Università degli Studi di Torino

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Informatica

Tesi di Laurea Magistrale

Parallel stochastic simulators in systems biology: the evolution of the species

Relatore:
Marco Aldinucci

Candidato:
Maurizio Drocco

Anno Accademico 2012-2013

The stochastic simulation of biological systems is an increasingly popular technique in bioinformatics. It is often an enlightening technique, especially for multi-stable systems whose dynamics can be hardly captured with ordinary differential equations. To be effective, stochastic simulations should be supported by powerful statistical analysis tools. The simulation/analysis workflow may however result in being computationally expensive, thus compromising the interactivity required especially in model tuning. In this work we discuss the main opportunities to speed up the framework by parallelisation on modern multicore and hybrid multicore and distributed platforms, advocating the high-level design of simulators for stochastic systems as a vehicle for building efficient and portable parallel simulators endowed with on-line statistical analysis. In particular, the Calculus of Wrapped Compartments (CWC) Simulator, which is designed according to the FastFlow's pattern-based approach, is presented and discussed in this work.

Preface

This work reports results on stochastic simulation for systems biology matured during my 3-year activity as research engineer at Department of Computer Science of University of Torino within Work Package 3 ("In Silico Design and Analysis of Biological Systems") of the BioBITs project. In this activity I have collaborated with number of researchers involved in the same project, they are:

- M. Aldinucci, M. Coppo, F. Damiani, E. Sciacca, S. Spinella, A. Troina from University of Torino, Department of Computer Science
- M. Torquati from University of Pisa, Department of Computer Science
- C. Calcagno from University of Torino, Department of Life Sciences and Systems Biology

Their contribute to the whole work is analitically reported in the next section. The presented material appeared in the following scientific publications: [50, 8, 7, 18, 3, 20].

Contributions

CC, MC, FD, and AT formalised the CWC calculus. MA and MD designed and implemented the stochastic simulation procedures. MD implemented all versions of the simulation-analysis pipeline, the parallel simulation and the parallel analysis stages. MA and MT designed and implemented the FastFlow programming framework. CC, ES and SS collected the models from the literature. MA, MD, ES and SS implemented the algorithms needed in the analysis of results, performed simulations and analysed the results. MA and MC supervised the activity.

Funding

This work has been supported by the BioBITs project (Converging technologies 2007, area: Biotechnology-ICT) - Regione Piemonte - Italy, and by Paraphrase project (FWP7 EC-STREP n. 288570).

Contents

List of Figures

List of Tables

1 Introduction

The stochastic simulation of biological systems is an increasingly popular technique in bioinformatics, as either an alternative or a complementary tool to traditional differential equations (ODEs) solvers. This trend, starting from Gillespie's seminal work [68], has been supported by a growing number of formalisms aiming to describe biological systems as stochastic models [21].

Unfortunately, the stochastic modelling and simulation approach is computationally more expensive than ODEs. Nevertheless, it is still considered attractive for its superior ability to describe transient and multi-stable behaviours of biological systems: rare or divergent trends, spikes and families of hypothetical conditions that are typically hidden in the averaged process described by ODEs.

The high computational cost of stochastic simulations is well known and has led, in the last two decades, to a number of attempts to accelerate them up by using several kinds of techniques, such as approximate simulation algorithms and parallel computing. In this work, this latter approach is taken into account.

1.1 Goals

Since stochastic simulations rely basically on Monte Carlo methods, many independent instances should be computed to achieve statistically meaningful results. These independent instances have been traditionally exploited in an *embarrassingly parallel* fashion, executing a partition of the instances on different platforms over large infrastructures (e.g. grids, clouds). This approach suffers from some drawbacks related to design, performance and usability of simulation tools. A relevant issue is slow time-to-solution as the single experiment (which is typically repeated several times to fine-tune stochastic parameters) requires to manage complex interactions with shared environments (enqueue jobs, deploy initial data, gather results etc.) and post-processing results.

In the meanwhile, the entire hardware industry has moved to multicore, which nowadays equips the large majority of computing platforms, included ones constituting large computing clusters. These platforms, which are increasingly diffused in scientific laboratories, typically offer moderate to high peak computational power. This potential power, however, cannot always be turned into actual application speedup. This flaw becomes worth of particular attention for I/O and memory-bound applications, since all the cores usually share the same memory and I/O subsystem.

Indeed, the simulation of biological systems produces a large amount of data, which can be regarded as streams of data resulting from the on-going simulations. The management of these streams is not trivial on multicore platforms, as the memory bandwidth cannot usually sustain a continuous flux of data coming form all the cores at the same time.

A related aspect concerns the filtering and the analysis of raw results, which require the merging of data obtained from different simulation instances – and possibly their statistical description or mining – with data reduction techniques. In distributed computing, this phase is often demoted to a secondary aspect in the computation and treated with off-line post-processing tools, frequently not even disclosed in performance results.

The outlined approach is no longer practical because of a number of reasons:

1. the ever-increasing size of produced data burdens on the main weaknesses of multicore platforms, i.e. memory bandwidth and core synchronisations;
2. the "sequencing" of simulation and analysis phases slow down the design-to-result process, which is particularly annoying during the tuning of the biological model;
3. the design of the tools is often specifically optimised for a specific parallel platform, either multicore or distributed (or not optimised at all).

Since the frequency and the size of data moved across the simulation/analysis process strictly depend by the required accuracy, the simulation and analysis of biological systems at high-precision happens to be a serious issue on modern shared-memory multicore (and multicore cluster) platforms.

This work presents a critical rethinking of the parallelisation of the whole simulation/analysis workflow[1] in the light of hybrid multicore and distributed platforms. In this vision, data analysis is managed as an on-line process working on streams of data resulting from the on-going simulations. This approach has non-trivial effects on tool design since both the parallel simulator and the parallel analysis should work on (high-frequency) streams, and requires efficient data dependencies management (both on distributed and shared-memory systems). While the execution of multiple Monte Carlo simulation instances "in insulation" is an embarrassingly parallel process, the whole simulation-analysis workflow definitively is not.

We also focus on the tools that are required to derive an efficient simulator from easy engineering viewpoints. This latter aspect will be of crucial importance for the next generation of biological tools, that will be prevalently used by bioinformatics scientists, who are likely to be more interested in the accurate modelling of natural phenomena rather than on the synchronisation protocols required to build efficient tools on both multicore platforms and large distributed execution environments.

1.2 Approach

In this work we investigate parallelism exploitation on tools for Monte Carlo simulation (and analysis) in several ways:

1. parallelism within a single simulation instance
2. parallelism among (independent) simulation instances

[1] in this work we deliberately use the term *workflow* – instead of the more common *pipeline* – in order to abstract its meaning from any implementation pattern.

3. parallelism between simulation and analysis stages (i.e. pipeline)

4. parallelism among analysis engines

The goal of point 1 is the parallelisation of the single step of a simulation instance. This issue has been approached using Single Instruction Multiple Data (SIMD) hardware accelerators.

The goal of point 2 is the parallel execution of independent simulation instances with dynamic load balancing, which tends to align instances to the simulation time, thus outputting aligned *data streams* to the analysis module and minimizing alignment buffering. The data structure on which the alignment is realized is called *selective memory*. Technically, a low-level scheduler has been designed which dynamically remaps n simulation instances over $k < n$ simulation workers at every time quantum. The scheduler prioritizes "slow" simulation tasks, in order to minimize the gap (naturally induced by random-walk effect) between execution and simulation times for all the simulation instances.

The goal of point 3 is the concurrent execution of simulation and analysis modules, i.e. the online statistical filtering of sample streams coming from multiple simulation instances. Exploited techniques include pipelines and fixed-size sampling windows which, together with selective memory, allow the realization of memory-efficient filters based on *partial* sample streams. This latter technique is exact for instantaneous estimators (e.g. sample mean and variance) while it exhibits an arbitrary approximation degree (depending on the size of the sampling windows) for the engines that work on the whole data set (e.g. clustering).

The goal of point 4 is the parallel execution of statistical analysis engines. It is actually an embarrassingly parallel schema, except from those engines which use results at sampling time t_n to initialize the computation at time t_{n+1} (e.g. clustering).

In order to asses the effectiveness of high-level parallel programming in supporting efficiency and performance portability on heterogeneous parallel computing platforms, the presented guidelines have been actually implemented on top of FastFlow [19], a high-level stream programming framework targeting hybrid multicore and distributed platforms.

1.3 Results

A simulator tool for the Calculus of Wrapped Compartments (CWC for short) [50] has been used as test-bed. CWC is a recently proposed formalism, based on term rewriting, for the representation of biological systems. The tool has been designed exploiting a high-level methodology based on parallel patterns on top of the FastFlow programming framework. The very same design has been used for several variants targeting different platforms, namely multicore, multicore cluster, public cloud and federated public cloud. This easiness of portability has been made possible by the high-level approach, enabled by the layered FastFlow architecture.

The tool has been implemented in FastFlow/C++ . In the multicore variant, threads execute in a lock-less fashion (non-blocking multithreading). The distributed variant ex-

[50]	CWC calculus
[8]	1, 2 (multicore)
[7]	3, 4 (multicore)
[18]	2, 3 (distributed)
[3]	2, 3, 4 (multicore)
[20]	2, 3, 4 (cloud)
[120]	CWC Simulator tool

Table 1.1: Publications

ploits zero-copy communications provided by the ZeroMQ library and works on TCP/IP networks. Supported operating systems include GNU/Linux, Mac OS X, Microsoft Windows XP and Windows 7 with different compilers (gcc, icc, Microsoft Visual Studio). The tool includes a Bison parser for CWC models and a Java graphical user interface which displays the output of the statistical analysis.

In order to assess the validity of the approach, the simulation/analysis workflow provided by the tool has been validated for performance and effectiveness in capturing biological systems behaviours. Experiments have been conducted, over representative proof-of-concept biological systems, on multicore, multicore cluster and cloud platforms.

1.4 Publications

Some of the contents of this work have been already published during the development process. Table 1.1 summarizes the produced publications, with respect to the points outlined in section 1.2.

The CWC Simulator tool has been released as open source software under LGPL licence [120].

1.5 Summary

In Chapter 2, we provide some background material about parallel computing (Sec. 2.1) and systems biology (Sec. 2.2). For the former topic we briefly review parallel computing platforms (2.1.1) and high-level parallel programming frameworks (2.1.2, 2.1.3), in particular FastFlow (2.1.4) that we use in this work. For the latter topic we introduce the formal modelling (2.2.1) and stochastic simulation (2.2.2) of biological systems (compared to ODE), reviewing existing modelling languages (2.2.3), simulation tools for systems biology (2.2.4) and cloud-based tools for bioinformatics (2.2.5).

In Chapter 3, we present CWC, the formal calculus for the stochastic modelling of biological systems we use in this work. First, we define the syntax and the qualitative

aspects of CWC (Sec. 3.1), also providing some modelling guidelines (3.1.4); then we present a stochastic quantitative model for CWC (Sec. 3.2).

In Chapter 4, we outline the simulation/analysis workflow, which optimisation is the main topic of this work. First, we discuss on what features a tool should exhibit in order to support biologists when studying a biological system by way of stochastic simulation (Sec. 4.1); then we review the most common optimisation approaches, highlighting their main drawbacks (Sec. 4.2).

In Chapter 5, we use the CWC calculus and its (sequential) simulator core (5.2.1) as paradigmatic example to discuss the key features required to overcome the issues outlined in 4.2 and derive an easy porting on multicore platforms (Sec. 5.1). In particular we will argument on the technical challenges required by the parallelisation of a single simulation instance and many independent instances (5.1.1). The key features discussed are turned into a family of solutions. The former issue is approached using SIMD hardware accelerators (5.2.2), the latter advocating a novel streamed simulation schema based on FastFlow accelerator that guarantees both easy development and efficient execution (5.2.3).

In Chapter 6 we focus on the analysis stage of the framework. We introduce the problem of rich on-line analysis of simulation data and the issues posed by dealing with large (even big) data streams (Sec. 6.1). We discuss the realisation of an efficient parallel analysis stage, featuring multiple statistical and mining methods on streamed simulation outcomes (6.2.2). Finally we put all together, building up the whole CWC Simulator workflow for multicore platforms (Sec. 6.2).

In Chapter 7 we show how our approach fits on increasingly popular distributed platforms for High Performance Computing (HPC) bioinformatics, namely multicore clusters and public IaaS clouds (7.1). Finally we present the distributed CWC Simulator workflow (7.2), a very low-effort extension targeting hybrid multicore and distributed platforms.

In Chapter 8 we evaluate experimentally our approach along two main lines: the effectiveness of the proposed on-line analysis to capture the behaviour of the simulated systems (Sec. 8.1), and the efficiency and speedup of the tool in executing the simulation/analysis workflow on multicore (8.2.2), hybrid multicore cluster (8.2.3) and cloud (8.2.4) platforms.

In Chapter 9 we give some concluding remarks.

In Appendix A we discuss some aspects related to statistical correctness of the implemented tools; in particular we focus on the correct use of PRNGs, in order to guarantee the statistical independence of the simulation instances.

2 Background

In this chapter, we provide some background material.

In Sec. 2.1 we review the most common parallel computing platforms (2.1.1); then we introduce the problem of effective programming of such platforms exploiting high-level skeleton-based approaches on multicore (2.1.2) and multicore cluster (2.1.3) platforms; in particular we present FastFlow (2.1.4), that we use in this work.

In Sec. 2.2 we give a brief introduction to systems biology, modelling of biological systems (2.2.1) and their stochastic simulation (2.2.2), especially by means of the Gillespie SSA algorithm; we review existing modelling languages (2.2.3), simulation tools for systems biology (2.2.4) and cloud-based tools for bioinformatics (2.2.5).

2.1 Parallel Computing

Computing hardware has evolved to sustain an insatiable demand for high-end performances along two basic ways. On the one hand, the increase of clock frequency and the exploitation of instruction-level parallelism boosted the computing power of the single processor. On the other hand, many processors have been arranged in multi-processors, multi-computers, and networks of geographically distributed machines.

Nowadays, after years of continual improvement of single core chips trying to increase instruction-level parallelism, hardware manufacturers realised that the effort required for further improvements is no longer worth the benefits eventually achieved. Microprocessor vendors have shifted their attention to thread-level parallelism by designing chips with multiple internal cores, known as multicores (or chip multiprocessors).

More generally, it is a matter of fact that *parallelism* at multiple levels is now the driving force of computer design across all classes of computers, from small desktop workstations to large warehouse-scale computers.

2.1.1 Platforms

We briefly recap the review of existing parallel computing platforms from [73].

Following the taxonomy in [66], we can define two main classes of architectures supporting parallel computing:

- Single Instruction stream, Multiple Data streams (SIMD): the same instruction is executed by multiple processors using different data streams. SIMD computers support *data-level parallelism* by applying the same operations to multiple items of data in parallel.

- Multiple Instruction streams, Multiple Data streams (MIMD): each processor fetches its own instructions and operates on its own data, and it targets task-level parallelism. In general, MIMD is more flexible than SIMD and thus more generally applicable, but it is inherently more expensive than SIMD (see 2.1.1).

  We can further subdivide MIMD into two subclasses:

  - tightly coupled MIMD architectures, which exploit thread-level parallelism since multiple cooperating threads operate in parallel on the same execution context;
  - loosely coupled MIMD architectures, which exploit request-level parallelism, where many independent tasks can proceed in parallel "naturally" with little need for communication or synchronization.

Although it is a very common classification, this model is becoming more and more coarse, as many processors are nowadays "hybrids" of the classes above (e.g. GPGPU).

SIMD computers

One of the simplest examples of SIMD computers are Intel Streaming SIMD Extensions (SSE) [75] of the x86 architecture. Processors implementing SSE (with a dedicated unit) can perform simultaneous operations on multiple operands in a single register. For example, SSE instructions can simultaneously perform eight 16-bit operations on 128-bit registers. Advantages of such approach is almost negligible overhead and little hardware cost, while they are very hard to integrate into existing code, which actually amounts to writing in assembly language.

An increasingly popular platform specifically targeting data-level parallelism consists in the use of GPU instructions for general-purpose computing, and it is known as General-Purpose computing on Graphics Processing Units (GPGPU). For a limited cost, anyone can buy a GPU with hundreds of parallel floating-point units, which makes high computational power more accessible. Moreover, using specific programming languages and frameworks (e.g. NVIDIA CUDA [94], OpenCL [78]) partially reduces the gap between high computational power and easiness of programming, though it still remains a low-level parallel programming approach, since the user has to deal with very close-to-metal aspects like memory allocation and data movement between the GPU and the host platform.

GPGPU is not strictly a SIMD architecture. For example, NVIDIA CUDA-enabled GPUs are based on multithreading, thus they support all types of parallelism; however, control hardware in these platforms is very limited (e.g. no global thread synchronisation), making GPGPU more suitable for data-level parallelism.

Symmetric shared-memory multiprocessors

Thread-level parallelism implies the existence of multiple program counters, hence is exploited primarily through MIMDs. We remark that threads can also be used to support data-level parallelism, but some overhead is introduced at least by thread communication

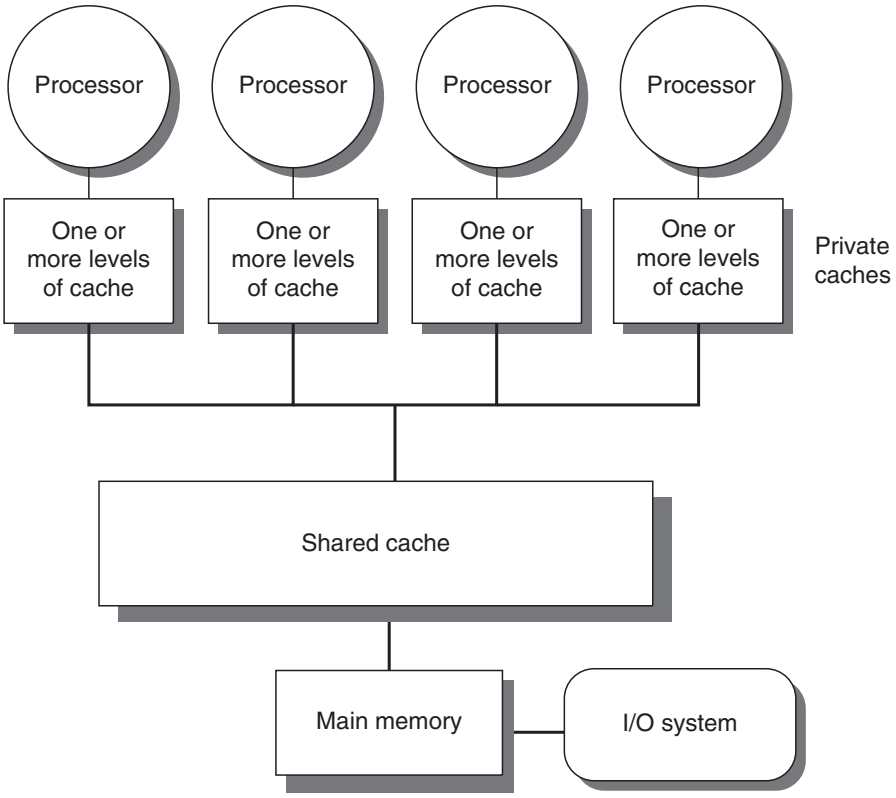


Figure 2.1: Structure of a SMP

and synchronisation. This overhead means the *grain size* (i.e. the ratio of computation to the amount of communication) is a key factor for efficient exploitation of thread-level parallelism.

The most common MIMD computers are multiprocessors, defined as computers consisting of *tightly coupled* processors that share memory through a shared address space. Single-chip systems with multiple cores are known as *multicores*. *Symmetric* (shared-memory) multiprocessors (SMPs) typically feature small numbers of cores (nowadays eight or fewer), thus it is possible for the processor to share a single *centralized* memory that all processors have equal access to (Fig. 2.1). In multicore chips, the memory is effectively centralized, and all existing multicores are SMPs. SMP architectures are also sometimes called uniform memory access (UMA) multiprocessors, arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks.

The alternative design approach consists of multiprocessors with physically distributed memory, called distributed shared memory (DSM). To support larger processor counts, memory must be distributed rather than centralized; otherwise, the memory system would not be able to support the bandwidth demands of processors without incurring in excessively long access latency. Such architectures are known as nonuniform memory

access (NUMA), since the access time depends on the location of a data word in memory.

