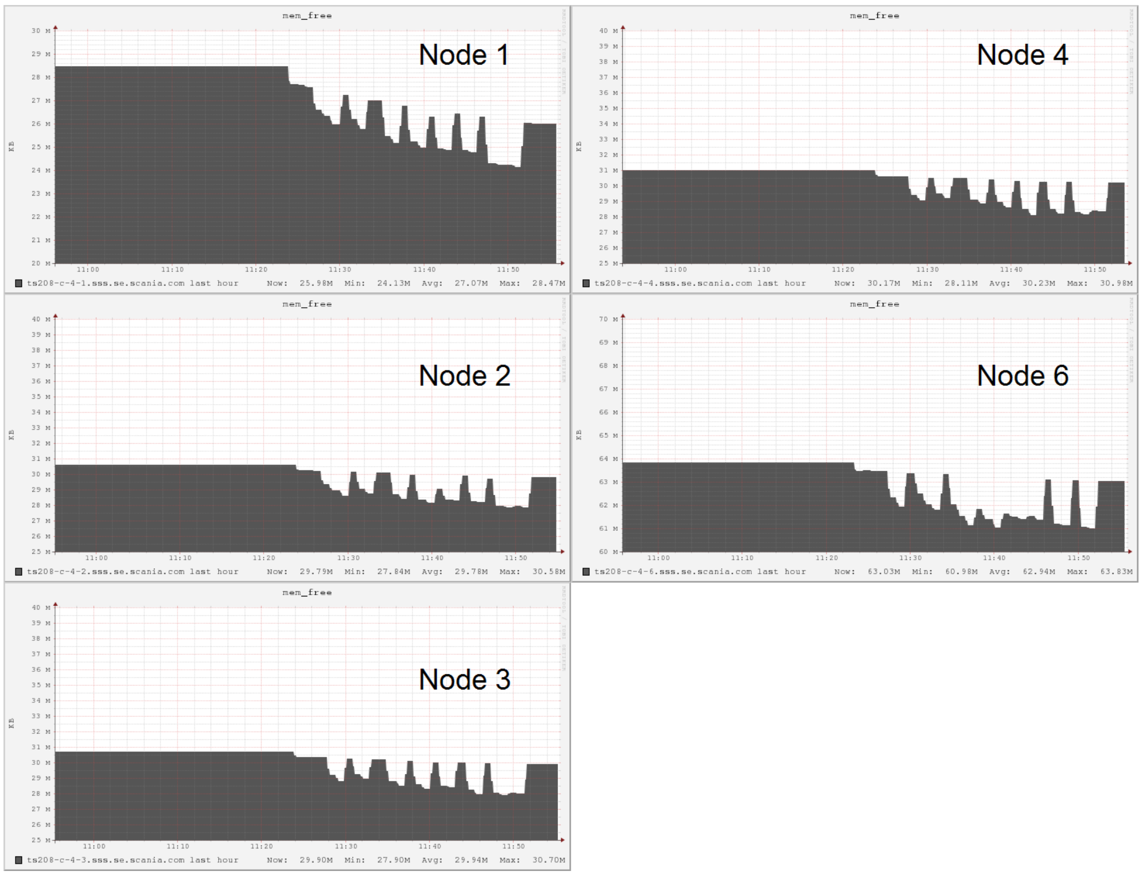


Figure 5.17. Memory usage of the Spark application when all 80 cores of 5 nodes were used, with sample size of 1000 in the particle filter

First, the actual average memory usage on each pure worker node was not the same as the setting in `spark.executor.memory`. This parameter is said to control the Java heap size in the official document of Spark 1.6.0[1] and more specifically the "maximum" Java heap size in the documentation of Spark 2.0.0[2]. The actual allocated Java heap size could be smaller than the maximum value and that is why the actual memory usage was lower than the value of `spark.executor.memory` when it was set to 3 and 4. Besides, the Java heap seems to be only a part of the memory used in a worker

[1]`http://spark.apache.org/docs/1.6.0/configuration.html`, accessed 2016-09-10
[2]`http://spark.apache.org/docs/2.0.0/configuration.html`, accessed 2016-09-10

Table 5.3. *Memory usage of the Spark application, calculated from the data provided by Ganglia*

`spark.executor.memory` (GB)	Memory usage on the master+worker node(GB)	Average memory usage on each pure worker node (GB)	Estimated memory usage by master only (GB)
1	2.52 - 2.72	1.75 - 2.02	0.50 - 0.97
2	3.32 - 3.52	2.07 - 2.78	0.54 - 1.45
3	3.65 - 3.71	2.28 - 2.88	0.77 - 1.43
4	4.24 - 4.34	2.68 - 2.87	1.37 - 1.66

node, so actual memory usage can be more than the bound set by `spark.executor.memory` (examples are the test sessions with `spark.executor.memory` set to 1 and 2).

Second, although the memory usage on worker node was not the same as `spark.executor.memory`, it was still positively related with `spark.executor.memory`. However, the upper bound of the usage did not increase with `spark.executor.memory` for settings beyond 3 GB. This might imply the maximum memory required by each worker node.

Third, the memory usage of Spark master was estimated by subtracting the average memory usage on pure worker node from the memory usage on Node 1 (served as both master and worker). The memory usage of Spark master also increased with `spark.executor.memory`, though it was not directly controlled by this parameter. We have not come up with a good explanation to this, but this phenomenon is worth noticing, together with that the memory usage of Spark master was about half of the worker node in these test sessions.

In comparison, the MPI application has obviously used much less memory. When executor memory was set to 1 GB, Spark slave node consumed 3 to 4 times as much memory as MPI worker node. This gap could be bigger if we set the executor memory to a bigger value. A possible reason for the big gap is that a lot of memory in Spark executors is dedicated to special use on Spark platform. This topic will be covered in Section 5.6.3 below. Furthermore, Spark master consumes a lot more memory than MPI root, since it manages all the nodes in the cluster instead of simply sending and receiving messages like MPI root.

5.4.3 Network usage

In the same test sessions of the MPI application in previous sections, we acquired some further insights on network usage (see Figure 5.18) in Node 4 (root+worker) and Node 3 (pure worker).

First, the network usage of both nodes first reached a peak, then quickly dropped to a much lower level until the end of each test run. We suspect that some initial setup between the nodes caused the peak usage at the beginning of each session.

Second, the maximum network usage by Node 4 linearly increased as more nodes were added, both in receiving packets and in sending packets. Whereas maximum network usage by Node 3 did not show much correlation with the number of nodes in use. Our explanation earlier for the initial peak usage can also account for the linear relationship between maximum network usage and number of nodes for Node 4. To elaborate, for root node, the main workload on network is communication between the root process and worker processes. As more worker processes were brought in, the communication from and to the root process will for sure increase. For worker nodes, the load on network for initialisation would not change with the number of nodes.

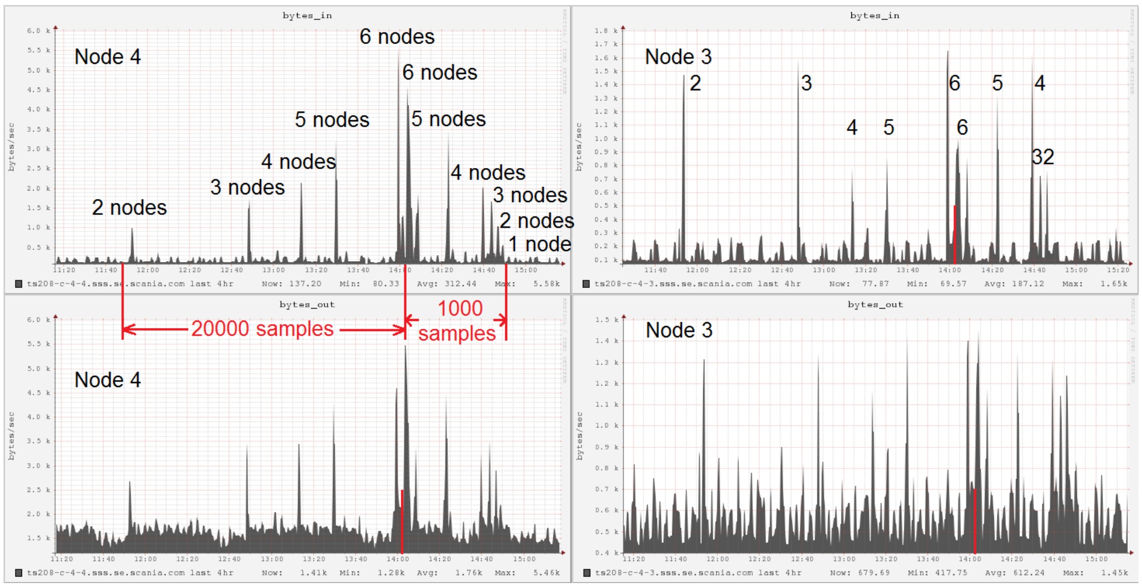


Figure 5.18. Network usage of the MPI application when different number of cores and sample size were used. Not all worker nodes are included in this graph, but their network usage was similar to Node 3 when all their nodes were used

Third, the stabilised network usage was much less than expected. The amount of data received and sent on all the nodes should be fixed: 50 (iterations) $\times$ 20000 (samples) $\times$ 5 (observations) $\times$ 3 (species) $\times$ 8 (bytes for address in 64-bit operating system) $\approx$ 114 MB. For the case of 5 nodes used, the network usage for Node 3 should at least be 114 MB $\div$ 5 $\div$ 794 (seconds; total running time) = 29 kB/sec (plus some other communication) at both receiving and sending end, which was much higher than the actual network usage in Figure 5.18. The big deviation from expectation also applies to the root+worker node. We expected the stabilised network usage (both sending and receiving) to be directly proportional to sample size and inversely proportional to the number of nodes. However, the results indicate almost fixed usage and the value was also far below expectation for the root+worker node. A possible explanation is that the sample data was not communicated through the network that Ganglia monitored, as Open MPI supported multiple transport networks including various protocols over Ethernet (e.g., TCP, iWARP, UDP, raw Ethernet frames, etc.), distributed shared memory, InfiniBand and so on [Squyres,]. For our test case, distributed shared memory technology was used in the Open MPI which might not be detectable by Ganglia.