Cache Coherence and False Sharing SMP machines usually support the caching of both shared and private data, reducing the average access time as well as the memory bandwidth required. Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which could end up seeing two different values. This difficulty is generally referred to as the *cache coherence problem* and several protocols have been designed to guarantee cache coherence. In cache-coherent SMP machines, *false sharing* is a subtle source of cache miss, which arises from the use of an invalidation-based coherence algorithm. False sharing occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into. In a false sharing miss, the block is shared, but no word in the cache is actually shared, and the miss would not occur if the block size were a single word. It is a matter of fact that false sharing is one of the most diffused (and underestimated) sources of performance flaws of parallel computing on cache-coherent SMP platforms.

Clusters and Clouds

In contrast with shared-memory architectures, *clusters* look like individual computers connected by a network. Since each processor has its own address space, the memory of one processor cannot be accessed by another processor without the assistance of software protocols running on both processors. In such design, message-passing protocols are used to communicate data among processors. Clusters are examples of loosely coupled MIMDs. These large-scale systems are typically used for *cloud computing* with a model that assumes either massive numbers of independent requests or highly parallel, intensive compute tasks.

In this work we focus on two classes of large-scale distributed systems:

- *Private* clouds, in particular multicore clusters, which are inter networked – possibly heterogeneous – multicore devices.
- *Public* clouds, which are (physical or virtual) infrastructures offered by providers in the form of inter networked clusters. In the most basic public cloud model, providers of IaaS (Infrastructures-as-a-Service) offer computers – physical or (more often) virtual machines – and other resources on-demand. Public IaaS clouds can be regarded as *virtual* multicore clusters.

The (public) cloud model encompasses a *pay-per-use* business model. End users are not required to take care of hardware, power consumption, reliability, robustness, security, and the problems related to the deployment of a physical computing infrastructure. In IaaS cloud usage, the aggregate computing power and storage space are provided to user applications in an elastic fashion. In principle, they can be scaled up and down according to user needs, and billed accordingly. The pay-per-use business model is typically applied in term of the Quality of Service (e.g. performance, latency, storage space) provided by the service.

2.1.2 Programming multicores

Shifting from sequential to parallel computing, a trend largely motivated by the advent of multicore platforms, does not always translate into greater CPU performance: multicores are small-scale but full-fledged parallel machines and they retain many of their usage problems. In particular, sequential code will get no performance benefits from them. A workstation equipped with a quad-core CPU but running sequential code is wasting 3/4 of its computational power. Developers are then facing the challenge of achieving a trade-off between performance and human productivity (total cost and time to solution) in developing and porting applications to multicore and parallel platforms in general.

Therefore effective parallel programming happens to be a key factor for efficient parallel computing, as we shall see in the following discussion.

Types of parallelism

Traditionally, types of parallelisms are categorised in three main classes:

- *Task Parallelism.* Parallelism is explicit in the algorithm and consists of running the same or different code on different executors (cores, processors, etc.). Different flows-of-control (threads, processes, etc.) communicate with one another as they work. Communication takes place usually to pass data from one thread to the next as part of a graph.

- *Data Parallelism* is a method for parallelising a single task by processing independent data elements of this task in parallel. The flexibility of the technique relies upon stateless processing routines implying that the data elements must be fully independent. Data Parallelism also supports *Loop-level Parallelism* where successive iterations of a loop working on independent or read-only data are parallelised in different flows-of-control (according to the model *co-begin/co-end*) and concurrently executed.

- *Stream Parallelism* is a method for parallelising the execution (a.k.a. filtering) of a stream of tasks by segmenting each task into a series of *sequential*[1] or *parallel* stages. This method can be also applied when there exists a *total* or *partial* order, respectively, in a computation preventing the use of data or task parallelism. This might also come from the successive availability of input data along time (e.g. data flowing from a device). By processing data elements in order, local state may be either maintained in each stage or distributed (replicated, scattered, etc.) along streams. Parallelism is achieved by running each stage simultaneously on *subsequent* or *independent* data elements.

In this work we are interested mainly in stream parallelism, since it underlies a large spectrum of parallel applications. In particular, stochastic simulation tools produce outcomes that can be naturally regarded as data streams.

[1] In the case of total sequential stages, the method is also known as *Pipeline Parallelism.*

Low-level approach

Typically, low-level approaches provide the programmers only with primitives for flows-of-control management (creation, destruction), their synchronisation and data sharing, which are usually accomplished in critical regions accessed in mutual exclusion (mutex). As an example, POSIX thread library can be used to this purpose. Programming parallel complex applications is this way is certainly hard; tuning them for performance is often even harder due to the non-trivial effects induced by memory fences (used to implement mutex) on data replicated in core's caches.

Indeed, memory fences are one of the key sources of performance degradation in communication intensive (e.g. streaming) parallel applications. Avoiding memory fences means not only avoiding locks but also avoiding any kind of atomic operation in memory (e.g. Compare-And-Swap, Fetch-and-Add). While there exists several assessed fence-free solutions for *asynchronous symmetric* communications[2], these results cannot be easily extended to *asynchronous asymmetric* communications[3], which are necessary to support arbitrary streaming networks. This ability is one of the core features of FastFlow (2.1.4).

A first way to ease programmer's task and improve program efficiency consists in raising the level of abstraction of concurrency management primitives. As an example, threads might be abstracted out in higher-level entities that can be pooled and scheduled in user space possibly according to specific strategies to minimise cache flushing or maximise load balancing of cores. Synchronisation primitives can be also abstracted out and associated to semantically meaningful points of the code, such as function calls and returns, loops, etc. Intel *Threading Building Block* (TBB) [76], *OpenMP* [97], and *Cilk* [46] all provide this kind of abstraction (even if each of them in its own way).

This kind of abstraction significantly simplifies the hand-coding of applications but it is still too low-level to effectively (semi-)automatise the optimisation of the parallel code. Moreover, the above-mentioned programming framework for multicore architectures are not specifically designed to support streaming applications.

Skeleton-based approach

In order to reduce complexity and simultaneously providing the user with a higher level of abstraction, algorithmic *skeletons* have been proposed [47]. They provide predefined parallel computation and communication patterns, hiding the parallelism management to the user.

Some basic forms of parallelism above have been encoded in high-level patterns (a.k.a. *skeletons*), that capture common parallel programming paradigms (e.g. MapReduce, ForAll, Divide & Conquer, etc.) and make them available to the programmer as high-level programming constructs equipped with well-defined functional and extra-functional semantics [11]. Many skeletons appeared in literature in the last two decades covering many different usage schema of the three classes of parallelism, on top of both the message passing [48, 55, 116, 5, 16, 34, 102, 12, 56, 9] and shared memory [1, 76] programming

[2]Single-Producer-Single-Consumer (SPSC) queues [81].
[3]Multiple-Producer-Multiple-Consumer queues (MPMC).

models.

The algorithmic skeleton community has proposed various programming frameworks, aimed at providing the application programmer with very high-level abstractions completely encapsulating parallelism exploitation patterns and solving most of the problems mentioned above [47, 69]. Initial skeleton-based programming frameworks targeted only cluster/network of workstations. More recently, some of the existing frameworks have been extended in such a way that (clusters of) multicores may also be exploited.

Skeletons for stream parallelism

A stream-parallel program can be naturally represented as a graph of independent *stages* (kernels or filters) which communicate over data channels. Conceptually, a streaming computation represents a sequence of transformations on the data streams in the program. Each stage of the graph reads one or more tasks from the input stream, applies some computation, and writes one or more output tasks to the output stream. Parallelism is achieved by running each stage of the graph simultaneously on *subsequent* or *independent* data elements.

Several skeletons exists that support stream-parallel programming.

The *pipeline* skeleton is one of the most widely-known, although sometimes it is underestimated. Parallelism is achieved by running each stage simultaneously on subsequent data elements, with the pipeline's throughput being limited by the throughput of the slowest stage.

The *farm* skeleton models functional replication and consists in running multiple independent stages in parallel, each operating on different tasks of the input stream. The farm skeleton is typically used to improve the throughput of slow stages of a pipeline. It can be better understood as a three stage – *emitter*, *workers*, *collector* – pipeline. The emitter dispatches stream items to a set of workers, which independently elaborate different items. The output of the workers is then gathered by the collector into a single stream. These logical stages are considered by a consolidated literature as the basic building blocks of stream programming.

The *loop* skeleton (also known as *feedback*), provides a way to generate cycles in a stream graph. This skeleton is typically used together with the farm skeleton to model recursive and Divide&Conquer computations.

Several languages and libraries are available for programming stream applications. Some of them, e.g. CUDA and OpenCL, are very close to the metal and cannot be considered as high-level languages. Some others provide programmers with a higher level of abstraction, but are oriented to coarse grain computations (e.g. ASSIST [126, 4], StreamIt [124], Brook [38]). Not many succeed to target streams, even at medium or fine grain with a high-level programming model, *inter-alia* Intel Threading Building Blocks (TBB) [76] and FastFlow (2.1.4).

2.1.3 Programming multicore clusters

Programming tools and frameworks are needed for efficiently target the architectures hosting inter networked – possibly heterogeneous – multicore devices, which appear to be *the* reference architecture ferrying programmers from the mainly sequential to mainly parallel programming era [27]. The urgency is even more crucial given that both grids and clouds provide application programmers with the possibility to reserve collections of multicores to support parallel applications eventually presented as (or orchestrated by) web services.

Shared-memory multicores and clusters/networks of (virtualised or not) processing elements require quite different techniques and tools to support efficient parallelism exploitation. The *de facto* standard tools in the two cases are OpenMP [97] and MPI [96], used either alone or in conjunction. Despite being very efficient on some classes of applications, OpenMP and MPI share a common set of problems: poor separation of concerns among application and system aspects, a rather low level of abstraction presented to the application programmers and poor support for really fine grained applications. These are all considerations that hinder the ease of use of MPI and OpenMP.

At the moment it is not clear if the mixed MPI/OpenMP programming model always offers the most effective mechanisms for programming clusters of SMP systems [39]. Furthermore, when directly using communication libraries such as MPI, the abstraction level is rather low. The programmer has to think about decomposing the problem, integrating the partial solutions, and bother with communication problems such as deadlocks and starvation.

Therefore we advocate the use of high-level parallel programming frameworks targeting hybrid multicore and distributed platforms as a vehicle for building efficient parallel software – possibly derived semi-automatically from sequential code – featuring performance portability over heterogeneous parallel platforms.

2.1.4 FastFlow

FastFlow [19] is a structured skeleton-based parallel programming framework originally designed for cache-coherent shared-memory multicore platforms. It has been recently extended to support distributed systems and cloud [6].

The key vision of FastFlow is that ease-of-development and runtime efficiency can both be achieved by raising the abstraction level of the design phase, thus providing developers with a set of parallel programming patterns (skeletons). They include stream-oriented patterns (farm, farm-with-feedback and pipeline) and data-parallel patterns (map, reduce). The predefined patterns provided by FastFlow may be customised in different ways. For example, default patterns may be arbitrarily nested in order to get pipelines with farm stages and vice-versa. Using the customisation features, different higher-level patterns may be implemented, such as *Divide&Conquer*, *map* and *reduce* patterns [15].

From the performance viewpoint, one distinguishing feature at the core of FastFlow is that it supports lock-free (fence-free) Multiple-Producer-Multiple-Consumer (MPMC)

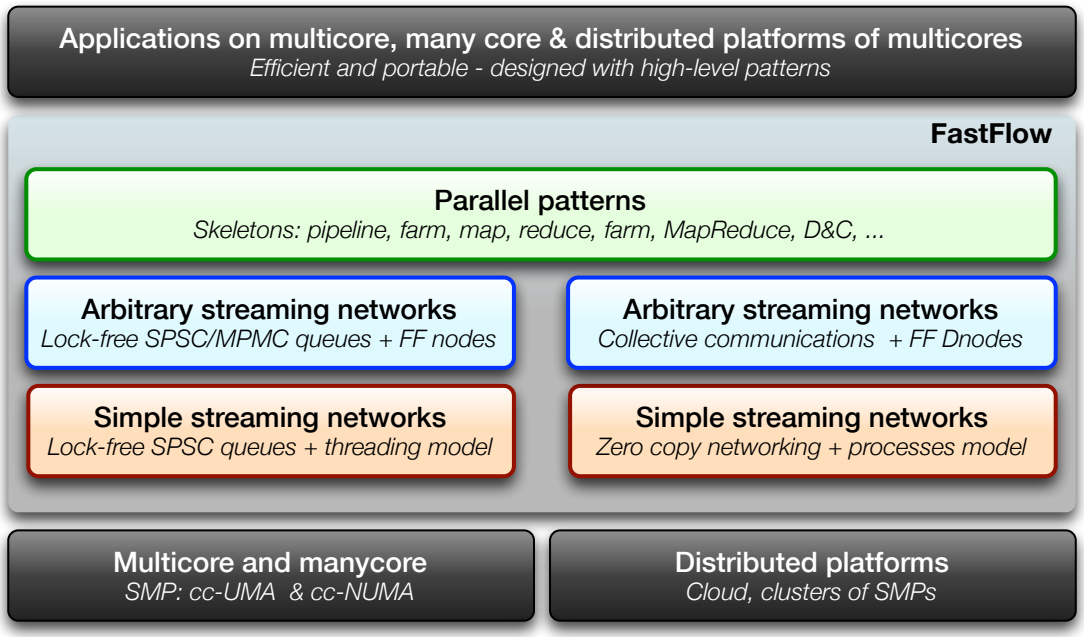


Figure 2.2: Layered FastFlow design

queues [17] able to support low-overhead high-bandwidth multi-party communications on multicore architectures, i.e. any *streaming network*, including cyclic graphs of threads. The key intuition underneath FastFlow is to provide the programmer with lock-free MP queues and MC queues (that can be used in pipeline to build MPMC queues) to support fast streaming networks. Traditionally, MPMC queues are built as passive entities: threads concurrently synchronise (according to some protocol) to access data; these synchronisations are usually supported by one or more atomic operations (e.g. Compare-And-Swap) that behave as memory fences. FastFlow design follows a different approach: in order to avoid any memory fence, the synchronisations among queue readers or writers are arbitrated by an active entity (e.g. a thread). We call these entities *Emitter* (E) or *Collector* (C) according to their role; they actually read an item from one or more lock-free SPSC queues and write onto one or more lock-free SPSC queues. This requires a memory copy but no atomic operations. The performance advantage of this solution descends from the higher speed of the copy with respect to the memory fence, that advantage is further increased by avoiding cache invalidation triggered by fences. This also depends on the size and the memory layout of copied data. The former point is addressed using data pointers instead of data, and enforcing that the data is not concurrently written: in many cases this can be derived by the semantics of the skeleton that has been implemented using MPMC queues (as an example this is guaranteed in a stateless farm and many other cases).

FastFlow design is layered (see Fig. 2.2). The lower layer, called *simple streaming*

```
1 class ff_node {
2 protected:
3     virtual bool push(void* data) { return qout->push(data);}
4     virtual bool pop(void** data) { return qin->pop(data);}
5 public:
6     virtual void* svc(void * task) = 0;
7     virtual int    svc_init () { return 0; };
8     virtual void  svc_end()   {}
9      ...
10 private:
11     SPSC* qin;
12     SPSC* qout;
13 };
```

Figure 2.3: FastFlow's `ff_node` class schema

networks, basically provides two basic abstractions:

- *process-component*, i.e. a control flow realised with POSIX threads and processes, for multicore and distributed platforms respectively.
- *1-1 channel*, i.e. a communication channel between two components, realised with wait-free single-producer/single-consumer queues (FF-SPSC) [14] and zero-copy *ZeroMQ* channels [127] for multicore and distributed platforms, respectively.

Both realisations of the 1-1 channel are the top of the state-of-the-art in their classes, in terms of latency and bandwidth. As an example, FF-SPSC exhibits a latency down to 10 ns per message on a standard Intel Xeon @2.0GHz [14]. ZeroMQ is an LGPL open-source communication library [127]. It provides the user with a socket layer that carries whole messages across various transports: inter-thread communications, inter-process communications, TCP/IP and multicast sockets. ZeroMQ offers an asynchronous communication model, which provides a quick construction of complex asynchronous message-passing networks, with reasonable performance. The message API offers the possibility to perform zero-copy sends and and non-blocking calls to the socket layer.

Above this mechanism, the second layer – called *arbitrary streaming networks* – further generalizes the two concepts, providing:

- *FastFlow node*, i.e. the basic unit of parallelism that is typically identified with a node in a streaming network. It is used to encapsulate sequential portions of code implementing functions, as well as high-level parallel patterns such as pipelines and farms. From the implementation viewpoint, the `ff_node` C++ class realizes a node in the shared-memory scenario and the `ff_dnode` extends it in the distributed memory setting. (see Fig. 2.3).
- *collective channel*, i.e. a collective communication channel, either among `ff_node`s or many `ff_dnode`s.

unicast	unidirectional point-to-point communication between two peers
broadcast	sends the same input data to all connected peers
scatter	sends different parts of the input data, typically partitions, to all connected peers
onDemand	the input data is sent to one of the connected peers, the choice of which is taken at run-time on the basis of the actual work-load
fromAll	a.k.a. *all-gather*, collects different parts of the data from all connected peers combining them in a single data item
fromAny	collects one data item from one of the connected peers

Table 2.1: Collective communication patterns among `ff_dnode`s

Eventually, the third layer provides the farm, pipeline and other parallel patterns as C++ classes.

Each `ff_node` is used to run a concurrent activity in a component, and it has associated two channels: one used to receive input data (pointers) to be processed and one to deliver the (pointers to the) computed results. The `svc` method encapsulates the computation to be performed on each input datum to obtain the output result. `svc_init` and `svc_end` methods are executed when the application is started and before it is terminated. The three methods constitute the only thing the programmer has to provide to instantiate an `ff_node`.

In the current version (see Fig. 2.2), which supports distributed platforms, many graphs of `ff_node`s can be connected by way of `ff_dnode`s (which support network collective channels), thus providing a useful abstraction for effective programming of hybrid multicore and distributed platforms.

A `ff_dnode`'s external channel can be specialized to provide different patterns of communication. The set of communication collectives allows one to provide exchange of messages among a set of distributed nodes, using well-known predefined patterns. The semantics of each communication pattern currently implemented are summarized in Table 2.1.

FastFlow programming model is based on streaming of pointers, which are used as synchronisation tokens. This abstraction is kept also in the distributed version (i.e. across network channels) by way of two auxiliary methods provided by `ff_dnode` for data *marshalling* and *unmarshalling*. These (virtual) methods provide the programmers with the tools to serialise and de-serialise data flowing across `ff_dnode`s. The hand-made serialisation slightly increases the coding complexity (e.g., with respect to Java automatic serialisation) but makes it possible to build very efficient network channels. As a matter of a fact, the lesson learned from Java RMI is that automatic serialisation and high-performance can be hardly coupled.

2.2 Systems Biology and Computer Science

The recent, often astonishing developments in biology have produced a huge amount of data on the structure of living matter; consider e.g. the success of the human genome project. Less instead is known on the versatile biological functions that cells and their components display. Consequently, in the last years we have seen a shift from structure to functionality, and the growth of a new paradigm, that moves from the classical reductionist approach to a system level understanding of life. It is called *systems biology*. We can describe it as the systematic study of an organism, viewed as an integrated and interacting network of components. Instead of analyzing individual aspects of the organism, systems biology takes a *holistic* approach, focusing on all the components and the interactions among them, all as part of one system.

There is a general understanding in the scientific community that computer science will be as indispensable for biology as mathematics has been for physics. E.g., mapping the human genome would be impossible without computers, algorithms and syntax to model structures: it has been crucial representing DNA as a formal language over a four character alphabet and using search and matching algorithms over strings. Much in the same way, computer science appears to be essential for understanding the behavior of living organisms: passing from structure to functions amounts to equipping syntax with semantics. Indeed, biological systems are often described as entities (syntax) that change their state because of the occurrence of (bio-chemical) interactions, giving rise to some observable behavior (semantics).

2.2.1 Modelling biological systems

The most common approaches used by biologists to describe biological systems have been mainly based on the use of deterministic mathematical means like, e.g., Ordinary Differential Equations (ODEs for short). ODEs make it possible to abstractly reason on the behaviour of biological systems and to perform a quantitative *in silico* investigation. However, this kind of modelling becomes more and more difficult, both in the specification phase and in the analysis processes, when the complexity of the biological systems taken into consideration increases. More recently, the observation that biological systems (for example in the case of chemical instability) are inherently stochastic [64], has led a growing interest in the stochastic modelling of chemical kinetics.