We monitored the network usage of the same 14 test sessions on the Spark application, with 1000 samples, 5 nodes (80 cores) and executor memory set to 1 GB. The results are shown in Figure 5.19 and 5.20. Unlike the MPI application, all the nodes in the test sessions stayed at the maximum level until the usage dropped to 0. The only exception is the receiving end of Node 1 (master + slave), where the usage stayed at a relative low level for some unknown reasons before reaching the peak. The usage for either sending or receiving sample data on each worker node should be around 100 (iterations) $\times$ 1000 (samples) $\times$ 5 (observations) $\times$ 3 (species) $\times$ 8 (bytes for an integer in 64-bit operating system) $\div$ 5 (number of nodes) $\div$ 286 (seconds; total running time)$\approx$ 8 kB/sec. This only accounts for a very small portion of the total network usage of Spark slave node. The other usage could come from other purposes on the Spark platform such as communication between slave node and driver program or master, as well as transmission overhead.

In comparison, with the same setting (1000 samples, 5 nodes with 80 cores), MPI consumed much less network resource (on networks monitored by Ganglia), in both sending and receiving end, for either root+worker (master+worker in Spark's term) or pure worker nodes. The details

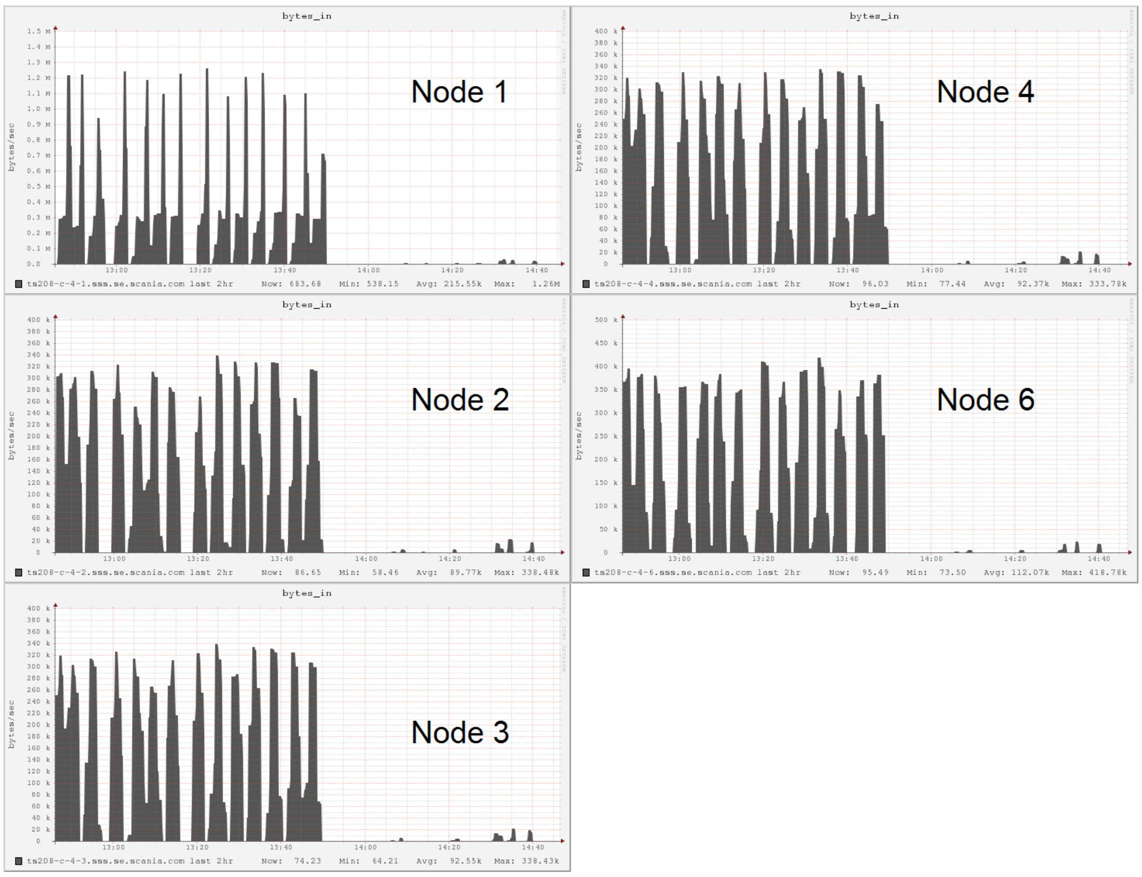


Figure 5.19. Network usage (receiving) of the Spark application when all 80 cores of 5 nodes were used, with sample size of 1000 in the particle filter

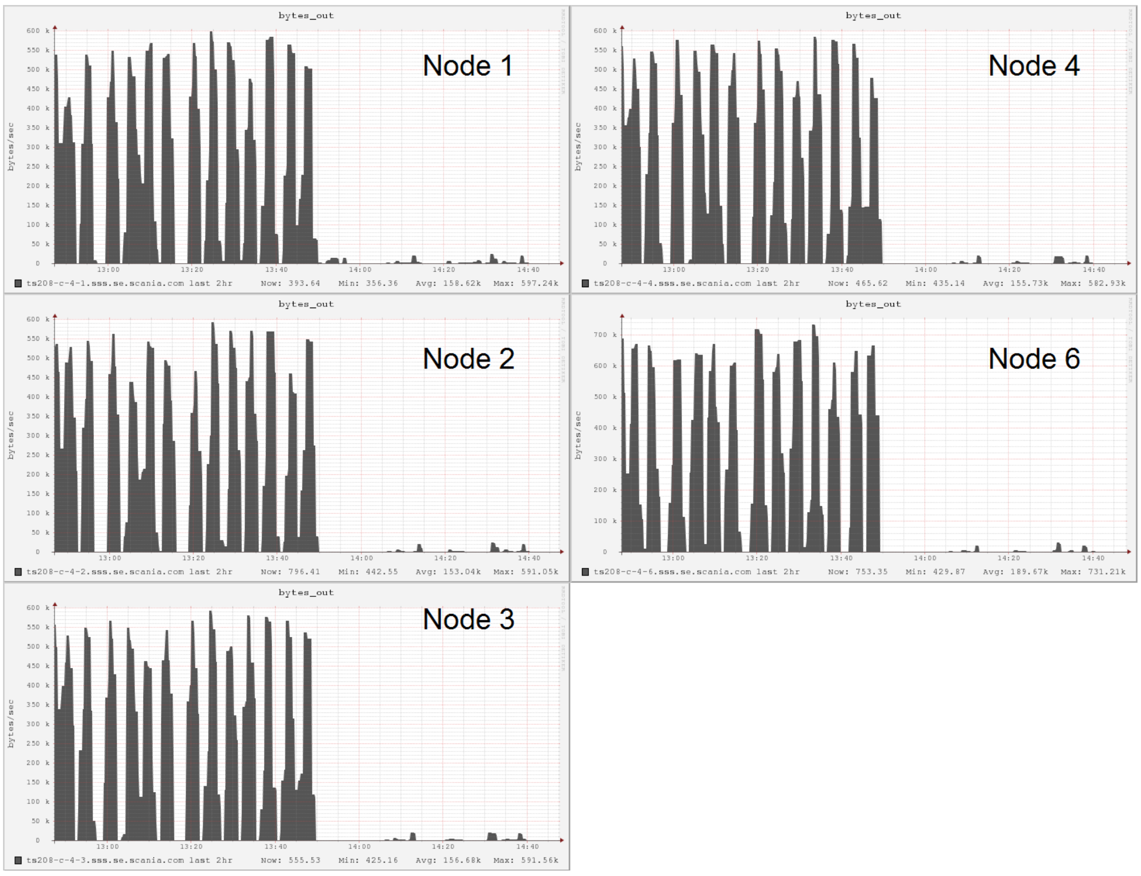


Figure 5.20. Memory usage (sending) of the Spark application when all 80 cores of 5 nodes were used, with sample size of 1000 in the particle filter

are summarized in Table 5.4. For each node, Spark used at least 293 times the amount of network resource MPI used. The biggest gap between Spark and MPI lies in the receiving side of the root+worker (master+worker) node. A possible reason might be the extra communication between the master node and the driver program and the state update received from worker node, which does not apply to MPI.

Table 5.4. *Network usage of the MPI and Spark applications*

	MPI	Spark	Spark/MPI difference
worker stable receiving usage (kB/sec)	0.05 - 0.35	250 - 340	971 - 5000
worker stable sending usage (kB/sec)	0.42 - 0.9	440 - 592	658 - 1048
root + worker node stable receiving usage (kB/sec)	0.12 - 0.48	963 - 1290	2688 - 8025
root + worker node stable sending usage (kB/sec)	1.30 - 2.05	430 - 600	293 - 331

5.5 Ease of use

Ease of use is an important factor for enterprises in selecting large-scale computing platforms. It affects not only the efficiency and ease of developing applications, but also the cost and complexity of maintenance afterwards.

In this project, we accumulated hands-on experience on both platforms by implementing the Bayesian inference of LV-12 model using both Spark and MPI. From our experience and official documents, we summarized some key factors that affect ease of development using Spark/MPI in Table 5.5. Below we will briefly elaborate on these key factors one by one.

Table 5.5. *Comparison between Spark and OpenMPI regarding ease of use*

	Spark	OpenMPI
Language supported	Scala, Java, Python, R	C, C++, Fortran
Parallelism mechanism	Data parallelism	Task parallelism
Monitoring	Spark WebUI	Third party software
Debug	hard	hard
Built-in libraries	SQL and DataFrames, machine learning, graph computing and streaming	None
Size of code (of the Bayesian inference application)	about 200 lines	about 600 lines

As listed in the table, Spark supports multiple higher level programming languages while OpenMPI only supports one high level language - C++, which is known for steeper learning curve than those supported by Spark.

The parallelism of Spark is mainly data parallelism over RDD, which makes it nearly impossible to run multiple different tasks in parallel. OpenMPI does not have this type of problem since it uses task parallelism, and it can also perform same tasks on different data.

As to job monitoring, we have had good experience using Spark's WebUI. It gives rich information at different levels of both the application and the cluster. We did not have time to test some third-party monitoring tools for OpenMPI, such as PERUSE [Keller et al., 2006] and VampirTrace[3]. But one thing for sure is that installing these monitoring tools requires extra work from the user.