Besides, the concurrently interacting structure of biological systems has inspired the possibility to describe them by means of formalisms developed in Computer Science for the description of computational entities [108]. Different formalisms have either been applied to (or have been inspired from) biological systems. Automata-based models [21, 90] have the advantage of allowing the direct use of many verification tools such as model checkers. Rewrite systems [54, 104, 30] usually allow describing biological systems with a notation that can be easily understood by biologists. Process calculi, including those commonly used to describe biological systems [108, 103, 40], have the advantage of being compositional, but their way of describing biological entities is often less intuitive. Quantitative simulations of biological models represented with these kind of frameworks

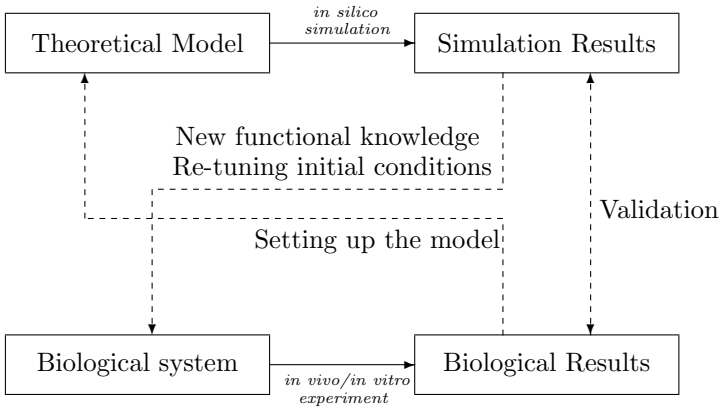


Figure 2.4: Integration of in silico simulations into the biological analysis process

(see, e.g. [103, 57, 80, 29, 58, 51]) are usually developed via a stochastic method derived from Gillespie's algorithm [68]. These formalisms, in fact, are mainly used to represent interactions between complex biological structures like cells, bacteria or genes whose results would be hard to validate in the ODE approach.

2.2.2 Simulation of biological systems

The *stochastic simulation* of biological systems is an increasingly popular technique in bioinformatics, as either an alternative or a complementary tool to traditional differential equations (ODEs) solvers. This trend has started from Gillespie's seminal work [68] (which is recapped below), in which the validity of the simulation approach is firmly assessed.

Thus, one of the most relevant contributions provided by computer science to systems biology regards the ability of running *in silico* simulations over biological systems modelled through some formalism. Figure 2.4 shows the integration schema of in silico simulations into biological analysis processes.

Gillespie SSA

The general problem studied by systems biology is to determine the status of the modelled system along its time evolution. More precisely, given the *initial state* of the modelled system and the set of *reactions* which can occur, we want to determine the status of the system at a later time.

The traditional way of treating this problem is to translate it into the mathematical language of ordinary differential equations (ODEs) and solve them. According to [68], this approach introduces some approximation induced by assuming that the time evolution of the reacting system is both continuous (i.e. population levels take values from $\mathbb{R}$) and deterministic. However, the time evolution of a reacting system is not a continuous

process, because molecular population levels obviously can change only by discrete integer amounts. Moreover, fairly simple kinetic theory arguments show that the stochastic formulation of chemical kinetics has a firmer physical basis than the deterministic formulation.

Computing a solution for the stochastic formulation of a problem by solving the *master equation* (which gives, over the time, the probability for the system to be in a certain state), is unfeasible, since the master equation is often mathematically intractable. Gillespie's *direct method* (also known as *stochastic simulation alogrithm*, SSA for short) uses a Monte Carlo procedure to numerically simulate the time evolution of the given system.

It works on a stochastic formulation of chemical kinetics that is based on the theory of collisions and that assumes a stochastic reaction constant c_μ for each considered chemical reaction R_μ. The reaction constant c_μ is such that $c_\mu dt$ is the probability that a particular combination of reactant molecules of R_μ will react in an infinitesimal time interval dt.

The probability that a reaction R_μ will occur in the whole solution in the time interval dt is given by $c_\mu dt$ multiplied by the number of distinct R_μ molecular reactant combinations. For instance, the reaction

$$R_1 : S_1 + S_2 \to 2S_1$$

will occur in a solution with X_1 molecules S_1 and X_2 molecules S_2 with probability

$$X_1 X_2 c_1 dt$$

Instead, the inverse reaction

$$R_2 : 2S_1 \to S_1 + S_2$$

will occur with probability

$$\binom{X_1}{2} c_2 dt = \frac{X_1(X_1 - 1)}{2!} c_2 dt$$

The number of distinct R_μ molecular reactant combinations is denoted with h_μ, hence, the probability of R_μ to occur in dt (denoted with $a_\mu dt$) is

$$a_\mu dt = h_\mu c_\mu dt$$

Now, assuming that $S_1, \ldots, S_n$ are the only molecules that may appear in a chemical solution, a state of the simulation is a tuple $(X_1, \ldots, X_n)$ representing a solution containing X_i molecules S_i with $i = 1, \ldots, n$. Given a state $(X_1, \ldots, X_n)$, a set of reactions $R_1, \ldots, R_M$, and a value t representing the current time, Gillespie's algorithm performs two steps:

1. The time $t + \tau$ at which the next reaction will occur is randomly chosen with τ exponentially distributed with parameter $\sum_{\nu=1}^{M} a_\nu$;

2. The reaction R_μ that has to occur at time $t+\tau$ is randomly chosen with probability $a_\mu dt$.

The function $P_g(\tau, \mu)dt$ represents the probability that the next reaction will occur in the solution in the infinitesimal time interval $(t + \tau, t + \tau + dt)$ and will be R_μ. The two steps of the algorithm imply

$$P_g(\tau, \mu)dt = P_g^0(\tau) \cdot a_\tau dt$$

where $P_g^0(\tau)$ corresponds to the probability that no reaction occurs in the time interval $(t, t + \tau)$. Since $P_g^0(\tau)$ is defined as

$$P_g^0(\tau) = e^{(-\sum_{\nu=1}^{M} a_\nu \tau)}$$

we have, for $0 \leq \tau < \infty$,

$$P_g(\tau, \mu)dt = e^{(-\sum_{\nu=1}^{M} a_\nu \tau)} \cdot a_\mu dt$$

Finally, the two steps of the algorithm can be implemented in accordance with $P_g(\tau, \mu)$ by choosing τ and μ as follows:

$$\tau = \left(\frac{1}{\sum_{\nu=1}^{M} a_\nu} \right) ln \left(\frac{1}{r_1} \right)$$

$$\mu = \text{the integer for which} \sum_{\nu=1}^{\mu-1} a_\nu < r_2 \sum_{\nu=1}^{M} a_\nu \leq \sum_{\nu=1}^{\mu} a_\nu$$

where $r_1, r_2 \in [0, 1]$ are two real values generated by a random number generator. After the execution of the two steps, the clock has to be updated to $t + \tau$ and the state has to be modified by subtracting the molecular reactants and adding the molecular products of R_μ.

2.2.3 Calculi for systems biology

In this section we will put CWC (Chapter 3) in the framework of qualitative and stochastic models for the description and analysis of biological systems.

Qualitative Models

In the last few years many formalisms originally developed by computer scientists to model systems of interacting components have been applied to Biology. Among these, there are Petri Nets [90], Hybrid Systems [21], and the π-calculus [103, 52, 109]. Moreover, new formalisms have been defined for describing biomolecular and membrane interactions [30, 40, 42, 54, 57, 107]. Others, such as P systems [104], have been proposed as biologically inspired computational models and have been later applied to the description of biological systems.

The π-calculus and new calculi based on it [57, 107] have been particularly successful in the description of biological systems, as they allow describing systems in a compositional manner. Interactions of biological components are modelled as communications on

channels whose names can be passed; sharing names of private channels allows describing biological compartments.

These calculi offer very low-level interaction primitives, but may cause the models to become very large and difficult to read. Calculi such as those proposed in [40, 42, 54] give a more abstract description of systems and offer special biologically motivated operators. However, they are often specialised to the description of some particular kinds of phenomena such as membrane interactions or protein interactions.

P systems [104] have a simple notation and are not specialised to the description of a particular class of systems, but they are still not completely general. For instance, it is possible to describe biological membranes and the movement of molecules across membranes, and there are some variants able to describe also more complex membrane activities. However, the formalism is not so flexible to allow describing easily new activities observed on membranes without extending the formalism to model such activities.

CWC can describe situations that cannot be easily captured by the previously mentioned formalisms, which consider membranes as atomic objects (extensions of P systems with objects on membranes can be found in [37, 41]). Representing the membrane structure as a multiset of the elements of interest allows the definition of different functionalities depending on the type and the number of elements on the membrane itself.

Danos and Laneve [54] proposed the κ-calculus. This formalism is based on graph rewriting where the behaviour of processes (compounds) and of set of processes (solutions) is given by a set of rewrite rules which account for, e.g., activation, synthesis and complexation by explicitly modelling the binding sites of a protein.

CLS [30] has no explicit way to model protein domains (however they can be encoded, and a variant with explicit binding has been defined in [28, 22]), but accounts for an explicit mechanism (the *looping sequences*) to deal with compartments and membranes. Thus, while the κ-calculus seems more suitable to model protein interactions, CLS allows for a more natural description of membrane interactions. Another feature lacking in many other formalisms is the capacity to express ordered sequences of elements. While we might encode ordered structures in CWC with nested compartments, CLS offers such a feature in an explicit way, thus allowing to naturally operate over proteins or DNA fragments which should be frequently defined as ordered sequences of elements.

Stochastic Models

The stochastic semantics of CWC is defined in terms of the collision-based paradigm introduced by Gillespie. A similar approach is taken in the quantitative variant of the κ-calculus ([53]) and in BioSPi ([103]). In [80], a stochastic semantics for bigraphs has been developed. An application in the field of systems biology has been provided by modelling a process of membrane budding. Compartmentalised stochastic simulations, addressing the problem of dynamic structure, have also been investigated in the domain of P systems, see, e.g., [99, 121].

A stochastic semantics for CLS (SCLS) has been defined in [29]. Such a semantics computes the transition rates by resorting to a complete counting mechanism to detect all the possible occurrences of patterns within a term. CWC rules, similar to what happens

in [121] for P systems, in Bio-PEPA [44], in [36] for a variant of the ambient calculus and in [59] for CLS, are equipped with rate functions, rather than with rate constants. Such functions may allow the definition of kinetics that are more complex than the standard mass-action ones. In particular, equipping the rewrite rules with a function leads to the definition of a stochastic semantics that can abstract from the classical one based on collision analysis (based on constant rates and practical for a very low level analysis, for example chemical interactions), and allows defining more complex rules (for higher simulation levels, for example cellular or tissue interactions) which might follow different probability distributions.

CWC has been originally proposed in [51] as a variant of (S)CLS with the aim of strongly simplifying the development of efficient automatic tools for the analysis of biological systems, while keeping the same expressiveness. The main simplification consists in the removal of the sequencing operator, thus lightening the formal treatment of the patterns to be matched in a term (whose complexity in SCLS is strongly affected by the variables matching in the sequences). Then, in [49] CWC has been extended with compartment labels, a feature that is not present in (S)CLS.

BioAmbients [107], is a calculus in which biological systems are modelled using a variant of the ambient calculus. In BioAmbients both membranes and elements are modelled by ambients, and activities by capabilities (enter, exit, expel, etc.). In [36], BioAmbients are extended by allowing the rates associated with rules to be context dependent. Dependency is realised by associating to a rule a function which is evaluated when applying the rule, and depends on the context of the application. The context contains the state of the sibling ambients, that is the ambients in parallel in the innermost enclosing ambient (membrane). The property of the context used to determine the value of the function is its volume that synthesises (with a real number) the elements present in the context.

MGS [67, 91], is a domain specific language for simulation of biological processes. The state of a dynamical system is represented by a collection. The elements in the collection represent either entities (a subsystem or an atomic part of the dynamical system) or messages (signal, command, information, action, etc.) addressed to an entity. The dynamics is defined by rewrite rules specifying the collection to be substituted through a pattern language based on the neighbourhood relationship induced by the topology of the collection. It is possible to specify stochastic rewrite strategies. In [92], this feature is used to provide the description of various models of the genetic switch of the λ phage, from a very simple biochemical description of the process to an individual-based model on a Delaunay graph topology.

Finally, the recent framework proposed by Oury and Plotkin [95] is based on stochastic multi-level multiset rewriting and is similar to CWC. Their models, constructed from species and agents (representing, respectively, atoms and compartments in CWC) evolve according to a stochastic semantics associating rates to rewrite rules. As pointed out in [95], the main difference with respect to CWC is that the analysis is strongly term rewriting oriented and compartment wrappings may not be specified explicitly (an encoding based on nested agents is shown to do the work).

2.2.4 Tools for systems biology

In this section, we will put the CWC Simulator tool in the framework of the tools for the simulation of biological systems. In particular, we will focus on tools exploiting parallel computing.

In the field of biological modeling, tools such as SPiM [101] and Dizzy [105] have been used to capture first order approximations to system dynamics using a combination of stochastic simulations and differential equation approximations. SPiM has long been the standard tool for simulating stochastic π calculus models.

Bio-PEPA [45] is a timed process algebra designed for the description of biological phenomena and their analysis through quantitative methods, such as stochastic simulations and probabilistic model-checking. Two software tools are available for modelling with Bio-PEPA: the Bio-PEPAWorkbench and the Bio-PEPA Eclipse Plugin.

The parallelisation of stochastic simulators has been extensively studied in the last two decades. Many of these efforts focus on hybrid multicore and distributed architectures.

The Swarm algorithm [106], which is well suited for biochemical pathways optimization, has been used in a distributed environment – e.g., in Grid Cellware [60], a grid-based modelling and simulation tool for the analysis of biological pathways that offers an integrated environment for several mathematical representations, ranging from stochastic to deterministic algorithms.

DiVinE is a general distributed verification environment meant to support the development of distributed enumerative model-checking algorithms, including probabilistic analysis features used for biological systems analysis [32].

StochKit [100] is a C++ stochastic simulation framework. Among other methods, it implements the Gillespie algorithm and in its second version it targets multicore platforms. It is therefore similar to our work. Anyway, it does not implement on-line trajectory reduction, which is performed in a post-processing phase.

In [77] a parallel computing platform has been employed to simulate a large biochemical network in hundreds different cellular volumes using Gillespie SSA on multiple processors. Parallel computing techniques made it possible to run massive simulations in reasonable computational times, but the analysis of the simulation results to characterize the intrinsic noise of the network has been done as a post-processing step.

The Hy3S software package [111], that includes hybrid stochastic simulation algorithms, and SRSim [70], that performs rule-based spatial modelling, are both embarrassingly parallelised by way of the MPI (Message Passing Interface) library. In this case, high latencies and communication connection problems of the computing clusters could decrease the speed efficiency.

In [86], the authors propose an adaptation for GPGPUs of the Gillespie SSA by way of the NVIDIA CUDA framework aiming to compute in parallel different trajectories (i.e. different simulation instances). The implementation, tested on two simple models, shows remarkable speedups compared to the execution on a single-core workstation. The implementation, which is hand-optimized for the specific architecture, is hardly portable on other platforms and does not implement any kind of on-line trajectory reduction. The latter problem specifically requires efficient global synchronization mechanisms that

Tool	Calculus	Simulation Schema	Parallelism	Data Analysis
CWC Simulator	CWC	Gillespie	FastFlow	online statistics
SPiM	π-calculus	Gillespie	none	none
Dizzy	Reaction Model	Gillespie, Gibson-Bruck, τ-leap, ODE	none	none
BioPEPA	Process Algebra	ODE, Gillespie	none	none
Cellware	Reaction Model	Gillespie, Gibson-Bruck, ODE	none	none
DiVinE	Model Checker	ODE	MPI	none
StochKit	Reaction Model	Gillespie, τ-leap	MPI	post-processing
StochKit2	Reaction Model	Gillespie, τ-leap	POSIX threads	post-processing
Hy3S	Reaction Model	Gibson-Bruck, Hybrid	MPI	post-processing
Li and Petzold's	Reaction Model	Gillespie	CUDA	none
StochSimGPU	Reaction Model	Gillespie, Gibson-Bruck, Li	CUDA	post-processing

Table 2.2: Comparison of some simulation tools for systems biology

are missing in GPGPUs.

StochSimGPU [79] exploits GPUs for parallel stochastic simulations of biological systems. The tool allows to compute averages and histograms of the molecular populations across the sampled realizations on the GPU. The tool relies on a GPU-accelerated version of the Matlab framework, which can be hardly compared in flexibility and performance with a C++ implementation.

A schematic comparison of the main features of the tools cited above is reported in Table 2.2.

2.2.5 Bioinformatics in the cloud

The cloud has the potentiality to become an enabling technology for bioinformatics and computational biology. It can seamlessly provide applications and their users with large amount of computing power and storage in an elastic and on-demand fashion. This naturally meets the need of simple availability of processing large amount of heterogeneous data, of storing massive amount of data and of using the existing tools in different fields of bioinformatics. The ability of managing the whole data set in the cloud, as we advocate in this work, has been widely recognized as necessary for next generation bioinformatics [123]. As an example, the typical workflow of DNA sequencing [114] foresees that biologists design the experiments and send samples to sequencing centres, which make available raw data (through specific services, such as FTP, HTTP) to biologists, who have to download and use terabytes of data. At the same time, biologists copy the data into local machines for being used by bioinformatics scientists for the subsequent data analysis. This typical workflow implies that large (possibly big) amount data are moved several times from sites to sites, thus slowing down the analysis and the interpretation of the results. These multiple data movements can be partially or entirely avoided by moving the whole workflow in the cloud.

Cloud tools for bioinformatics

DNA sequencing and sequence alignment are classic examples of computational biology application in which having computing power as more as possible is never enough [85]. Examples of these applications are: Crossbow [83], a software pipeline for genome re-sequencing analysis which runs in the cloud (according to a MapReduce paradigm [128]) on top of Hadoop [25]; CloudBurst [113], which accomplishes mapping of next-generation sequence data to the human genome for a variety of biological experiments (e.g. SNP discovery and genotyping) achieving a significant speedup with respect to sequential execution, and Myrna [82], a differential gene expression calculation tool in large RNA-Seq datasets that integrates all the RNA sequencing steps (read alignment, normalisation, aggregation and statistical modelling) in a single cloud-based computational pipeline.

DNA sequencing is not the only bioinformatics application field for which the cloud has been adopted. Another example is in-silico organ modelling, which is a relatively new method for studying the development and functionality of human, and not only, body parts with computers. In [110], for example, a model of the human liver is emulated in a cloud-based system where each liver lobule is represented by Monte Carlo samples. By using this architecture, the authors demonstrate that the parallel computing paradigm permits to develop systems emulating organs with functionalities equivalent to those of an in-vitro specimen. A multi-scale model for the progression of pancreatic cancer that can be executed on a cloud platform is presented in [63]. This platform is designed for the needs of life science and pharmaceutical research allowing the integration of physiologically based and classical approaches to model drug pharmacokinetics and pharmacodynamics as well as metabolic and signalling networks.

Protein folding simulation is another notable example of calculation intensive process. In particular, it is the process that converts a two-dimensional unfolded polypeptide in a three dimensional structure. Folding@home initiative [122] attempted to attack the problem via opportunistic computing, distributing tasks to Internet users. Although Folding@home can be executed on a multitude of hardware platforms, given the unreliability of internet computer clients, the performance of the project is hindered by errors in the local network or the computers themselves wasting, otherwise useful, computer resources. The Microsoft@home project permits the execution of generic scientific computer-intensive applications, including Folding@home, in the cloud.

Simulation modelling of biological processes is the backbone of systems biology and discrete stochastic models are particularly effective for describing molecular interaction at different levels [43]. Nevertheless, it is common knowledge that these types of stochastic simulations, as for instance the Monte Carlo ones, are computationally intensive, and among the bioinformatics applications they are the ones that could benefit from distributed implementations on the cloud.

Despite the evident advantages of carrying out simulations on the cloud, at the moment, cloud based simulators occur at a slow pace and the scientific community is not fully exploiting the opportunity to grasp the potential of the cloud paradigm. While implementations and services for Monte Carlo simulations on the cloud [63, 117, 62, 125, 93] are few, the convenience of having virtually unlimited resources will make the cloud plat-

System	Description	Used resources	Cloud	Data Analysis
CloudBurst	System to map sequence data to a reference genome	Apache Hadoop	Amazon EC2	Source code to be compiled and executed on a Hadoop cluster
Bayer Computational Systems Biology Software Suite [33]	Platform for computational biology by integrating body physiology, disease biology and molecular reaction networks	Apache Hadoop, PK-Sim, MoBi-R, MoBi-Matlab	D-Grid GmbH	Executable to be installed
Crossbow	Software pipeline for genome resequencing analysis	Bowtie, SoapSNP, Apache Hadoop	Amazon EC2	Crossbow Web Application
Folding@home	Protein folding simulation	–	Windows Azure	Folding@home website
Myrna	Calculate differential gene expression in RNA-seq datasets	Bowtie, R/Bioconductor, Apache Hadoop	Amazon EC2	Myrna Web Application

Table 2.3: Comparison of some cloud-based tools for computational biology and bioinformatics available on the web

form the perfect candidate for calculation-intensive applications.