When we implemented the application in Spark and MPI, we first wrote it in pure Scala and C, then adapted the particle filter part to Spark/MPI model. Debugging pure Scala and C programs is easy with the help of IDEs. However, once the application runs in Spark or MPI, debugging becomes harder because there were hardly any good debugging tools for Spark or MPI. The method we used was mostly reading the code or searching on the Internet with the error messages. In our opinion, better debugging support should be provided by both platforms to improve ease of use.

Last but not least, the biggest difference between Spark and MPI is that the former is a platform while the latter is a library. Therefore, Spark has a lot of components built inside the platform, such as Spark core, standalone scheduler, different libraries (SparkSQL, DataFrames, machine learning etc.), WebUI, etc. Spark takes care of all the details of parallel operations on RDD, including data transfer, parallel computing, resource scheduling, fault tolerance, monitoring, etc. On the contrary, users using MPI have to specify many low-level operations, for example which nodes to use, what message should be sent to which nodes, what operations must be performed on which nodes, at which point should a checkpoint be added etc. This no doubt increases the workload at the user side (also implied from the lines of code of our specific application), and raise the chance of error/failure.

To summarize, Spark is more attractive than MPI regarding ease of use. It helps the users to focus more on how to solve a problem itself instead of implementing the solution. Spark covers most of the details of the parallelism, efficient usage of the resources and other aspects, while MPI leaves these parts to the users. One weakness of Spark is that it cannot distribute different tasks to multiple nodes, as it uses data parallelism. On the other hand, MPI does not have good support for classical big data problems such as streaming processing, real-time analysis and so on.

5.6 Tuning the Spark application

There are a few parameters that can be tuned in both Spark[4] and OpenMPI[5]. We priortised Spark as the implementation of the OpenMPI application would take much longer time than our project timeframe.

5.6.1 Deploy mode

Deploy mode indicates where the driver program runs. In cluster mode, the driver program is inside the cluster, while in client mode the driver program is inside the node which is usually not part of the cluster. Since the driver sends all the tasks to the executors and receives the results from them, the connection between the driver and the executors can affect the performance of Spark applications. We run the Spark application in both cluster and client mode and their running time is shown in Table 5.6. In all cases, test sessions running in cluster mode finished faster than those

[3]`https://www.vampir.eu/`, accessed 2016-09-10
[4]`http://spark.apache.org/docs/1.6.0/tuning.html`, accessed 2016-09-10
[5]`https://www.open-mpi.org/faq/?category=tuning`, accessed 2016-09-10

in client mode, though the difference was not big. For applications with bigger task size and more data transferred between the driver and the executors, cluster mode will for sure help shorten the running time.

Table 5.6. *Running time of the Spark application in client and cluster modes, with 1000 samples and 100 iterations. Other settings were kept as default*

Number of cores	Average running time in client mode (sec)	Average running time in cluster mode (sec)
16	740	660
32	438	426
48	339	318
64	308	300
80	286	276
96	280	258
112	271	264

5.6.2 Number of partitions

As mentioned in Section 4.4.1, we used 3 $\times$ number of cores as the number of partitions for the Spark application, based on the programming guide of Spark and our tuning results. With a fixed number (80) of cores used, we tested the running time of the Spark application with different number of partitions. The test results are given in Table 5.7. It shows that when 3 partitions were distributed to each core on average, the running time would be shortest compared with using other number of partitions. When less partitions were used, the CPU resource was not well utilised as at the end of each stage, some CPUs had to wait for others to finish tasks when there were no tasks available to execute. When more partitions were used, extra overheads caused by more partitions led to longer running time. Therefore, 240 partitions can be seen as an optimal number of partitions that well balances between overall CPU usage and overheads brought by partitioning in this use case.

Table 5.7. *Running time of the Spark application when different number of partitions were used, with 80 cores used and other default settings*

Number of partitions	Average running time (sec)
80	312.0
160	324.4
240	313.6
320	305.4
400	315.8

5.6.3 Memory tuning

Since introduction of Spark 1.6.0, a new memory management model has been adopted. This model is briefly described in Spark's tuning guide, and more details can be found on the Internet[6][7].

[6]`https://0x0fff.com/spark-memory-management/`, accessed 2016-09-10

[7]`http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/`, accessed 2016-09-10

As illustrated in Figure 5.21, the job on each executor is a Java process, and the JVM heap is all the memory it can use. In the Java heap, 300 MB is reserved for other purposes out of Spark. The remaining part is divided into user memory and Spark Memory. The fraction of Spark memory is set though `spark.memory.fraction` and the default value is 0.75. User memory is used for storing user data structures, internal metadata in Spark, and safeguarding against OOM errors in the case of sparse and unusually large records. Spark memory is further divided into execution memory and storage memory. Execution memory is used for computation in shuffles, joins, sorts and aggregations, while storage memory is used for caching and propagating internal data across the cluster. The fraction of storage memory is set through `spark.memory.storageFraction` and the default value is 0.5. In the official documentation, it is recommended to leave the two parameters unchanged.

Let us look at a specific example. If the heap size of an executor is 1 GB, then the reserved memory is 300 MB. User memory is (1024-300) $\times$ (1 - 0.75) = 181 MB by default. Execution memory and storage memory are both (1024-300) $\times$ 0.75 $\times$ 0.5 = 271.5 MB by default. Execution and storage memory can share the memory to each other if their own part is free. Execution memory can evict storage memory until storage memory is below its pre-set size, but storage memory cannot evict execution memory due to some reason. This design aims to make better use of the memory and protect a fixed size of cache data from being evicted, and it proves to contribute to reasonable performance improvement.

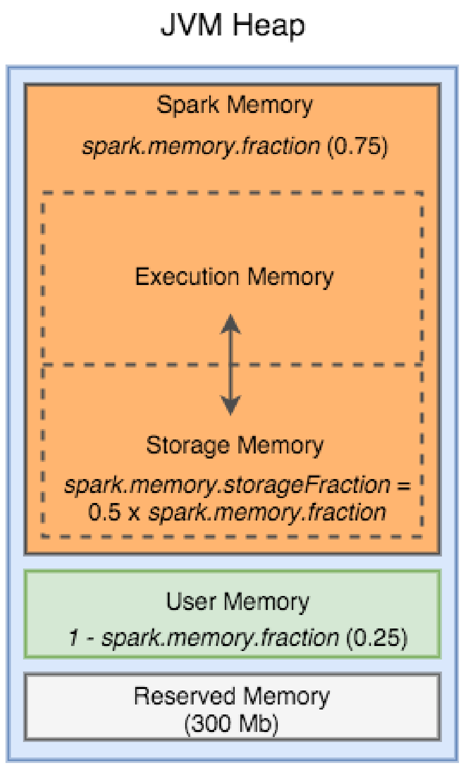

JVM Heap
Spark Memory
spark.memory.fraction (0.75)
Execution Memory
Storage Memory
spark.memory.storageFraction =
0.5 x spark.memory.fraction
User Memory
1 - spark.memory.fraction (0.25)
Reserved Memory
(300 Mb)

Figure 5.21. Memory management model of Spark 1.6.0 and above; the figure comes from the blog by Anton Kirillov at `http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/`

Although changing `spark.memory.fraction` or `spark.memory.storageFraction` is not recommended by official documentation, we still tried to explore these parameters out of interests. However, the tests did not succeed because we could neither find the size of execution memory from the webUI nor see any change in storage memory from the WebUI after changing `spark.memory. storageFraction`.

Additionally, we tested how `spark.executor.memory` would affect the running time. The mismatch between the setting and the actual memory used has already been covered in Section 5.4.2, so the value of `spark.executor.memory` and actual memory used are both listed in Table 5.8. The average running time decreased as more memory was used on each executor, though the exact contributor could not be told from the WebUI or other sources.

Table 5.8. *Running time of the Spark application when executor memory was set to different values, with 80 cores and 50 iterations and other default settings*

`spark.executor.memory` (GB)	Average memory usage on each pure slave node (GB)	Average running time (sec)
1	1.75 - 2.02	157
2	2.07 - 2.78	153
3	2.28 - 2.88	137
4	2.68 - 2.87	128

6. Related work

There has been some previous work investigating Spark on HPC clusters. For example, Tous et al. developed a framework called Spark4MN to deploy Spark on a cluster of MareNostrum petascale Supercomputer at Barcelona Supercomputing Center [Tous et al., 2015]. This project investigated optimised Spark configurations for data-intensive workloads on HPC compute-centric paradigm. Configurations in aspects of parallism, storage and network were examined. While this project and ours share interests in deploying Spark on HPC cluster IBM LSF workloader manager, we do differ in our focus areas. Their work emphasises on fine-tuning data-intensive application while we places main attention to comparison between Spark and MPI regarding performance, scalability and fault tolerance on a compute-intensive application.

Another related literature [Chaimov et al., 2016] centers around how Spark scales up in HPC environment. They found that the performance and scalability are affected by file system metadata access latency, because HPC environment usually uses global distributed file system (Lustre in that case). In this experiment, efforts have been made with regard to both hardware and software. A NVRAM Burst Buffer architecture was introduced to build a hierarchical memory to accelerate the performance of I/O. On software side, `ramdisk` and `filepool` have been examined. Three classical benchmarks *BigData Benchmark*, *PageRank* and *GroupBy* have been used to reveal how different tools and settings affect the performance on data transfer inside a node and data shuffling. With a similar test environment with ours, the main focus of this project, however, is on the evaluation of software and hardware to achieve better performance and scalability. In our thesis project, we did not notice the latency due to coarse-grained performance monitoring and absence of big data files. Nevertheless, we consider scalability optimisation using software and hardware as an area for further work.

Another work of relevance [Jha et al., 2014] involves Hadoop ecosystem. It compares two data-intensive paradigms: HPC and Apache Hadoop, and tests 6 hybrid implementation using *K-means* as benchmark. This project touches upon a large span of infrastructure, with many popular data models and resource managers involved. Although different benchmarks are used, we reached similar conclusion on the superior performance of MPI against Spark. No details of scalability, fault tolerance and resource utilisation are covered in their work.

7. Conclusions

This project aims to explore the possibility of deploying Apache Spark (a large-scale distributed computing platform) in a HPC cluster in Scania IT. We compare Spark with MPI (a traditional standard for cluster computing on HPC) in different aspects such as performance, scalability, fault tolerance, dynamic resource handing, ease of use etc.