Table 2.3 compares the features of some computational biology and bioinformatics tools freely available on the web.

From a more general perspective, given that many existing bioinformatics tools and simulators rely on web services, their transition to a cloud based infrastructure will be quite natural and we expect, in the near future, that cloud-based bioinformatics applications and services will be created at accelerating pace. Examples of such a transition, which have been already put in place, are CloudBurst [113] which (as above described) maps next generation sequencing data [118] and Cloud Blast, a "clouded" implementation of NCBI BLAST [89], which basically have kept the same web service based architecture but changed the underlying hardware infrastructure to a cloud-based one.

Cloud computing, however, poses a few "still unsolved" problems both for developers and users of cloud based software, ranging from data transfers over low-bandwidth networks to privacy and security issues. These aspects lead to inefficiency for some types of problems and future solutions should address such issues [114].

3 The Calculus of Wrapped Compartments

The modelling and analysis of biological systems has deep roots in Mathematics, specifically in the field of Ordinary Differential Equations (ODEs). Alternative approaches based on formal calculi, often derived from process algebras or term rewriting systems, provide a quite complementary way to analyse the behaviour of biological systems. These calculi allow to cope in a natural way with notions like compartments and membranes, which are not easy (sometimes impossible) to handle with purely numerical approaches, and are often based on stochastic simulation methods.

In this chapter, we present the Calculus of Wrapped Compartments (CWC) – the formal calculus we use in this work – on which we can express the compartmentalisation of a biological system whose evolution is defined by a set of rewrite rules.

In Sec. 3.1 we define the syntax of CWC terms (3.1.1), contexts (3.1.2) and rewrite rules (3.1.3); we also provide some modelling guidelines (3.1.4).

In Sec. 3.2 we define CWC quantitative systems and we present a quantitative model based on stochastic evolution (3.2.1).

The contents of this chapter have been published in [50].

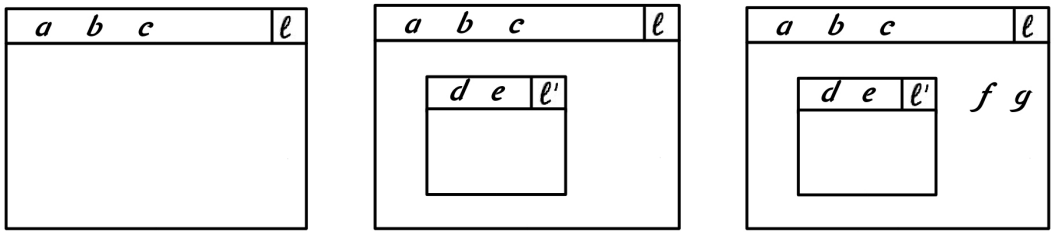
3.1 The CWC Formalism

Like most modelling languages based on term rewriting (notably CLS), a CWC model consists of a term, representing the (biological) system and a set of rewrite rules which model the transformations determining the system's evolution. Terms are defined from a set of atomic elements via an operator of compartment construction. Compartments are enriched with a nominal type, represented as a label, which identifies the set of rewrite rules which may be applied to them.

3.1.1 Terms and structural congruence

Terms of the CWC calculus are intended to represent a biological system. A *term* is a multiset of *simple terms*. Simple terms, ranged over by t, u, v, ... are built by means of the *compartment* constructor, $(- \rfloor -)^{-}$, from a set $\mathcal{A}$ of *atomic elements* (*atoms* for short), ranged over by a, b, c, ... and from a set $\mathcal{L}$ of *compartment types* (represented as *labels* attached to compartments), ranged over by $\ell, \ell', \ell_1, \ldots$ and containing a distinguished element $\top$ which characterises the top level compartment. The syntax of simple terms is defined as follows:

$$t \quad ::= \quad a \quad \mid \quad (\overline{a} \rfloor \overline{t})^{\ell}$$



(a) $(a\ b\ c \rfloor \bullet)^{\ell}$ (b) $(a\ b\ c \rfloor (d\ e \rfloor \bullet)^{\ell'})^{\ell}$ (c) $(a\ b\ c \rfloor (d\ e \rfloor \bullet)^{\ell'}\ f\ g)^{\ell}$

Figure 3.1: Some examples of CWC terms

We write $\overline{t}$ to denote a (possibly empty) multiset of simple terms $t_1 \cdots t_n$. Similarly, with $\overline{a}$ we denote a (possibly empty) multiset of atoms. The set of simple terms will be denoted by $\mathcal{T}$.

Then, a simple term is either an atom or a compartment $(\overline{a} \rfloor \overline{t})^{\ell}$ consisting of a *wrap* (represented by the multiset of atoms $\overline{a}$), a *content* (represented by the term $\overline{t}$) and a *type* (represented by the label ℓ). Note that we do not allow nested structures within wraps but only in compartment contents. We write $\bullet$ to represent the empty multiset and denote the union of two multisets $\overline{u}$ and $\overline{v}$ as $\overline{u}\ \overline{v}$. The notion of inclusion between multisets, denoted as usual by $\subseteq$, is the natural extension of the analogous notion between sets. The set of terms (multisets of simple terms) and the set of multisets of atoms will be denoted by $\overline{\mathcal{T}}$ and $\overline{\mathcal{A}}$, respectively. Note that $\overline{\mathcal{A}} \subseteq \overline{\mathcal{T}}$.

Since a term $\overline{t} = t_1 \cdots t_n$ is intended to represent a multiset we introduce a relation of structural congruence between terms of CWC defined as the least equivalence relation on terms satisfying the following rules:

$$\begin{array}{l} \overline{t}\ u\ w\ \overline{v} \equiv \overline{t}\ w\ u\ \overline{v} \\ \text{if } u \equiv w \quad \text{then} \quad \overline{t}\ u\ \overline{v} \equiv \overline{t}\ w\ \overline{v} \\ \text{if } \overline{a} \equiv \overline{b} \ \text{ and } \ \overline{t} \equiv \overline{u} \quad \text{then} \quad (\overline{a} \rfloor \overline{t})^{\ell} \equiv (\overline{b} \rfloor \overline{u})^{\ell} \end{array}$$

From now on we will always consider terms modulo structural congruence. To denote multisets of atomic elements we will sometimes use the compact notation na where a is an atomic element and n its multiplicity, so for instance $3a\,2b$ is a notation for the multiset $a\,a\,a\,b\,b$.

An example of term is $\overline{t} = 2a\ 3b\ (c\ d \rfloor e\ f)^{\ell}$ representing a multiset consisting of two atoms a and three b (for instance five molecules) and an ℓ-type compartment $(c\ d \rfloor e\ f)^{\ell}$ which, in turn, consists of a wrap (a membrane) with two atoms c and d (for instance, two proteins) on its surface, and containing the atoms e (for instance, a molecule) and f (for instance a DNA strand whose functionality can be modelled as an atomic element). See Figure 3.1 for some graphical representations.

Notation 3.1.1 (Top-level compartment) *For sake of uniformity we assume that the term representing the whole system is always a single compartment labelled $\top$ with an empty wrap, i.e., all systems are represented by a term of the shape $(\bullet \rfloor \overline{t})^{\top}$, which we will also write as $\overline{t}$ for simplicity.*

3.1.2 Contexts

The notion of reduction in CWC is formalised via the notion of reduction context. To define them, the syntax of terms is enriched with a new element $\Box$ representing a hole which can be filled only by a *single* compartment. *Reduction contexts* (ranged over by C) are defined by:

$$C \quad ::= \quad \Box \quad \mid \quad (\overline{a} \rfloor C \ \overline{t})^{\ell}$$

where $\overline{a} \in \overline{\mathcal{A}}$, $\overline{t} \in \overline{\mathcal{T}}$ and $\ell \in \mathcal{L}$. Note that, by definition, every context contains a single hole $\Box$. The set of contexts is denoted by $\mathcal{C}$.

Given a compartment $t = (\overline{a} \rfloor \overline{u})^{\ell}$ and a context C, the compartment obtained by filling the hole in C with t is denoted by $C[t]$. For instance, if $t = (\overline{a} \rfloor \overline{u})^{\ell}$ and $C = (\overline{b} \rfloor \Box \ \overline{c})^{\ell'}$, then $C[t] = (\overline{b} \rfloor (\overline{a} \rfloor \overline{u})^{\ell} \ \overline{c})^{\ell'}$.

The composition of two contexts C and C', denoted by $C[C']$, is the context obtained by replacing $\Box$ with C' in C. For example, given $C = (a \rfloor \Box \ b)^{\ell}$, $C' = (c \rfloor \Box \ d \ e)^{\ell'}$, we get $C[C'] = (a \rfloor (c \rfloor \Box \ d \ e)^{\ell'} \ b)^{\ell}$.

The following proposition follows straightforwardly by the definitions of term, structural congruence and context.

Proposition 3.1.2 (Unique decomposition) *Given a term $\overline{t}$ and a subterm $\overline{u}$ occurring in $\overline{t}$ there is a unique context C such that $\overline{t} = C[(\overline{a} \rfloor \overline{t'})^{\ell}]$ and $\overline{u} \subseteq \overline{a}$ or $\overline{u} \subseteq \overline{t'}$.*

Note that this property would not be true if we allow $C, \overline{t}$ to be a context.

3.1.3 Rewrite rules and qualitative reduction semantics

A rewrite rule is defined by a pair of compartments (possibly containing variables), which represent the patterns along which the system transformations are defined. The choice of defining rules at the level of compartments simplifies the formal treatment, allowing a uniform presentation of the system semantics.

In order to formally define the rewrite rules, we introduce the notion of open term (a term containing variables) and pattern (an open term that may be used as left part of a rewrite rule). To respect the syntax of terms, we distinguish between "wrap variables" which may occur only in compartment wraps (and can be replaced only by multisets of atoms) and "content variables" which may only occur in compartment contents or at top level (and can be replaced by arbitrary terms)

Let $\mathcal{V}_{\overline{\mathcal{T}}}$ be a set of *content variables*, ranged over by X, Y, Z, and $\mathcal{V}_{\overline{\mathcal{A}}}$ a set of *wrap variables*, ranged over by x, y, z such that $\mathcal{V}_{\overline{\mathcal{T}}} \cap \mathcal{V}_{\overline{\mathcal{A}}} = \emptyset$. We denote by $\mathcal{V}$ the set of all variables $\mathcal{V}_{\overline{\mathcal{T}}} \cup \mathcal{V}_{\overline{\mathcal{A}}}$, and with ρ any variable in $\mathcal{V}$. *Open terms* are terms which may contain occurrences of wrap variables in compartment wraps and content variables in compartment contents. Similarly to terms, open terms are defined as multisets $\overline{o}$ of simple open terms defined in the following way:

$$\begin{array}{lcl|l|l} o & ::= & a & X & (\overline{q} \rfloor \overline{o})^{\ell} \\ q & ::= & a & x & \end{array}$$

(i.e. $\overline{q}$ denotes a multiset formed only of atomic elements and wrap variables). Let $\mathcal{O}$ and $\overline{\mathcal{O}}$ denote the set of simple open terms and the set of open terms (multisets of simple open terms), respectively. An open term is *linear* if each variable occurs in it at most once.

An *instantiation* (or substitution) is defined as a partial function $\sigma : \mathcal{V} \to \overline{\mathcal{T}}$. An instantiation must preserve the type of variables, thus for $X \in \mathcal{V}_{\overline{\mathcal{T}}}$ and $x \in \mathcal{V}_{\overline{\mathcal{A}}}$ we have $\sigma(X) \in \overline{\mathcal{T}}$ and $\sigma(x) \in \overline{\mathcal{A}}$, respectively. Given $\overline{o} \in \overline{\mathcal{O}}$, with $\overline{o}\,\sigma$ we denote the term obtained by replacing each occurrence of each variable $\rho \in \mathcal{V}$ appearing in $\overline{o}$ with the corresponding term $\sigma(\rho)$.

Let Σ denote the set of all the possible instantiations and $\mathit{Var}(\overline{o})$ denote the set of variables appearing in $\overline{o} \in \overline{\mathcal{O}}$.

To define the rewrite rules, we first introduce the notion of patterns, which are particular simple open terms representing the left hand side of a rule. *Patterns*, ranged over by p, are the *linear* simple open terms defined in the following way:

$$p \quad ::= \quad (\overline{a}\ \ x \rfloor \overline{b}\ \ \overline{p}\ \ X)^{\ell}$$

where $\overline{a}$ and $\overline{b}$ are multisets of atoms, $\overline{p}$ is a multiset of pattern, x is a wrap variable, X is a content variable and the label ℓ is called the *type of the pattern*. The set of patterns is denoted by $\mathcal{P}$. Patterns are intended to match with compartments. Note that we force *exactly* one variable to occur in each compartment content and wrap. This prevents ambiguities in the instantiations needed to match a given compartment.[1] The linearity condition, in biological terms, corresponds to excluding that a transformation can depend on the presence of two (or more) identical (and generic) components in different compartments (see also [95]).

Some examples of patterns are:

- $(x \rfloor a\ \ b\ \ X)^{\ell}$ which matches with all compartments of type ℓ containing at least one occurrence of a and one of b.
- $(x \rfloor (a\ \ y \rfloor Y)^{\ell_1}\ \ X)^{\ell_2}$ which matches with compartments of type ℓ_2 containing a compartment of type ℓ_1 with at least an a on its wrap.

A *rewrite rule* is a pair (p, o), denoted by $p \longmapsto o$, where $p = (\overline{a}\ \ x \rfloor \overline{b}\ \ \overline{p}\ \ X)^{\ell} \in \mathcal{P}$ and $o = (\overline{q} \rfloor \overline{o})^{\ell} \in \mathcal{O}$ are such that $\mathit{Var}(o) \subseteq \mathit{Var}(p)$. Note that rewrite rules must respect the type of the involved compartments. A rewrite rule $p \longmapsto o$ then states that a compartment $p\sigma$, obtained by instantiating variables in p by some instantiation function σ, can be transformed into the compartment $o\sigma$ with the same type ℓ of p. Linearity is not required in the r.h.s. of a rule, thus allowing duplication (or erasure).

A CWC *system* over a set $\mathcal{A}$ of atoms and a set $\mathcal{L}$ of labels is represented by a set $\mathcal{R}_{\mathcal{A},\mathcal{L}}$ ($\mathcal{R}$ for short when $\mathcal{A}$ and $\mathcal{L}$ are understood) of rewrite rules over $\mathcal{A}$ and $\mathcal{L}$.

[1] The presence of two (or more) variables in the same compartment content or wrap, like in $(x \rfloor a\ \ X\ \ Y)^{\ell}$, would introduce the possibility of matching the same path in different although equivalent ways. For instance we could match a term $(x \rfloor a\ \ a\ \ b\ \ b)^{\ell}$ with $X = a,\ Y = b\ \ b$ or $X = a\ \ b,\ Y = b$, etc.

A *transition* between two terms t and u of a CWC system $\mathcal{R}$ (denoted $t \longrightarrow u$) is defined by the following rule:

$$\frac{R = p \longmapsto o \in \mathcal{R} \quad \sigma \in \Sigma \quad C \in \mathcal{C}}{C[p\sigma] \xrightarrow{R} C[o\sigma]}$$

where $C[p\sigma] \equiv t$ and $C[o\sigma] \equiv u$.

In a rule $p \longmapsto o$ the pattern p represents a compartment containing the reactants of the reaction that will be simulated. The crucial point for determining an application of the rule to a term t is to find the compartments matching with p (i.e. the compartments in which the corresponding reaction can take place).

Note that the applicability of a rewrite rule depends on the type of the involved compartments but not on the context in which it occurs. This corresponds to the assumption that only the compartment type can influence the kind of reaction that takes place in them but not their position in the system.

Note that a same pattern can have more than one match in a term. Take for instance the term $(a \rfloor\ 2b\ 2c)^\ell\ (a\ b \rfloor 4b)^\ell$ and the pattern $p = (a\ x \rfloor b\ X)^\ell$. Then p matches the first compartment with the substitution $\sigma_1(x) = \bullet,\ \sigma_1(X) = b\ 2c$ and the second compartment with the substitution $\sigma_2(x) = b,\ \sigma_2(X) = 3b$

Remark 3.1.3 *(i) The same reduction rule can determine different transitions of the same term if applied in different contexts. Consider, for instance, the rule (p, o) where $p = (a\ x \rfloor b\ X)^\ell$ and $o = (a\ x \rfloor c\ X)^\ell$. Now consider the term $t = (\bullet \rfloor (a \rfloor b\ b')^\ell\ (a \rfloor b\ b'')^\ell)^\top$. Considering different contexts we can have both $t \longrightarrow (\bullet \rfloor (a \rfloor c\ b')^\ell\ (a \rfloor b\ b'')^\ell)^\top$ and $t \longrightarrow (\bullet \rfloor (a \rfloor b\ b')^\ell\ (a \rfloor c\ b'')^\ell)^\top$*

(ii) For some rewrite rules $\ell : p \longmapsto o$ there may be, in general, different substitutions σ such that $p\sigma \equiv t$ (for some term t) but the results $o\sigma$ produced by them are different. Consider, for instance, the rewrite rule $(y \rfloor a\ (b\ x \rfloor X)^\ell\ Y)^\top \longmapsto (y \rfloor (a\ b\ x \rfloor X)^\ell\ Y)^\top$ modelling a catalysed membrane joining at top level. In this case, a term $t = (\bullet \rfloor a\ (b\ b \rfloor c)^\ell\ (b \rfloor c)^\ell)^\top$ can make a transition in two different terms, depending on which membrane will be joined by the element a. Namely, $(\bullet \rfloor (a\ b\ b \rfloor c)^\ell\ (b \rfloor c)^\ell)^\top$, given an instantiation σ such that $\sigma(x) = b$ and $(\bullet \rfloor (b\ b \rfloor c)^\ell\ (a\ b \rfloor c)^\ell)^\top$, given an instantiation σ' such that $\sigma'(x) = \bullet$. We remark that this can happen only when compartments are involved in the rewriting. We will need to take it into account in the stochastic approach.

Notation 3.1.4 (Rules that involve only content or wrap) *Usually, rules involve only the content or the wrap of a compartment. Moreover, in a rule $(\overline{a}\ x \rfloor \overline{b}\ \overline{p}\ X)^\ell \longmapsto (\overline{q} \rfloor \overline{o})^\ell$ very often X has single occurrence, at top level, in $\overline{o}$ and x in $\overline{q}$. We therefore introduce the following notations:*

- $\ell : \overline{a}\ \overline{p} \longmapsto_{\mathsf{c}} \overline{o}$ *(or simply $\ell : \overline{a}\ \overline{p} \longmapsto \overline{o}$) as a short for $(x \rfloor \overline{a}\ \overline{p}\ X)^\ell \longmapsto (x \rfloor \overline{o}\ X)^\ell$, and*
- $\ell : \overline{a} \longmapsto_{\mathsf{w}} \overline{b}$ *as a short for $(\overline{a}\ x \rfloor X)^\ell \longmapsto (\overline{b}\ x \rfloor X)^\ell$*

where x and X are canonically chosen variables not occurring in $\overline{a}$, $\overline{p}$, $\overline{o}$ or $\overline{b}$. Note that, according to Notation 3.1.1, rules of the shape $\top : \overline{a} \longmapsto_{\mathsf{w}} \overline{b}$ are forbidden (since the top level compartment must always have an empty wrap).

3.1.4 Modelling guidelines

In this section we give some explanations and general hints about how CWC could be used to represent the behaviour of various biological systems. Here, entities are represented by terms of the rewrite system, and events by rewrite rules.

First of all, we should select the biomolecular entities of interest. Since we want to describe cells, we consider molecular populations and membranes. Molecular populations are groups of molecules that are in the same compartment of the cells and inside them. As we have said before, molecules can be of many types: we classify them as proteins, chemical moieties and other molecules.

Membranes are considered as elementary objects: we do not describe them at the level of the phospholipids they are made of. The only interesting properties of a membrane are that it may have a content (hence, create a compartment) and that in its phospholipid bilayer various proteins are embedded, which act for example as transporters and receptors. Since membranes are represented as multisets of the embedded structures, we are modelling a fluid mosaic in which the membranes become similar to a two-dimensional liquid where molecules can diffuse more or less freely [119].