During the five-month project period, we obtained deep understanding of HPC, MPI and Spark, and successfully deployed Spark in a HPC cluster. We implemented a compute-intensive application for both Spark and MPI. More specifically, the application solves the Bayesian inference problem of an extended Lotka-Volterra biological model using particle marginal Metropolis-Hastings algorithm. With a few test cases, we collected many insightful results, which can be summarised below

1. Performance: The application could run faster on MPI than on Spark, if the overhead did not exceeded the speedup brought by parallelism.
2. Scalability: Neither applications could scale beyond 48 cores in the initial tests with 1000 samples. As sufficient amount of computation was added, the MPI application showed linear scalability and we would expect similar properties with Spark.
3. Fault tolerance: The automatic fault tolerance feature of Spark was validated by our tests. The Spark application could keep running even after some nodes were disconnected. Unfinished tasks were distributed to other active nodes for re-execution. Whereas MPI application crashed when a node was down.
4. Dynamic resource handling: When nodes were recovered/added, the running application could make use of these nodes right away in Spark, but not in MPI without additional configuration.
5. Resource consumption: with the same setting, the MPI application made better use of the CPUs, consumed much less memory, and much less network than the Spark application.
6. Ease of use: As Spark is a platform with many components and advanced mechanism, it is easier to use compared with OpenMPI, which is a library mainly for message-passing.

The summary is based on the test results of the specific application used in the project. But we believe they definitely shed some lights on a more general comparison between Spark and MPI.

As Spark has shown good fit in the HPC cluster, our work serves as a good pre-study for the Big Data initiatives on HPC clusters at Scania IT.

7.1 Distribution of work

This project is conducted by Huijie Shen and Tingwei Huang, with work distributed evenly, including the thesis writing. The specific contributions from each person are:

Huijie Shen:

- Extending the Lotka-Volterra model and producing the observation data

- Studying and applying Monte Carlo methods to the use case, and tuning the parameters in the solution
- Studying Scala and Spark, and adapting Darren Wilkinson's implementation to a Spark application for LV-12 model
- Running test cases on the Spark application in HPC cluster

Tingwei Huang:

- Studying MPI and HPC
- Deploying Spark in a HPC cluster at Scania IT
- Implementing the application in C with OpenMPI
- Running test cases on the MPI application in the HPC cluster

7.2 Future work

There are a few directions to pursue as the next step of the project.

Firstly, we are interested in finding out how Infiniband can accelerate MPI and Spark. Infiniband is a network communication standard used in HPC. It offers very high throughput and very low latency. There is 4X FDR (56 Gb/sec) Infiniband installed on some HPC clusters at Scania IT. Although we have successfully installed RDMA for Apache Spark 0.9.1[1] on a HPC cluster with Infiniband (Infiniband is an implementation of RDMA), we did not observe any speedup for applications running on this RDMA-enabled Spark. We encountered the same problem when running the MPI application with Infiniband. So far we are not sure about the cause of it, but it would be interesting to look into this problem in the future.

Secondly, we would like to try some third-party software for monitoring MPI application. These software may provide fine-grained details of the whole execution process. If we can find such tool to monitor MPI application, it will enable us to understand resource/time consumption on each part of the execution, thereby further improve the performance.

Finally, Apache Mesos is not used in this project due to potential conflict with LSF on the HPC cluster. Mesos is well known for advanced scheduling mechanism and wide application on Big Data platforms. It would be interesting to see if Mesos can run on HPC cluster, either with LSF or as the only resource management platform, and the subsequent outcome.

[1]Developed by the Network-Based Computing Laboratory of The Ohio State University, `http://hibd.cse.ohio-state.edu/`, accessed 2016-09-10

References

[hor, 2011] (2011). Apache hadoop yarn - the architectural center of enterprise hadoop. `http://hortonworks.com/apache/yarn`. Accessed: 2016-09-19.

[tec, 2016] (2016). High-performance computing (hpc). `https://www.techopedia.com/definition/4595/high-performance-computing-hpc`. Accessed: 2016-09-10.

[Bellot, 2010] Bellot, D. (2010). log-sum-exp trick. `http://ai-owl.blogspot.se/2010/06/log-sum-exp-trick.html`. Accessed: 2016-09-10.

[Chaimov et al., 2016] Chaimov, N., Malony, A., Canon, S., Iancu, C., Ibrahim, K. Z., and Srinivasan, J. (2016). Scaling spark on hpc systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 97–110. ACM.

[Christensen et al., 2005] Christensen, O. F., Roberts, G. O., and Rosenthal, J. S. (2005). Scaling limits for the transient phase of local metropolis–hastings algorithms. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):253–268.

[Del Moral, 2004] Del Moral, P. (2004). Feynman-kac formulae. genealogical and interacting particle approximations.

[Ding, 2013] Ding, Z. (2013). Hadoop yarn. `https://www.ibm.com/.../Platform_BPG_LSF_MPI_v2.pdf`. Accessed: 2016-09-19.

[Doucet et al., 2001] Doucet, A., De Freitas, N., and Gordon, N. (2001). An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer.

[Doucet et al., 2015] Doucet, A., Pitt, M., Deligiannidis, G., and Kohn, R. (2015). Efficient implementation of markov chain monte carlo when using an unbiased likelihood estimator. *Biometrika*, page asu075.