Compartment labels are useful to identify the kind of a compartment and the properties which are common for each compartment of that type. For example, we may use compartment labels to denote the nuclei of a set of cells, the different organelles, etc..

Table 3.1 lists the guidelines for the abstraction into CWC rules of some basic biomolecular events, some of which will be used in our experiments. Entities are associated with CWC terms: elementary objects (genes, domains, etc...) are modelled as atoms, molecular populations as CWC terms, and membranes as atom multisets. Biomolecular events are associated with CWC rewrite rules.

The simplest kind of event is the change of state of an elementary object. Then, there are interactions between molecules: in particular complexation, decomplexation and catalysis. Interactions could take place between simple molecules, depicted as single symbols, or between membranes and molecules: for example a molecule may cross or join a membrane. There are also interactions between membranes: in this case there may be many kinds of interactions (fusion, vesicle dynamics, etc. . .). Finally, we can model a state change of a compartment (for example a cell moving onto a new phase during the cell cycle), by updating its label.[2] Changing a label of a compartment implies changing the set of rules applicable to it. This can be used, e.g., to model the different activities of a cell during the different phases of its cycle.

[2]Note that, like in the other cases, this reaction is intended to take place in a compartment of a type ℓ''. Without the simplification made in Table 3.1 this rule should be written as $\ell'' : (x \rfloor X)^{\ell} \longmapsto (x \rfloor X)^{\ell'}$.

Biomolecular Event	**CWC Rewrite Rules**
State change (in content)	$a \longmapsto_{\mathsf{c}} b$
State change (on membrane)	$a \longmapsto_{\mathsf{w}} b$
Complexation (in content)	$a\ b \longmapsto_{\mathsf{c}} c$
Complexation (on membrane)	$a\ b \longmapsto_{\mathsf{w}} c$ $a\ (b\ x \rfloor X)^{\ell} \longmapsto_{\mathsf{c}} (c\ x \rfloor X)^{\ell}$ $(b\ x \rfloor a\ X)^{\ell} \longmapsto_{\mathsf{c}} (c\ x \rfloor X)^{\ell}$
Decomplexation (in content)	$c \longmapsto_{\mathsf{c}} a\ b$
Decomplexation (on membrane)	$c \longmapsto_{\mathsf{w}} a\ b$ $(c\ x \rfloor X)^{\ell} \longmapsto_{\mathsf{c}} a\ (b\ x \rfloor X)^{\ell}$ $(c\ x \rfloor X)^{\ell} \longmapsto_{\mathsf{c}} (b\ x \rfloor a\ X)^{\ell}$
Membrane crossing	$a\ (x \rfloor X)^{\ell} \longmapsto_{\mathsf{c}} (x \rfloor a\ X)^{\ell}$ $(x \rfloor a\ X)^{\ell} \longmapsto_{\mathsf{c}} a\ (x \rfloor X)^{\ell}$
Catalysed membrane crossing	$a\ (b\ x \rfloor X)^{\ell} \longmapsto_{\mathsf{c}} (b\ x \rfloor a\ X)^{\ell}$ $(b\ x \rfloor a\ X)^{\ell} \longmapsto_{\mathsf{c}} a\ (b\ x \rfloor X)^{\ell}$
Membrane joining	$a\ (x \rfloor X)^{\ell} \longmapsto_{\mathsf{c}} (a\ x \rfloor X)^{\ell}$ $(x \rfloor a\ X)^{\ell} \longmapsto_{\mathsf{c}} (a\ x \rfloor X)^{\ell}$
Catalysed membrane joining	$a\ (b\ x \rfloor X)^{\ell} \longmapsto_{\mathsf{c}} (a\ b\ x \rfloor X)^{\ell}$ $(b\ x \rfloor a\ X)^{\ell} \longmapsto_{\mathsf{c}} (a\ b\ x \rfloor X)^{\ell}$ $(x \rfloor a\ b\ X)^{\ell} \longmapsto_{\mathsf{c}} (a\ x \rfloor b\ X)^{\ell}$
Compartment state change	$(x \rfloor X)^{\ell} \longmapsto_{\mathsf{c}} (x \rfloor X)^{\ell'}$

Table 3.1: Guidelines for modelling biomolecular events in CWC, written in the compact notation of 3.1.4. The types (labels) associated to the rules are omitted for simplicity.

3.2 Quantitative CWC Systems

In order to make the formal framework suitable for modelling quantitative aspects of biological systems, each transition is usually associated with a numerical parameter characterizing the kinetic rate of the corresponding reaction.

In a stochastic simulation algorithm, this parameter and the quantity of reagents involved contribute *stochastically* to determine the next state of the system and the time needed to reach it. The system is then described as a Continuous Time Markov Chain (CTMC) [98]. This allows to simulate its evolution by means of standard simulation algorithms (see e.g. [68]). Stochastic simulation techniques can be applied to all CWC systems but, in several cases, at a high computational cost. The deterministic method based on pure ODEs is computationally more efficient, but can be applied, in general,

only to systems in which compartments are absent or have a fixed, time-independent, structure.

In our calculus we will associate to a reaction a *rate function* having a parameter depending on the overall content of the compartment in which the reaction takes place. This allows to tailor the reaction rates on the specific characteristics of the system, as for instance when representing nonlinear reactions as it happens for Michaelis–Menten kinetics, or to describe more complex interactions involving compartments that may not follow the standard mass action rate. These latter, more classical, collision based stochastic semantics (see [68]) can be encoded as a particular choice of the rate function (see 3.2.1). A similar approach is used in [59] to model reactions with inhibitors and catalysts in a single rule.

Obviously some care must be taken in the choice of the rate function: for instance it must be complete (defined on the domain of the application) and non-negative.

Definition 3.2.1 *A* quantitative *rewrite rule is a triple* (p, o, f)*, denoted* $p \overset{f}{\longmapsto} o$*, where* (p, o) *is a rewrite rule and* $f : \Sigma \to \mathbb{R}^{\geq 0}$ *is the* rate function *associated to the rule.*[3]

The rate function takes an instantiation σ as parameter. Such an instantiation models the actual compartment content determining the structure of the environment in which the l.h.s. of a rule matches and that may actively influence the rule application. Notice that, different instantiations that allow the l.h.s. p of a rule to match a term $\bar{t}$ can produce different outcomes which could determine different rates in the associated transitions.

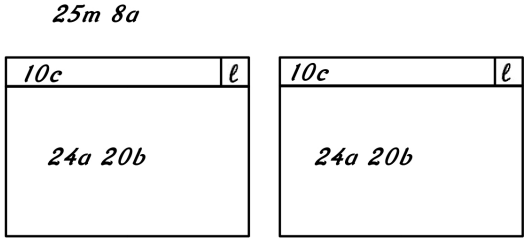
In the following we will use the function $\textsc{Occ} : \mathcal{A} \times \overline{\mathcal{T}} \to \mathbb{N}$ to count the occurrences of an atom within the multiset of atoms at the top level of a term. Namely, $\textsc{Occ}(b, \bar{t})$ returns the number of occurrences of the atom b at the top level of $\bar{t}$.

Example 3.2.2 *Consider again the term given in Remark 3.1.3. If the rate function of the rewrite rule is defined as* $f(\sigma) = 0.0002 \cdot (\textsc{Occ}(b, \sigma(x)) + 1)$*, the initial term* $\bar{t}$ *results in* $(a\ \ b\ \ b \rfloor c)^{\ell}\ \ (b \rfloor c)^{\ell}$ *with a rate* 0.0004 *and in the term* $(b\ \ b \rfloor c)^{\ell}\ \ (a\ \ b \rfloor c)^{\ell}$ *with rate* 0.0002*.*

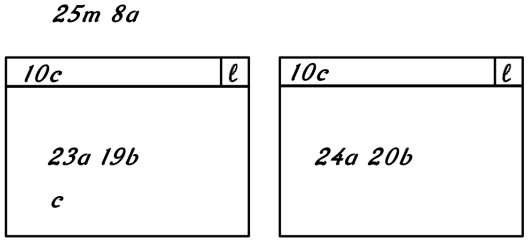
We already mentioned that equipping rewrite rules with a function leads to the definition of a stochastic semantics that can abstract from the classical one based on collision analysis (practical for very low level simulations, for example chemical interactions), and allows defining more complex rules (for higher simulation levels, for example cellular or tissue interactions) which might follow different probability distributions. An intuitive example could be a simple membrane interaction: in the presence of compartments, a system could not be considered, in general, as well stirred. In such a case, the classical collision based analysis could not always produce faithful simulations and more factors (encapsulated within the context in which a rule is applied) should be taken into account.

A *quantitative* CWC *system* over a set $\mathcal{A}$ of atoms and a set $\mathcal{L}$ of labels is represented by a set $\mathcal{R}_{\mathcal{A},\mathcal{L}}$ ($\mathcal{R}$ for short when $\mathcal{A}$ and $\mathcal{L}$ are understood) of quantitative rewrite rules over $\mathcal{A}$ and $\mathcal{L}$.

[3] The value 0 in the codomain of f models the situations in which the given rule cannot be applied, for example when the particular environment conditions forbid the application of the rule.



(a) $\bar{t}$



(b) $\overline{t'}$

Figure 3.2: Application of a rule to a CWC term

3.2.1 Stochastic evolution

In the stochastic framework, the rate of a transition is used as the parameter of an exponential distribution modelling the time spent to complete the transition. A quantitative CWC system $\mathcal{R}$ defines a Continuous Time Markov Chain (CTMC) in which the rate of a transition $C[p\sigma] \xrightarrow{R} C[o\sigma]$ is given by $f(\sigma)$, where the rule $R = (p, o, f) \in \mathcal{R}$ is the quantitative rule which determines a transition.[4] The so defined CTMC determines the *stochastic reduction semantics* of CWC.

When applying a simulation algorithm to a CWC system we must take into account, at a given time, all the system transitions (with their associated rates) that are possible at that point. They are identified by:

- the rewrite rule applied;
- context which selects the compartment in which the rule is applied;
- the outcome of the transition.

Remark 3.2.3 *We must take some care in identifying transitions involving compartments. For instance, if we consider the CWC term*

$$\bar{t} = 25m\ \ 8a\ \ (10c \rfloor 24a\ \ 20b)^{\ell}\ \ (10c \rfloor 24a\ \ 20b)^{\ell}$$

[4] When it does not give rise to ambiguities we omit the label R, identifying the rewrite rule, from the stochastic transition.

shown in Figure 3.2 (a) there are two compartments that are exactly the same. If we apply to $\bar{t}$ the rule $\ell : a\ b \longmapsto c$ we obtain the term $\overline{t'}$ shown in Figure 3.2 (b). Actually, starting from $\bar{t}$ there are two compartments on which the rule can be applied, producing the same term $\overline{t'}$ (up to structural congruence). Although the transition is considered as one (up to structural congruence), the quantitative evolution must take this possibility into account by counting two transitions.

From the transition rates we can define, following a standard simulation procedure [68], the exponential probability distribution of the moment in which the next reaction will take place and the probability distribution of the next transition that will take place.

In particular, given a term $\bar{t}$ and a global time δ, we first identify all the transitions $e_1, \ldots, e_M$ that can be applied to $\bar{t}$. Note that a transition is identified by both the corresponding rewrite rule and the compartment in which it takes place (see also Remarks 3.1.3 and 3.2.3). Let $\pi_1, \ldots, \pi_M$ be the corresponding rates. Defining $\pi = \sum_{i=1}^{M} \pi_i$, the simulation procedure allows to determine, following Gillespie's direct method:

1. The time $\delta + \tau$ at which the next stochastic transition will occur, randomly chosen with τ exponentially distributed with parameter π;
2. The transition e_i that will occur at time $\delta + \tau$, randomly chosen with probability $\frac{\pi_i}{\pi}$.

Mass action law

Gillespie's stochastic simulation algorithm is defined essentially for well stirred systems, confined to a constant volume and in thermal equilibrium at some constant temperature. In these conditions we can describe the system state by specifying only the molecular populations, ignoring the positions and velocities of the individual molecules. Different approaches such as Molecular Dynamics, Partial Differential Equations or Lattice-based methods are required in case of molecular crowding, anisotropy of the medium or canalisation.

We might restrict CWC in order to match Gillespie's framework. Namely, since we just need to deal with simple molecular populations, we might restrict terms to multisets of atoms.

The usual notation for chemical reactions can be expressed by:

$$n_1 a_1 + \ldots + n_\rho a_\rho \xrightarrow{k} n'_1 b_1 + \ldots + n'_\gamma b_\gamma \tag{3.1}$$

where, $a_i, \ldots, a_\rho$ and $b_i, \ldots, b_\gamma$ are the reagents and product molecules, respectively, $n_i, \ldots, n_\rho$ and $n'_i, \ldots, n'_\gamma$ are the stoichiometric coefficients and k is the kinetic constant.

We only consider now rewrite rules modelling chemical reactions as in reaction 3.1. A chemical reaction of the form 3.1 (that takes place within a compartment of type ℓ) can be expressed by the following CWC rewrite rule:

$$\ell : n_1 a_1\ \ldots\ n_\rho a_\rho \xmapsto{f} n'_1 b_1\ \ldots\ n'_\gamma b_\gamma \tag{3.2}$$

which is short for $(x \rfloor n_1 a_1 \ \ldots \ n_\rho a_\rho \ X)^\ell \xmapsto{f} (x \rfloor n'_1 b_1 \ \ldots \ n'_\gamma b_\gamma \ X)^\ell$, where the rate function f of rule 3.2 should be suitably defined to model Gillespie's assumption of well stirred systems. In particular, the framework defined by Gillespie, based on molecular collision analysis, leads to binomial distributions of the reagents involved. Namely, we define the rate function f as:

$$f(\sigma) = \binom{\textsc{Occ}(a_1, \sigma(X)) + n_1}{n_1} \cdot \ldots \cdot \binom{\textsc{Occ}(a_\rho, \sigma(X)) + n_\rho}{n_\rho} \cdot k \tag{3.3}$$

where k is the kinetic constant of the modelled chemical reaction.

When the stoichiometric coefficients are low (e.g. ≤ 2) and the molecular populations are high, this can be approximated as:

$$\frac{(\textsc{Occ}(a_1, \sigma(X)) + n_1)^{n_1} \cdot \ldots \cdot (\textsc{Occ}(a_\rho, \sigma(X) + n_\rho))^{n_\rho}}{n_1 \cdot \ldots \cdot n_\rho} \cdot k \tag{3.4}$$

By construction, the following holds.

Fact 3.2.4 *Molecular populations defined as multisets of atoms that evolve according to a fixed set of transformations of the form given by reaction 3.1, represented by rule 3.2, interpret into the stochastic semantics of CWC the law of mass action within Gillespie's framework for the evolution of chemically reacting systems.*

Example: Prey-Predator-Virus

In order to illustrate the quantitative semantics of CWC we consider, as a running example, a toy case study derived from the well-known Lotka-Volterra prey-predator dynamics. Let us consider the prey-predator oscillatory dynamics to be confined into a compartment *IN* interfered with rare events causing dramatic changes in the species evolution. A rare event like these could be schematically represented as a viral epidemic entering and exiting compartment *IN* with a relatively slow rate. Once inside the compartment *IN* the viral epidemic has the capability of killing some preys.

The set of CWC rules modelling this example – which we call the *Prey-Predator-Virus* model – is given in Figure 3.3. The preys (atoms a) and predators (atoms b) are located

$$\begin{array}{ll}
(N_1) & \top : \mathit{Vir}\ (x \rfloor \ X)^{IN} \overset{0.03}{\longmapsto} (x \rfloor \mathit{Vir}\ X)^{IN} \\
(N_2) & \top : (x \rfloor \mathit{Vir}\ X)^{IN} \overset{0.1}{\mapsto} \mathit{Vir}\ (x \rfloor \ X)^{IN} \\
(B_1) & \mathit{IN} : a\ X \overset{1}{\mapsto} a\ a\ X \\
(B_2) & \mathit{IN} : a\ b\ X \overset{0.001}{\longmapsto} b\ b\ X \\
(B_3) & \mathit{IN} : b\ X \overset{1}{\mapsto} X \\
(B_4) & \mathit{IN} : \mathit{Vir}\ a\ X \overset{0.1}{\mapsto} \mathit{Vir}\ X
\end{array}$$

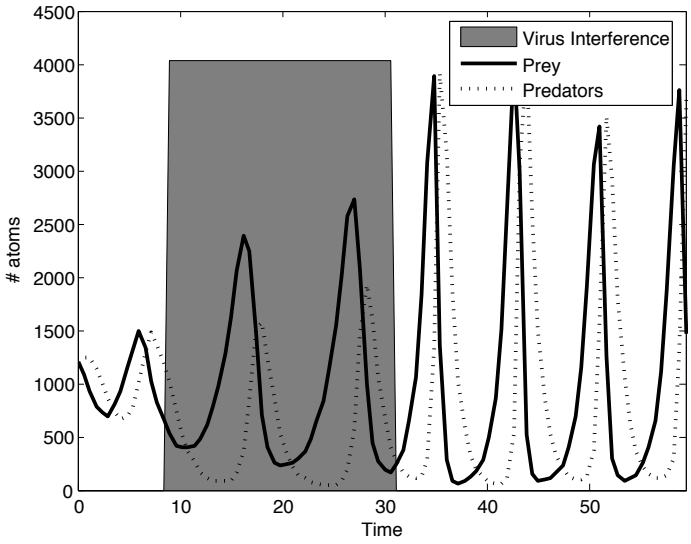
Figure 3.3: CWC rules for the *Prey-Predator-Virus* dynamics

in compartment IN and follow a dynamics given by the rules (B_1),(B_2) and (B_3). The viral epidemic (atom Vir) enters and leaves the compartment with rules (N_1) and (N_2) respectively, and kills the preys with rule (B_4).

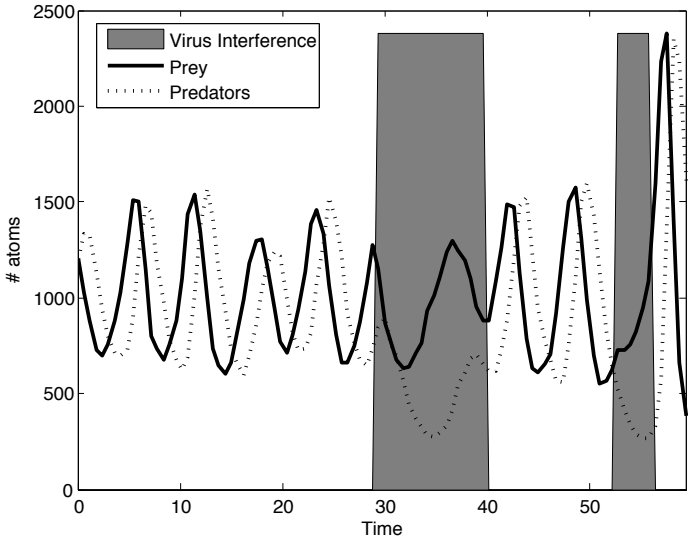
The simulations are performed for 60 time units, with the starting term:

$$Vir\ (\bullet \rfloor 1200a\ 1200b)^{IN}.$$

Several stochastic simulations of the toy case study were performed, showing different possible system evolutions of the dynamics of the species inside the compartment IN depending on the viral epidemic factor. Two of these runs are shown in Figure 3.4. A characteristic of this example is that the evolution of the system is strongly determined by the virus epidemic random event that can change dramatically the dynamics of the species.



(a) First experiment



(b) Second experiment

Figure 3.4: Two different simulation runs of the *Prey-Predator-Virus* model, showing the different behaviours of the dynamics of the species inside the compartment *IN* depending on the viral epidemic factor

4 The Simulation/Analysis Workflow

A biological scientist aiming to study the behaviour of a system by means of stochastic simulation, basically has to perform two operations: execute several *simulation* instances of the system under investigation – modelled through some formalism – and filter the simulation outcomes with *analysis* methods (e.g. statistics, mining).

In this chapter, we outline a general simulation/analysis workflow, which optimisation is the main topic of this work.

In Sec. 4.1 we discuss on the main aspects that should be taken into account while designing a modern simulation tool for biological systems.

In Sec. 4.2 we review the most common optimisation approaches, highlighting their main drawbacks.

4.1 What do Bioinformatics Scientists Need?

To be functional, simulation tools for systems biology should effectively support scientists in executing the two main steps they have to perform when studying a biological system by means of stochastic simulation, namely:

- the *simulation* step, in which independent simulation instances of the modelled system are generated and executed;
- the *analysis* step, in which simulation results are merged and filtered with a set of analysis (e.g. statistics, mining) engines.

The design (and optimisation) of such tools under the perspective of increasingly complex modelled systems – thus requiring more and more complex analysis – and modern high-performance computing platforms is a challenging task, since a number of aspects should be taken into account:

1. *multiple analysis modules*: in many biological case studies the searched pattern in experimental results is unknown and might require to try different kinds of analysis. Therefore tools should support the execution of multiple analysis engines and the addition of new ones in a plug-and-play fashion.
2. *statistical significance*: in order to infer statistically meaningful results, large samples should be collected, then many simulation instances have to be executed and tools should scalable in performance;
3. *high-precision*: the sampling resolution depends on the required accuracy, thus fine grain sampling resolution could be required to discriminate system state changes

(in particular for multi-stable systems). This implies that tools should be able to manage (analyse and eventually store) high-frequency data streams coming from simulation instances;

4. *low-latency*: during the initial phase of the study, the (stochastic) parameters of the model need to be fine-tuned. To this aim, the whole experiment is usually repeated several times. In this phase, the time-to-solution is a crucial aspect, thus the execution of the simulation instances should be as fast as possible and the analysis should be ideally a real-time process.

5. *multicore platforms*: either in the form of commodity desktop workstations or interconnected in (physical or virtualised) clusters, multicores are increasingly diffused in scientific laboratories. Thus modern tools should efficiently exploit their high computational power.

Aspects 1, 2, 3 and 4 suggest that modern simulator tools should feature fast execution of simulation instances, multiple analysis (and eventually storage) of large (even big) high frequency data streams, all in a low-latency fashion. This is enough to claim that the simulation/analysis process on biological systems at high-precision happens to be a computationally expensive task.

The high computational cost of stochastic simulations has led, in the last two decades, to a number of attempts to accelerate them up by exploiting parallel computing. The most common approach – which naturally couples with distributed HPC infrastructures (e.g. grid, clouds) which have been the reference architectures in scientific laboratories in the last decades – results in a family of solutions that we call "embarassingly parallel". As discussed in Section 4.2, this approach have several drawbacks that are further exacerbated on modern multicore-based computing platforms.

In this work we advocate a novel approach, based on regarding the whole simulation/analysis process as a monolithic stream-processing workflow, thus naturally dealing with on-line (possibly real-time) analysis, and optimising it by way of high-level parallel programming frameworks, in order to guarantee performance portability and low-effort porting on heterogeneous platforms.

4.2 The "Embarrassingly Parallel" Approach

The most common approach for the optimisation of simulation tools consists in executing (independent) simulation instances in an *embarrassingly parallel* fashion, executing a partition of the instances on different machines and demoting the analysis to a secondary aspect treated with off-line post-processing tools (Fig. 4.1).

This approach has been often coupled with High Performance Computing (HPC) infrastructures, such as grid or clusters of multi-/many-core. However, it suffers from some drawbacks from design, performance and usability viewpoints.

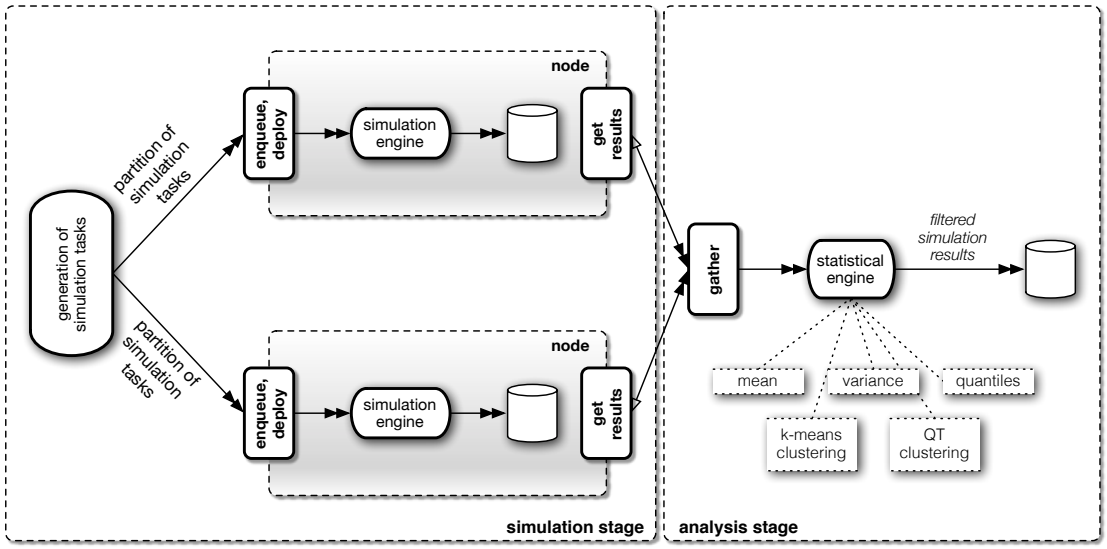


Figure 4.1: The "embarrassingly parallel" approach

4.2.1 Usability drawbacks

Traditional HPC platforms are expensive to deploy (and rent); their configuration is hardly customisable. Moreover, HPC platforms suffer from reduced interactivity and might induce slow time-to-solution. Each experiment requires to enqueue the simulations in a shared environment, deploy initial data, simulate the model, gather results from a distributed environment, post-process them (often sequentially) and then eventually access the results. Moreover, as we discussed in aspect 4, this process is typically repeated several times to fine-tune simulation parameters.

4.2.2 Performance drawbacks

Demoting the analysis of results to a secondary aspect, without considering it in performance evaluation, is definitively a too coarse simplification. When dealing with multiple statistical and mining engines working on large data streams, the analysis of results could result in a very expensive task, comparable in cost to the execution of the simulation instances. Thus, the "sequencing" of simulation and analysis phases actually slows down the design-to-result process, as experimentally demonstrated in 8.2.2.

Moreover, as we discuss in 5.2.3, (stochastic) simulation instances could exhibit highly *unbalanced* behaviours – in particular of multi-stable systems – thus proceeding at very different execution speeds. But the filtering of the results cannot start until each simulation instance has produced at least one data point. This is a "pitfall" which actually invalidates any attempt of minimising the transient storage of data –thus the reducing either shared-memory or I/O traffic – by simply putting the simulation and analysis stages in pipeline.

Finally, with respect to aspect 5, the design of efficient simulator tools targeting

multicore (and multicore cluster) platforms is even more challenging, since the outlined workflow actually becomes an I/O-bound problem, as we discuss in 5.1.1.

The very same approach has been used also in recent efforts exploiting GPGPUs [86]. The ever-increasing size of produced data makes this approach no longer viable.

4.2.3 Engineering drawbacks

In the previous scenario, the design of the simulator is often specifically optimised for a specific parallel platform, either multi-core or distributed (or not optimised at all). This reduce dramatically the portability of code from both engineering and performance viewpoints. In such a scenario porting a tool to a different architecture amounts to re-design the whole low-level parallel infrastructure of the tool (i.e. communication and synchronisation between computing elements), thus rethinking the exploitation of parallelism - which is the basis of performance optimisation – on the new platform.

5 Parallel Simulations on Multicore Platforms

The key vision we advocate to overcome the issues outlined in Sec. 4.2, is based on considering the optimisation of the whole simulation/analysis workflow, regarding data analysis as an on-line process working on streams of data resulting from the on-going simulations. Supporting efficient on-line data analysis has non-trivial effects on the design of the simulation stage. Moreover, in this approach both the simulation and the analysis stages should work on (high-frequency) streams, and require efficient data dependencies management (both on distributed and shared-memory systems). While running multiple instances of a Monte Carlo simulation "in insulation" is actually an embarrassingly parallel process, running them in a way that supports efficient on-line analysis of large (possibly big) data is not.

In this chapter, we use the CWC calculus (Chapter 3) and its (sequential) simulator core (5.2.1) as paradigmatic example to discuss the key features required to derive an easy porting on multicore platforms (Sec. 5.1). In particular we will argument on the parallelisation of a single simulation instance, many independent instances, and the technical challenges they require (5.1.1). In particular we will exploit the stream oriented pattern-based parallel programming supported by the FastFlow framework (2.1.4).

The key features discussed in Sec. 5.1 are turned into a family of solutions to speed up both the single simulation instance and many independent instances, providing at the same time the support for on-line data analysis. The former issue is approached using SIMD hardware accelerators (5.2.2), the latter advocating a novel simulation schema based on FastFlow accelerator that guarantees both easy development and efficient execution (5.2.3).

5.1 Exploiting Parallelism in Simulations

Gillespie algorithm realizes a Monte Carlo type simulation method, thus it relies on repeated random sampling to compute the result. An individual simulation, which tracks the state of the system at each time-step, is called a *trajectory*. Many thousands of trajectories might be needed to get a representative picture of how the system behaves on the whole.

For this, stochastic simulations are computationally more expensive than ODEs numerical unfolding. This balance is well-known and it motivated many attempts to speed up their execution time along last two decades [65]. They can be roughly categorised in attempts that tackle the speeding up of a *single simulation* and a *bulk of independent simulations*. In the following these (not mutually exclusive) approaches are discussed

under the viewpoint of parallel computing techniques and their exploitation on commodity multicore platforms. This discussion is not intended to be an encyclopaedic review of other techniques that can be used to achieve the same aim, such as ones related to the approximation of the simulation results, such as τ-leaping and hybrid techniques [21].

5.1.1 What can be accelerated? Where parallelism can be found?

Speeding up a single simulation

Parallelizing a single Gillespie-like stochastic simulation, i.e. the derivation of a simulation trajectory, is intrinsically hard. Unless introducing algorithmic relaxations – which correctness should be proved and typically lead to approximate simulation results – two successive Monte Carlo steps of the same simulation instance cannot be concurrently executed since there exists a strict data dependency between the two steps. Also, at the single step grain, speculative execution is unfeasible because of the excessive branching of possible future execution paths. As result, the only viable option to exploit parallel computing within a single simulation consists in parallelizing the single Monte Carlo step. Here, the available concurrency could be determined via data dependency analysis that can be made for any given specific simulator code (see Sec. 5.2). Typically, parallelism exploited at this level is extremely fine-grained since the longest concurrent execution path may at most count few machine instructions.

In this range, currently, no software mechanisms can support an effective inter-core or multi-processor parallelization: the overhead will easily overcome any benefit; the only viable option is hardware parallelism within a single core. Since, typically, instruction stream parallelism is already exploited by superscalar processor architecture, the only additional parallelization opportunity has to be searched in data parallelism to be exploited via a hardware accelerator, such as internal SSE or external GPGPU accelerators (see 2.1.1). In both cases, the simulator code should be deeply re-designed in a contiguous sequence of SIMD instructions. As we shall see in Sec. 5.2, this generally may lead to very modest advantages with respect to the required effort.

Speeding up independent simulation instances

The intrinsic complexity in the parallelization of the single step has traditionally led to the exploitation of parallelism in the computation of independent instances of the same simulation, which should anyway be computed to achieve statistical convergence of simulated trajectories (as in all Monte Carlo methods). The problem is well understood; it has been exploited in the last two decades in many different flavours and distributed computing environments, from clusters to grid to clouds, as discussed in Sec. 4.2.

Thanks to their independence, the different instances needed to simulate a biological model can be easily computed in an *embarrassingly parallel* fashion. However, the complete simulation workflow needed to derive simulation results includes additional phases, such as the dispatching and scheduling of simulations, result gathering, trajectory data assembling and analysis phases, which exhibit data dependencies (thus requires communication and/or synchronizations). Often, to simplify the design of the simulation tool,

these phases are neither parallelized nor considered in the performance evaluation. These phases, often (questionably) considered as pre-processing and post-processing phases, may result as expensive as the simulation itself.

The "embarrassingly parallel" simplification has negligible effect under the assumption that output data has a negligible size, but this is not likely to happen in this and next generation of biological simulations. As an example, a simulation of the HIV diffusion problem (computed using the StochKit toolkit for 4 years of simulation time) may easily produce over 5 GBytes of raw data per instance [2]. As clear, the data size is n-folded when n instances are considered. Eventually, this data should be gathered and often reduced to a single trajectory via statistical methods or analyzed with data mining methods, that can be much more time expensive to be figured out than bare statistical estimators.

These potential performance flaws are further exacerbated in multi-core and many-core platforms. These platforms do not exhibit the same degree of replication of hardware resources that can be found in distributed environments, and even independent processes actually compete for the same hardware resources within the single platform, such main and secondary memory, which performances represent the real challenge of forthcoming parallel programming models (a.k.a. *memory wall* problem). While simulation is substantially a CPU-bound problem on distributed platform, it may become mainly an I/O-bound problem on a multicore platform due to the need to store and post-process many trajectories. In particular, multi-stable simulations may require very fine grain resolution to discriminate trajectory state changes, and as it is clear, the finer the observed simulation time-step the strongest the computational problem is characterized as I/O-bound.

5.1.2 How to parallelize? A list of guidelines

In the previous section we discussed where parallelism can be found in Gillespie-like algorithms; the question that naturally follows is *how* this parallelism can be *effectively* exploited. We advocate here a number of parallelization issues that, we believe, can be used as pragmatic "guidelines" for the efficient parallelization of this kind of algorithms on multicore. Observe that, in principle, they are quite independent of the source of parallelism; however, they focus on inter-core parallelism, thus cannot be expected to be applied to other kinds of parallelism (e.g. SIMD parallelism). They will be then used along Sec. 5.2 as "instruments" to evaluate the quality of the parallelization work for the execution of independent instances of the CWC simulator.

Data stream as a first-class concept

The *in silico* (as well as *in vitro*) analysis of biological systems produces a huge amount of data. Often, they can be conveniently represented as data streams since they sequentially flows out from one or more hardware or software devices (e.g. simulators); often the cost of full storage of these streams overcomes their utility, as in many cases only a statistical filtering of the data is needed. These data streams can be conveniently represented

as *first-class concept*; their management should be performed *on-line* by exploiting the potentiality of underlying multicore platforms via suitable *high-level programming tools*.

Effective, high-level programming tools

To date, parallel programming has not embraced much more than low-level communication and synchronisation libraries. In the hierarchy of abstractions, it is only slightly above toggling absolute binary in the front panel of the machine (see 2.1.2). We believe that, among many, one of the reasons for such failure is the fact that programming multicore is still perceived as a branch of high-performance computing with the consequent excessive focus on absolute performance measures. By definition, the raison d'être for high-performance computing is high performance, but MIPS, FLOPS and speedup need not be the only measure. Human productivity, total cost and time to solution are equally, if not more, important. The shift to multicore is required to be graceful in the short term: existing applications should be ported to multicore systems with moderate effort. This is particularly important when parallel computing serves as tools for other sciences since non expert designer should be able to experiment different algorithmic solutions for both simulations and data analysis. This latter point, in particular, may require data synchronisation and could represent a very critical design point for both correctness and performance.

Cache-friendly synchronization for data streams

Current commodity multicore and many-core platforms exhibit a cache-coherent shared memory since it can effectively reduce the programming complexity of parallel programs (whereas different architectures, such as IBM Cell, have exhibited their major limits in programming complexity). Cache coherency is not for free, however. It largely affects synchronisations cost and may require expensive performance tuning (see 2.1.1). This is both an opportunity and a challenge for parallel programming framework designers since a properly designed framework should support the application with easy exploitation of parallelism (either design from scratch or porting from sequential code) and high-performance.

Load balancing of irregular workloads

Stochastic processes exhibit an irregular behavior in space and time by their very nature since different simulations may cover the same simulation time span following many different, randomly chosen, paths and number of iterations. Therefore, parallelization tools should support the dynamic and active balancing of workload across the involved cores.

5.2 The CWC Simulator Testbed

The proposed guidelines are validated using the CWC Simulator as running example. It has been developed as a plain C++ sequential code, then it has been parallelised

for multicore, in order to evaluate the effectiveness of the methodology also in term of development effort.

In the parallelisation two main frameworks have been used: the GCC compiler SSE intrinsics [75] to speed up a single simulation, and the FastFlow parallel programming framework (2.1.4) to speed up independent simulation instances, which provides the basic facilities described in 5.1.1.

5.2.1 The CWC Simulator core

The CWC Simulator core basically implements the Gillespie SSA over CWC systems. Thus, given a term and a set of rules, it iterates the following logical steps until the end-of-simulation time is reached:

1. (*Match*) it searches all the occurrences of the rules matching in some compartment or wrap of the term. Then it associates a stochastic rate to each match. This step results into a weighted matchset.

2. (*Resolve*) it stochastically decides the *time* offset at which the next reaction will occur and the *rule* that will activate it. Moreover, since in CWC reactions can occur at different contexts, it consults the matchset in order to decide how portion of the system will react.

3. (*Update*) it actually applies the selected reaction, affecting both the system and the clock, moving forward the simulation process.

Each iteration of the steps above is called a *simulation step*. The pseudo-code of the simulation step is sketched in Fig. 5.1.

5.2.2 Speeding up a single simulation

As discussed in Sec. 5.1.1, the parallelisation of the single CWC simulation step is theoretically feasible via the SSE accelerator. In Fig. 5.1, the phases of the code that can be parallelised in SIMD fashion with moderate effort are marked with the *"SIMD"* label. The exploited parallelism degree is 4 since 4x32-bit operation has been used. The experimental results reported in Sec. 8.2.1 show that the achieved speedup is almost negligible because only a fraction of the whole simulation step has been actually parallelised. Similar parallelisation efforts conducted on GPGPU accelerators, which exploit a much larger potential SIMD parallelism, do not actually result in satisfactory results. As an example, see the parallelisation of Gillespie's first reaction method on NVIDIA CUDA [61].

Unfortunately, the extension of the SIMD parallelism to larger fractions of the code may require a very high coding effort since the redesign of the original code is required. As an example recursive patterns (used for tree-matching, marked with *"non-SIMD"* parallelism in Fig. 5.1) are not easily manageable using SIMD parallelism and should be differently coded before being parallelised. Observe that these recursive kernels cannot either be parallelised across cores because they are excessively fine-grained; as a

```
1 Simulation_Step {
2 // 1. Match
3 foreach r ∈ ruleset {
4   Match(r, T, TOP_LEVEL); // [non−SIMD parallelism]
5 }
6
7 // 2. Resolve (Monte Carlo)
8 (tau, mu) = Gillespie(matchset);
9 context = stochastic_choice on matchset[mu];
10
11 // 3. Update
12 (P,O) = left_and_right_side (mu);
13 delete P_sigma from T at context; // SIMD
14 put deleted_elements in sigma;
15 add O_sigma to T at context; // SIMD
16 simclock += tau;
17 }
```

Figure 5.1: CWC Simulator pseudo-code with sources of fine-grain parallelism

qualitative example, the parallelisation via POSIX threads (tested with FastFlow and Intel TBB) is, on the reference multicore platform we use in 8.2.2, from 10 to 100 times slower with respect to sequential version due to synchronisation overheads (i.e. cache coherence, cache misses, etc.).

All in all, intra-core SIMD parallelism appears the only viable way to this kind of parallelisation. Observe however that it might require, for this class of algorithms, a coding effort that easily overcomes the potential benefits.

5.2.3 Speeding up independent simulation instances

Starting from the CWC sequential simulator code, we here advocate a parallelisation schema supporting the parallel execution of many self-balancing simulation instances on multicore. Its design aims to address all the issues discussed in 5.1.1: it is realised by means of the FastFlow framework (2.1.4) that natively supports high-level parallel programming patterns working on data streams and it exhibits an efficient lock-free run-time support that can be integrated with SIMD code. It therefore makes it possible the easy porting of the sequential CWC code on multicore for the execution of multiple simulation instances (either replicas or the parameter sweeping of a simulation), and the on-line synthesis of their trajectories.

For the sake of simplicity, here we regard the reduction phase of the trajectories as a monolithic process, which performs buffering, alignment and synthesis by means of some reduction function. We present a more detailed view of the reduction phase in Chapter 6.