[Dursi,] Dursi, J. Hpc is dying, and mpi is killing it. `http://www.dursi.ca/hpc-is-dying-and-mpi-is-killing-it/`. Accessed: 2016-09-10.

[Flynn, 1972] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960.

[Francisco and Erick, 2010] Francisco, Fernández, d. V. and Erick, C.-P. (2010). *Parallel and Distributed Computational Intelligence*, volume 269. Springer-Verlag Berlin Heidelberg.

[Gillespie, 1976] Gillespie, D. T. (1976). A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of computational physics*, 22(4):403–434.

[Gillespie, 1977] Gillespie, D. T. (1977). Exact stochastic simulation of coupled chemical reactions. *The journal of physical chemistry*, 81(25):2340–2361.

[Golightly and Gillespie, 2013] Golightly, A. and Gillespie, C. S. (2013). Simulation of stochastic kinetic models. *In Silico Systems Biology*, pages 169–187.

[Gropp et al., 1996] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828.

[Gropp et al., 1999] Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press.

[Gropp, 1998] Gropp, W. D. (1998). An introduction to mpi. web page. Accessed: 2016-09-15.

[Jha et al., 2014] Jha, S., Qiu, J., Luckow, A., Mantha, P., and Fox, G. C. (2014). A tale of two data-intensive paradigms: Applications, abstractions, and architectures. In *2014 IEEE International Congress on Big Data*, pages 645–652. IEEE.

[Keller et al., 2006] Keller, R., Bosilca, G., Fagg, G., Resch, M., and Dongarra, J. J. (2006). Implementation and Usage of the PERUSE-Interface in Open MPI. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Bonn, Germany. Springer-Verlag.

[Lotka, 1925] Lotka, A. J. (1925). Elements of physical biology.

[MacKenzie, 1998] MacKenzie, D. A. (1998). *Knowing machines: Essays on technical change*. MIT Press.

[Neal et al., 2006] Neal, P., Roberts, G., et al. (2006). Optimal scaling for partially updating mcmc algorithms. *The Annals of Applied Probability*, 16(2):475–515.

[Pitt et al., 2010] Pitt, M., Silva, R., Giordani, P., and Kohn, R. (2010). Auxiliary particle filtering within adaptive metropolis-hastings sampling. *arXiv preprint arXiv:1006.1914*.

[Pitt et al., 2012] Pitt, M. K., dos Santos Silva, R., Giordani, P., and Kohn, R. (2012). On some properties of markov chain monte carlo simulation methods based on the particle filter. *Journal of Econometrics*, 171(2):134–151.

[Prodan and Fahringer, 2007] Prodan, R. and Fahringer, T. (2007). Experiment management, tool integration, and scientific workflows. *Grid computing*, 4340 2007:1–4.

[Ristic and Skvortsov, 2015] Ristic, B. and Skvortsov, A. (2015). Chapter 25 - predicting extinction of biological systems with competition. In Tran, Q. N. and Arabnia, H., editors, *Emerging Trends in Computational Biology, Bioinformatics, and Systems Biology*, pages 455 – 466. Morgan Kaufmann.

[Roberts et al., 1997] Roberts, G. O., Gelman, A., Gilks, W. R., et al. (1997). Weak convergence and optimal scaling of random walk metropolis algorithms. *The annals of applied probability*, 7(1):110–120.

[Roberts et al., 2001] Roberts, G. O., Rosenthal, J. S., et al. (2001). Optimal scaling for various metropolis-hastings algorithms. *Statistical science*, 16(4):351–367.

[Sherlock et al., 2015] Sherlock, C., Thiery, A. H., Roberts, G. O., Rosenthal, J. S., et al. (2015). On the efficiency of pseudo-marginal random walk metropolis algorithms. *The Annals of Statistics*, 43(1):238–275.

[Squyres,] Squyres, J. M. Open mpi. `http://aosabook.org/en/openmpi.html`. Accessed: 2016-09-10.

[Tous et al., 2015] Tous, R., Gounaris, A., Tripiana, C., Torres, J., Girona, S., Ayguadé, E., Labarta, J., Becerra, Y., Carrera, D., and Valero, M. (2015). Spark deployment and performance evaluation on the marenostrum supercomputer. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 299–306. IEEE.

[Ullah and Wolkenhauer, 2011] Ullah, M. and Wolkenhauer, O. (2011). *Stochastic approaches for systems biology*. Springer Science & Business Media.

[Volterra, 1926] Volterra, V. (1926). Fluctuations in the abundance of a species considered mathematically. *Nature*, 118:558–560.

[Wilkinson, 2011a] Wilkinson, D. (2011a). Bayesian parameter and state estimation for partially observed nonlinear markov process models using particle mcmc (with application to sysbio).

[Wilkinson, 2011b] Wilkinson, D. (2011b). *Stochastic modelling for systems biology*. CRC press.

[Wilkinson, 2014] Wilkinson, D. (2014). Tuning particle mcmc algorithms. `https://darrenjw.wordpress.com/2014/06/08/tuning-particle-mcmc-algorithms/`. Accessed: 2016-09-10.

[Wu, 1999] Wu, X. (1999). *Performance Evaluation, Prediction and Visualization of Parallel Systems*. Springer Science & Business Media.

[Zaharia et al., 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association.

[Zaharia et al., 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. *HotCloud*, 10:10–10.