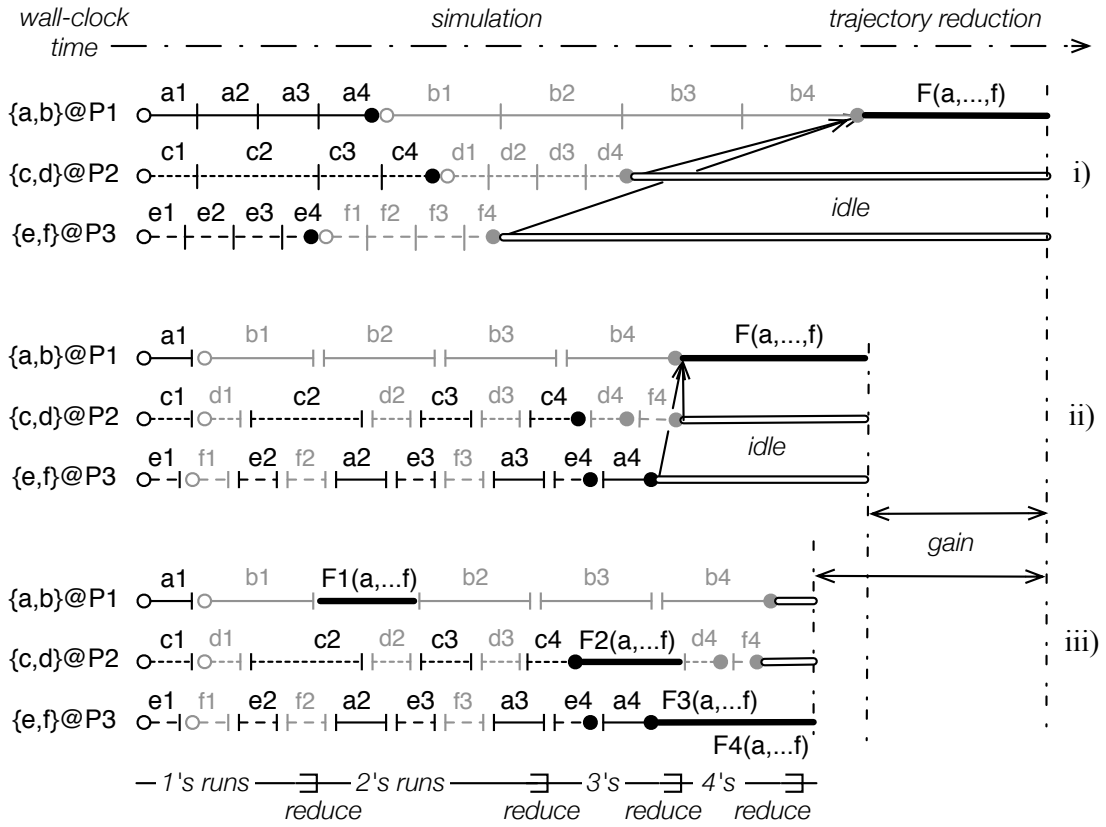


Figure 5.2: Three alternative parallelisation schemas exemplified on 6 simulation instances and 3 processors. i) Round-robin execution of simulations followed by a reduction phase. ii) Auto-balancing schema with time-slicing at constant simulation time (variable wall-clock time) followed by a reduction phase. iii) Previous schema with on-line pipelined reduction.

The schema supports three main behaviours, which are exemplified in Fig. 5.2:

i) The different simulation instances (called a,b,c,d,e,f) are dispatched for the execution on different workers threads of a FastFlow farm, which run on different cores; a worker sequentially runs all the simulations it received. The dispatching of instances to workers could be either performed before the execution according to some static policy (e.g. Round-Robin) or via an on-line scheduling policy (e.g. on-demand). Workers stream out the trajectories, which are sampled at fixed time steps along simulation time. Streams are buffered in the farm collector and then reduced in a single stream according to one or more functions (e.g. F). Observe that the constant sampling assumption simplifies the reduction process even if it is not strictly required since data could be on-line re-aligned during the buffering [2]. Also notice that since simulation time advances according to a random variable, different instances advance at different wall-clock time rates. The phenomenon is highlighted in Fig. 5.2-i splitting each instance in four equal fractions of the simulation time (e.g. ⟨a1, a2, a3, a4⟩, ⟨b1, b2, b3, b4⟩), which exhibit different wall-clock times to be computed (segment length). This may induce even a significant load unbalance that could be only partially addressed using on-line scheduling policies.

ii) A possible solution to improve load balancing of the schema consists in coupling the on-line scheduling policy with the reduction of execution time-slice that is subject to the scheduling policy. At this end, each simulation instance can be represented as an object that incorporate its current progress and provide the scheduler with the possibility of stopping and restarting an instance. In this way, as it happens in a time-sharing operating system, (fixed or variable length) slices of an instance can be scheduled on different workers provided slices of the same instances are sequenced (possibly on different workers). Thanks to cache-coherent shared memory the scheduling can be efficiently realized via pointer management. The idea is exemplified in Fig. 5.2-ii. Also, scheduling and dispatching to workers can be equipped with predictive heuristics based on instance history in order to characterize the relative speed of the simulation instances.

iii) The previous schema can be further improved by pipelining the reduction phase that is performed on-line. Since instance time-slicing can make all the instances to progress, a running window of all the trajectories can be reduced while they are still being produced. The reduction process, which is logically made within a separate computing element (i.e. the farm collector thread in this simplified schema), can be either run on an additional processor or interleaved with the execution of simulation instances (see Fig. 5.2-iii). The solution also significantly reduces the amount of data to be kept in memory because:

   - thanks to interleaving all the trajectories advances almost aligned with respect to simulation time;
   - the already reduced parts of the trajectories can be deleted from main memory (and stored in secondary memory if needed). We call *selective memory* the data structure implementing this solution.

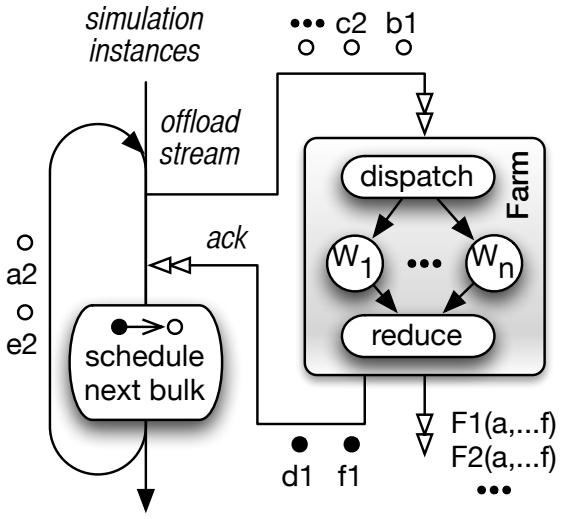


Figure 5.3: Architecture of the FastFlow-based CWC parallel simulations

The three schemas can be effectively implemented using FastFlow as sketched in Fig. 5.3. In particular, the FastFlow *farm accelerator* feature [13] fits well the previous design since it makes possible to offload a stream of object pointers onto a farm of workers, each of them running a CWC simulation engine, and to implement user-defined dispatching and reduction functions via standard Object Oriented sub-classing. FastFlow natively provides the programmer with streams, a configurable farm pattern, and an efficient run-time support based on lock-free synchronisations. All these features effectively made it possible to port the CWC sequential simulator to multicore with moderate effort. In addition, the complexity of the achieved solution can be gracefully improved by successive refinements in order to test different scheduling policies or variants to the basic schema. In this regard the accelerator feature represents a key issue since it enables the programmer to make very local changes to the original code that in first approximation consists in changing a method call into the offload of the same method.

We present the actual realisation of the simulation stage of the CWC Simulator worflow in 6.2.1.

6 On-line Analysis of Irregular Streamed Data

Most biological data from dynamical kinetics of species might require further processing with statistical or mining tools to be really functional to biological scientists. In particular, the bulk of trajectories coming from Monte Carlo simulators can exhibit a natural unevenness due to the stochastic nature of the tool and are typically represented with many and large data series. This unevenness, in the form of deviant trajectories, high variance of results and multi-stable behaviours, often represents the real nature of the phenomena that is not captured by traditional approaches, such as ODEs.

In this chapter, we introduce the problem of rich on-line analysis of simulation data and the issues posed by dealing with large (even big) data streams (Sec. 6.1). We discuss the realisation of an efficient parallel analysis stage, featuring multiple statistical and mining methods on streamed simulation outcomes (6.2.2). In this context, we provide a coarse categorization of biological systems targeted by the provided filters, namely mono-stable, bi-stable/multi-stable and oscillatory systems.

Finally we return to the main topic of this work, building up the whole CWC Simulator workflow for multicore platforms (Sec. 6.2).

6.1 Effective Analysis of Stochastic Simulation Outcomes

To be effective, stochastic methods in systems biology require many trajectories with a fine grain resolution in order to make observable deviant trajectories, peaks, high variance of results and multi-stable behaviours, which often represent the real nature of the phenomena that is not well captured by traditional approaches, such as ODEs. These events are often not immediate to detect in the bulk of gross simulation results. Several techniques for analysing such data, e.g. principal components analysis, linear modelling and canonical correlation analysis have been proposed. It can be imagined that next generation software tools for natural sciences should be able to perform this kind of processing in pipeline with the running data source, as a partially or totally on-line process because:

- it will be needed to manage an ever increasing amount of experimental data, either coming from measurement or simulation;
- it will substantially improve the overall experimental workflow by providing the natural scientists with an almost real-time feedback, enabling the early tuning or sweeping of the experimental parameters, and thus scientific productivity.

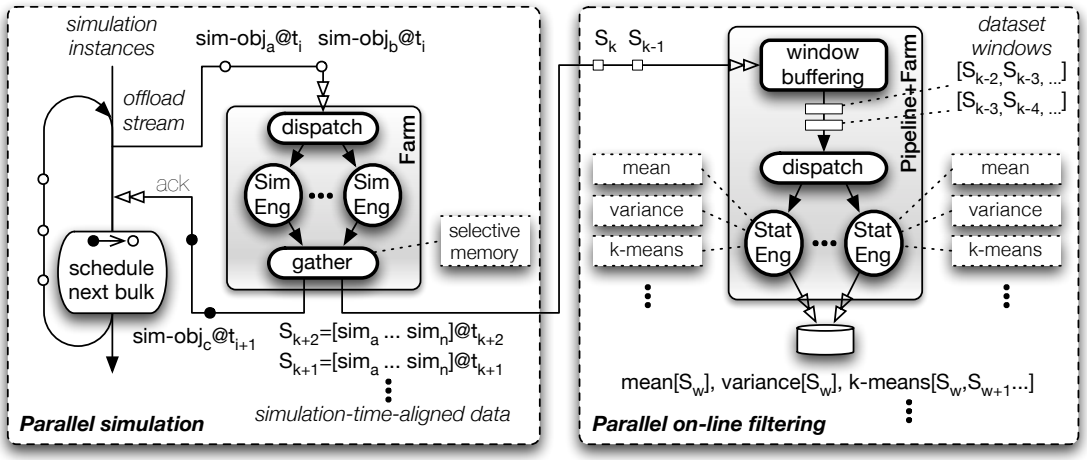


Figure 6.1: The (pipelined) interaction between simulation and analysis stages

Moreover, the flexibility given by the possibility of running many different analysis modules in parallel is of particular interest, as in many biological case studies the searched pattern in experimental results is unknown and might require to try different kinds of analysis.

The parallel analysis of the system dynamics (e.g. along time) is more challenging since on-line data processing requires statistical and mining operators to work on streamed data and, in general, random access to data is guaranteed only within a limited window of the whole dataset, while already accessed data can be only stored in synthesised form.

When data description techniques, which require accessing the whole data set in random order, cannot be used, on-line data description and mining require novel algorithms. The extensive study of these algorithms is an emerging topic in data discovery community and is beyond the scope of this work, which focuses on the design of a parallel infrastructure featuring easy engineering of battery of filters. Nevertheless, the design we present, which is based on *sliding windows* over the data set, provides the support for approximated methods, which approximation degree depends on the width of the sliding windows .

6.2 The CWC Simulator Workflow

In this section we present the CWC Simulator workflow, which is composed of a three-stage pipeline: simulation stage (Sec. 6.2.1), analysis stage (Sec. 6.2.2) and display of results (Sec. 6.2.3), which is is realised by way of a Graphical User Interface (GUI). The simulation and analysis stages are pipelined (Fig. 6.1) in such a way that, at each observed simulation time t_i, the simulation stage streams out the partial results of all simulation trajectories (aligned at t_i) to the analysis stage that immediately produces a partial result. Analysis stage, which can be equipped with user-defined statistic and mining operators, works on sliding data windows and does not require to keep in memory

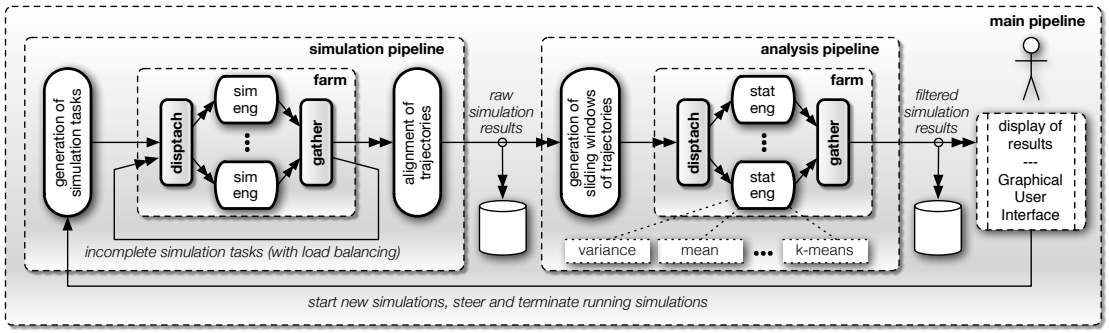


Figure 6.2: Architecture of the CWC Simulator workflow

the full data set with both performance and response time benefits.

Fig. 6.1 shows a detailed view of the interaction between simulation and analysis stages; for the sake of simplicity, in this (simplified) representation the alignment of the trajectories is assumed to be performed by the gather node, while in the actual realisation of the tool this task is performed by a dedicated stage.

The big picture of the whole workflow is shown in Fig. 6.2. The easiness of engineering is guaranteed by the high level of abstraction provided by FastFlow: in the picture, all the grey boxes as well as all the code needed for synchronisation and data streaming (double-headed arrows) is automatically generated by the FastFlow framework. The implementation of the whole software actually consists in declaring the structure of the workflow in terms of FastFlow objects (i.e. farm and pipelines) and filling white boxes with sequential code.

6.2.1 The simulation pipeline

The design of the simulation pipeline has been discussed in 5.2.3. In particular, we consider the schema in Fig. 5.2-iii.

The pipeline is composed of three main parts: a *generation of simulation tasks* stage, a *farm of simulation engines* stage and a *alignment of trajectories* stage (Fig. 6.2).

The input of the simulation pipeline (either from GUI or from file) contains the model to be simulated and the parameters of the simulation. The output is a stream of arrays of simulation results. Each of these arrays holds a point for each of the trajectories of all (independent) simulations, aligned at a given simulation time. Actually, each array represents a snapshot (called "cut") at a given simulation time of the whole dataset of results (Fig. 6.1). This not necessarily represents the current status (at a given point in wall-clock time) of all running simulations. Stochastic simulations exhibit an irregular behaviour in space and time according to their nature, since different simulations may cover the same simulation timespan, following many different (randomly-chosen) paths, in a different number of iterations. Therefore, parallelisation tools should support the dynamic and active balancing of workload across the involved cores. This mainly motivates the structure of the simulation pipeline. The first stage generates a number of

independent simulation tasks, each of them wrapped in a C++ object. These objects are passed to the farm of simulation engines, which dispatch them (on-demand) to a number of simulation engines (*sim eng*). Each simulation engine brings forward a simulation that lasts a precise simulation time (simulation quantum). Then it reschedules back the operation along the feedback channel. Simulation results produced in this quantum are streamed towards the next stage which sorts out all received results and aligns them according to the simulation time. Once all simulation tasks overcome a given simulation time, an array of results is produced and streamed to the analysis pipeline.

In this process, the farm scheduler prioritises "slow" simulation tasks, in such a way that the front-line task proceeds as much aligned as possible to simulation time. This solves both the load balancing problem by keeping all simulation engines always busy and reduces to the minimum the transient storage of incomplete results, thus reducing the shared-memory traffic.

We remark that the generation of simulation tasks should guarantee the *statistical independence* of the instances. This aspect is discussed in Appendix A.

6.2.2 The analysis pipeline

The analysis pipeline is depicted in Fig. 6.1.

By design, each snapshot at a given simulation time of all simulation trajectories (i.e. an array of simulation results), can be analysed immediately and independently (thus concurrently) one each other. For example, the mean and variance (as well as other statistical estimators) can be immediately computed and streamed out to the display stage. More complex analyses, i.e. ones aimed to understand system dynamics, have further requirements. In the most general case, they require the access to the whole dataset. Unfortunately this can be hardly done with a fully on-line process. In many cases, however, it is possible to derive reasonable approximation of these analyses from a sliding window of the whole dataset (e.g. for trajectory clustering). For this, the stream incoming (from the selective memory) in the analysis pipeline is passed through a stage that creates a stream of (partially overlapping) sliding windows of trajectories cuts. Each sliding window can be eventually processed in parallel and therefore is dispatched to a farm of statistic engines. Results are collected and re-ordered (i.e. gathered) and streamed toward the user interface and the permanent storage.

The analysis pipeline is provided with three families of pre-defined estimators covering most common kind of statistical analysis. Current filters are aimed to analyse the following families of systems:

1. *Mono-stable systems (via statistical estimators).* Mean, standard deviation and other statistical estimators are typically used to evaluate, both qualitatively and quantitatively, the behaviour of stable systems and the reliability of the stochastic models used for their simulation (i.e. comparison with ODE results). Quantiles calculation is also often useful to approximate the distribution of simulation trajectories over time as it performs an histogram which summarises the involved quantities without the effects of long-tailored asymmetric distribution or outliers.

In fact, in those cases, descriptive statistics could not underline a central tendency.

2. *Multi-stable systems (via trajectory clustering).* The clustering of trajectories helps the analysis of biological systems exhibiting a multi-stable behaviour. Each cluster can automatically separate and distinguish different cases which can be eventually analysed by statistical estimators. Concentrations of a given instant from all simulations are numerically filtered from stochastic noise and the global trends are extrapolated from clusters. In this work we employed two clustering techniques: K-means [71] and Quality Threshold (QT) [74] clustering. The clustering procedure collects the filtered data contained into a sliding time window Δ_W centered in the current data point $x_i \equiv f(t_i)$ where $t_i \equiv t_0 + i\Delta_S$ (where Δ_S is a constant sampling time) for all simulation trajectories and the extrapolated forecast point x_i^E, referred to the local trend, using the information of the Savitzky-Golay filter [112], i.e. a low-pass filter suitable for data smoothing. The main idea underneath Savitzky-Golay filtering is to find filter coefficients c_n that preserve higher moments, i.e. to approximate the underlying function within the moving window not by a constant (whose estimate is the average), but by a polynomial of higher order. This scheme also allows the computation of numerical derivatives considering the coefficient of the derived polynomial.

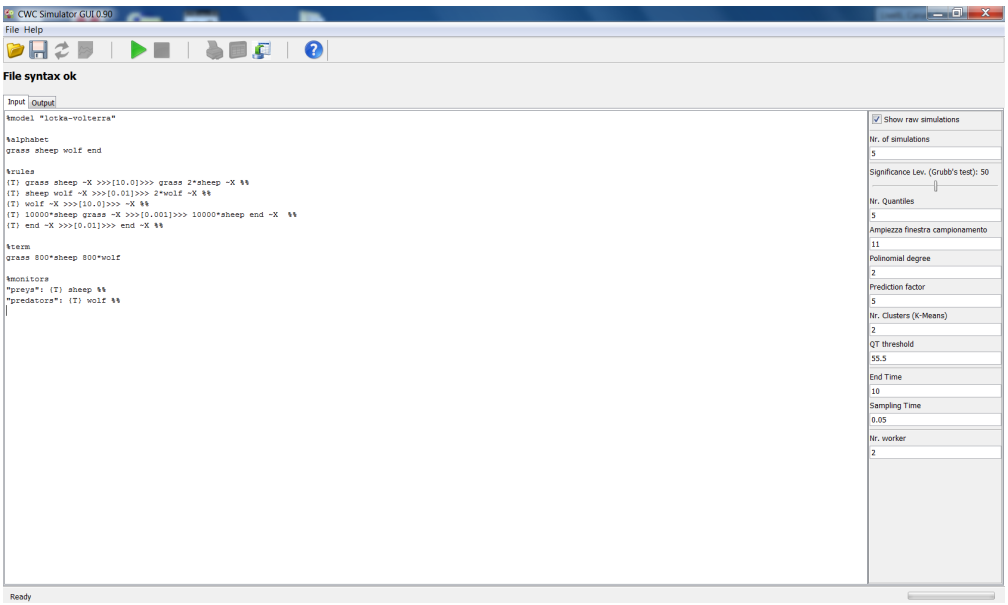
3. *Periodic phenomena (via peak detection).* Many processes in living organisms are oscillatory. For these kind of systems the analysis must be focused on the recurrence of phenomena, for instance concentration spikes or peaks of biological quantities, which also make it possible to determine the frequency of occurrence of a given phenomenon. The peak detection is basically performed by way of the analysis of the local maximum in a continuous curve, which is in turn detected through the analysis of the derivatives of the curve estimated by the Savitzky-Golay filter. From the period between successive peaks, the frequency of the related event is then inferred.

The effectiveness of such filters will be discussed in Sec. 8.1.

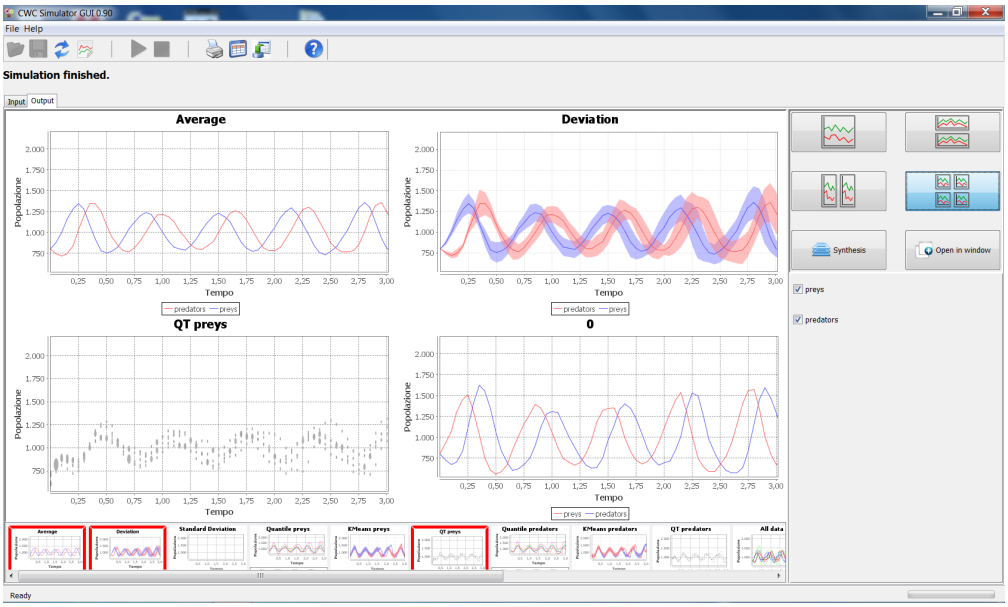
We remark that, thanks to modular design of the pipeline, additional filtering functions can be easily plugged in by simply extending the list of statistics with additional sequential or parallel functions (i.e. adding a function pointer to that list).

6.2.3 The Graphical User Interface

The CWC Simulator workflow is wrapped in a back-end tool that can be steered either via a command line tool or a graphical user interface, which make it possible to design the biological model, to run simulations and analysis and to view partial results during the run. Also, the front-end makes it possible to control the simulation workflow from a remote machine. Two screenshots of the graphical front-end are reported in Fig. 6.3.



(a) Specification of the CWC model



(b) Visualisation of real-time results of multiple analysis engines

Figure 6.3: Screenshots of the CWC Simulator GUI

7 Simulation/Analysis Workflow in the Cloud

Public IaaS cloud platforms (which can be regarded as *virtual* multicore clusters) are attractive for HPC bioinformatics since they provide virtually unlimited computational resources on-demand. Moreover, multicore clusters (which can be regarded as private clouds), are increasingly diffused in scientific laboratories. Thus, a well engineered tool should be able to adapt on such heterogeneous platforms (which are informally referred to as "Clouds"), with respect to both performance portability and low porting effort.

In this chapter, we show how our approach fits on such increasingly popular distributed platforms for HPC bioinformatics (7.1).

Then we present the distributed CWC Simulator workflow (7.2), developed on top of FastFlow framework as a very low-effort extension of the multicore variant presented in Chapter 6.

7.1 The Workflow on Modern Distributed Platforms

Clouds are increasingly diffused in HPC scientific laboratories. They make available on-demand and on a pay-per-use basis an elastic parallel computing platform that can be customised with a specific set of tools such as simulators for systems biology. Cloud "elasticity" enables the users to deploy the same application on a virtualised parallel platform of configurable type, size and computational power. The typical platform can be abstracted as a virtual cluster of shared-memory multicore platforms (see 2.1.1). Once deployed, the virtualised platform is immediately ready to compute and can be interactively used by the end user.

The potentiality of such platforms, however, can be fully exploited only if the running software (e.g. simulation tool) exhibits a similar flexibility and interactivity:

1. The application should benefit from both levels of parallelism available in a (virtualised) multicore (and many-core if available) cluster, hopefully providing the end user with performance scalability with respect to both levels.

2. The programming model should manage parallelism as a first-class concept to make the tools (e.g. simulator) easy to design, develop and extend; the programming model should be able to capture parallelism at all levels, i.e. distributed platforms, multicore and many-core, and possibly it should be able to support the seamless porting of application across the described platforms with performance portability.

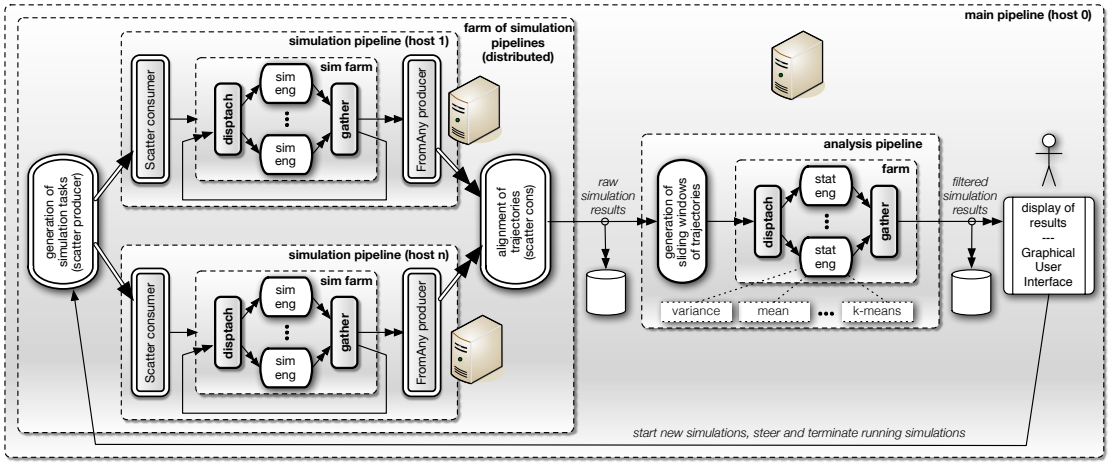


Figure 7.1: Architecture of the distributed CWC Simulator. The distributed workflow is obtained by replicating the simulation pipeline of the multicore version in Fig. 6.2 via the farm pattern.

3. The software tools itself should be designed to be reactive and interactive in order to be dynamically steered by bioinformatics scientists.

To the best of our knowledge, many related works (2.2.5) cover some of these aspects, but none of them addresses all three aspects at the same time.

We claim our approach targets all these issues. We designed the workflow on top of the high-level FastFlow programming framework, which guarantees a very low-effort porting to (and between) distributed platforms (see 7.2); the exploitation of parallelism on different platforms is almost entirely in charge of the parallel programming methodology provided by the FastFlow programming framework, which natively targets multicores, many-cores and cluster platforms, exhibiting performances at the top of the state-of-the-art, thus providing performance portability; finally, the on-line data analysis allows to offload the whole workflow in the cloud as a single parallel pipeline with no storage of intermediate results on virtualised storage, thus improving interactivity, as we discussed in Chapter 6.

We remark that the ability of managing the whole data set in the cloud, minimizing (possibly avoiding) large data movements, has been widely recognized as necessary for next generation bioinformatics (see 2.2.5).

7.2 The Distributed CWC Simulator

Thanks to the high-level design of the CWC simulator and the extension of the FastFlow to distributed platforms [6], the porting of the CWC simulator to distributed platforms has been possible with a very limited coding effort.

The architecture of the distributed CWC Simulator is shown in Fig. 7.1. It is clear that the design directly derives from the existing multi-core version, shown in Fig. 6.2,

where the simulation pipeline has been substituted with a farm of simulation pipelines. Each of them can be run on a different platform: it receives simulation parameters from the *generation of simulation tasks* node, and feeds the *alignment of trajectories* node with a stream of results. These nodes have been upgraded from `ff_node`s to `ff_dnode`s to support network channels. Also, the simulation pipeline is extended with two fringe `ff_dnode`s to implement de-serialising and serialising activities without touching existing code. Notice that they are not strictly needed since their functionality can be also embedded in the farm's *dispatch* and *gather* nodes, but this requires to subclass them, thus altering rather than reusing previous version. Observe that the two additional nodes do not bring service-time penalties, since all data exchanges between them and the dispatch/gather process happen via pointer passing, and they are working in pipeline with the farm. Moreover, they can be used to tune communication grain by coalescing successive items in the result streams, e.g. to tune the workflow against the network latency.

Fig. 7.1 shows the semi-automatic porting provided by high-level parallel programming approach, namely solid boxes are `ff_node`s, double-stroked boxes are `ff_dnode`s, and dashed boxes are FastFlow patterns; double-headed arrows are streams; grey boxes are automatically generated by the FastFlow framework, while white boxes hold user-defined code which is the same as the multicore version one.

If needed, also the analysis pipeline can be made distributed, following exactly the same schema.

7.2.1 The distributed CWC Simulator on public cloud

Since the typical IaaS cloud platform can be abstracted as a virtual cluster of shared-memory multicore platforms, the distributed version of the CWC Simulator workflow naturally fits on such kind of platform. The overall architecture of the distributed CWC Simulator workflow deployed on cloud is reported in Fig. 7.2. Note that the communication between simulation and analysis stages does not require storage of intermediate results on virtualised storage.

In Fig. 7.2, the FastFlow framework automatically generates the implementation of patterns connecting `ff_node`s and `ff_dnode`s with streams, which are implemented either in the shared-memory model within the single virtual machine or in the message-passing model across virtual machines.

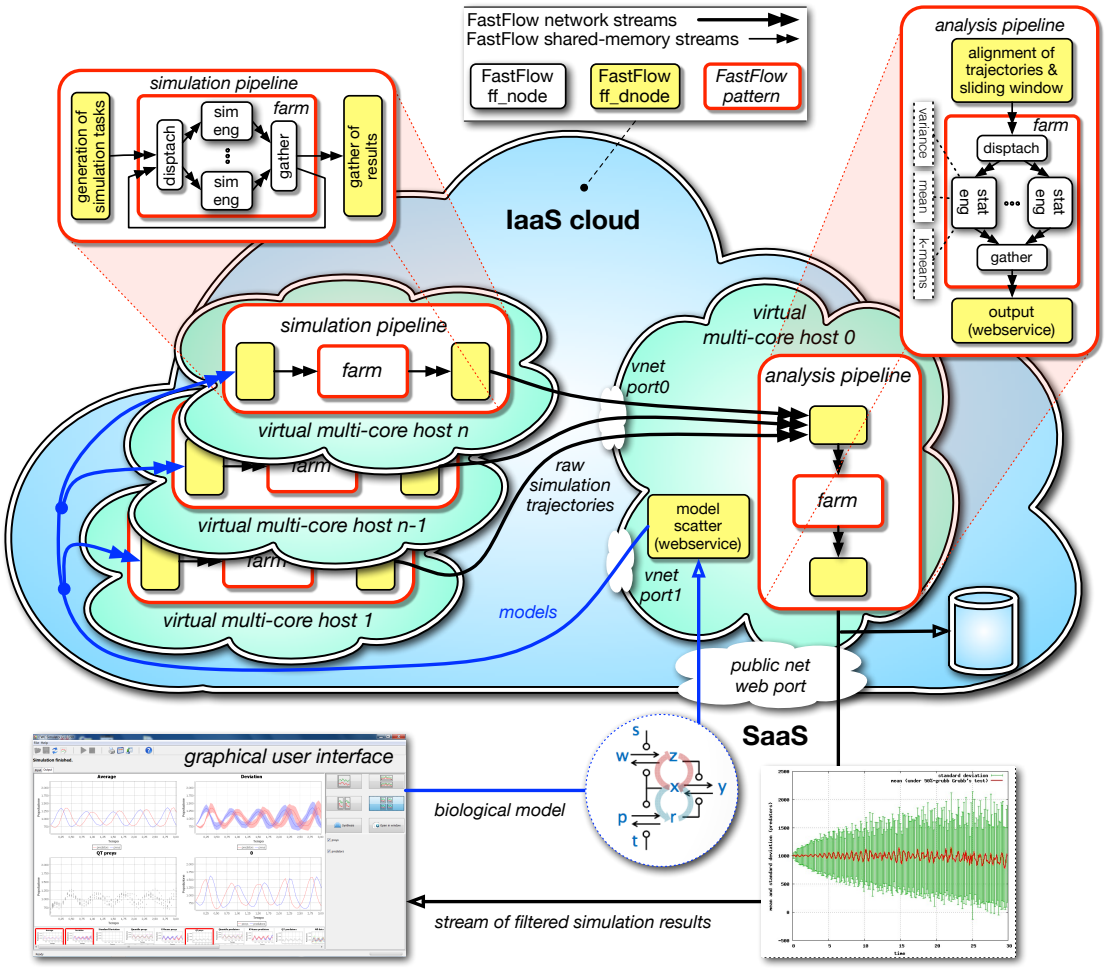


Figure 7.2: Architecture of the distributed CWC Simulator on the cloud.

8 Experimental Evaluation

In this chapter we evaluate experimentally the validity of our approach. Experiments follow two main lines: the effectiveness of the proposed on-line analysis to capture the behaviour of biological systems (Sec. 8.1) and the efficiency of the tool in executing the workflow on different platforms (Sec. 8.2).

For the former purpose, paradigmatic examples of three classes of biological systems are discussed, i.e. mono-stable (8.1.1), bi-stable/multi-stable (8.1.2, 8.1.3), and oscillatory (8.1.4); the key behaviour of the systems in these classes is studied by way of the three classes of on-line analysis tools, namely statistical description, clustering, and frequency detection.

For the latter purpose, we evaluate the efficiency, especially in terms of speedup, taking into account the performance on both multicores (8.2.2) and multicore clusters (8.2.3); we also present some preliminary experiments conducted on public cloud platforms (8.2.4).

8.1 Expressivity and Effectiveness

In order to evaluate the effectiveness of the proposed approach, paradigmatic examples of three classes of biological systems are discussed, i.e. mono-stable, bi-stable/multi-stable, and oscillatory systems. The key behaviour of the systems in these classes is studied by way of the three classes of on-line analysis tools introduced in 6.2.2, respectively, statistical description, clustering, and frequency detection.

In the first three examples, all biochemical reactions are assumed to follow the mass action law. In this case the reaction rules are decorated simply with the corresponding kinetic constants.

8.1.1 Mono-stable systems

Consider a simple biological model that specifies ammonium transport from external environment into the cells of *Escherichia Coli* based on [87, 31]. In defining this model some CWC specific features, although in a simplified form, are used.

The transport of ammonia/ammonium (Am) is fundamental to nitrogen metabolism. *Escherichia Coli* has the ability to grow at Am concentrations by expressing an Am transport protein AmtB. AmtB functions as a channel protein that binds ${NH_4}^+$ at the entrance gate of the channel, deprotonates it and conducts the product NH_3 into the cytoplasm. In addition to the above mentioned AmtB mediated transport, the bidirectional free diffusion of the uncharged ammonia through the membrane is also included in the model.

$$
\begin{array}{ll}
(rDif_1) & \top : NH_3(x \rfloor X)^{cell} \overset{1.4}{\longmapsto} NH_3(x \rfloor NH_3\ X)^{cell} \\
(rDif_2) & cell : NH_3 \overset{1.4}{\longmapsto} \bullet \\
(rAT_1) & \top : {NH_4}^+(AmtB\ x \rfloor X)^{cell} \overset{50000}{\longmapsto} (AmtB\text{-}{NH_4}^+\ x \rfloor X)^{cell} \\
(rAT_2) & \top : (AmtB\text{-}{NH_4}^+\ x \rfloor X)^{cell} \overset{0.5}{\longmapsto} {NH_4}^+(AmtB\ x \rfloor X)^{cell} \\
(rAT_3) & \top : (AmtB\text{-}{NH_4}^+\ x \rfloor X)^{cell} \overset{0.005}{\longmapsto} H(AmtB\text{-}NH_3\ x \rfloor X)^{cell} \\
(rAT_4) & \top : (AmtB\text{-}NH_3\ x \rfloor X)^{cell} \overset{0.005}{\longmapsto} H(AmtB\ x \rfloor NH_3\ X)^{cell} \\
(rN_1) & cell : NH_3 \overset{3000}{\longmapsto} {NH_4}^+ \\
(rN_2) & cell : {NH_4}^+ \overset{56.2}{\longmapsto} NH_3 \\
(rGS) & cell : {NH_4}^+ \overset{0.008}{\longmapsto} \bullet
\end{array}
$$

Figure 8.1: CWC rules for the *Escherichia Coli* dynamics

The *E. Coli* cell has $AmtB$ molecules on the wrap. We do not consider the dynamics of the external ammonium form (NH_3) keeping it constant in the rules ($rDif_1$) and ($rDif_2$). The set of rewrite rules modelling the kinetics is given in Fig. 8.1.

This model is useful to analyse how the setting of the model parameters affects the maximal concentration level of NH_3 and ${NH_4}^+$ inside the cell reachable from given initial conditions.

The performed simulations concern the computation of the first two moments of the species NH_3 inside the cell using the on-line statistics based upon 100 simulations running for 200 time units using a sampling time $\Delta_S = 1$ time unit. The starting term was:

$$30NH_3\ 5000{NH_4}^+\ (50AmtB \rfloor)^{cell}.$$

Fig. 8.2 shows the on-line computation of the mean and standard deviation for species NH_3 inside the cell compared with the solution of the corresponding deterministic simulation using ODEs.

Fig. 8.3 reports the analysis of convergence of the ammonium flow transported. Increasing the number of simulations, the confidence interval of 95% of the sample average reduces its amplitude.

These results about a mono-stable behaviour suggest that the mean follows closely the deterministic trajectory. Furthermore, fluctuations around the mean captured through the standard deviation calculation indicate the effect of the noise on the system. If this effect is severe, a deterministic approach to the analysis of such a system can be misleading and calls for a thorough stochastic treatment.

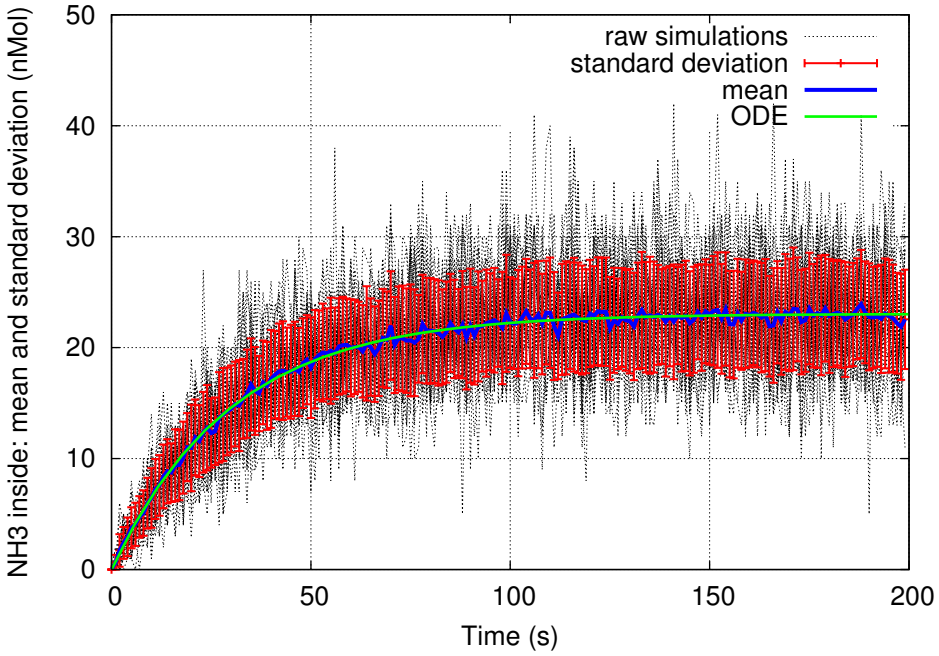


Figure 8.2: Mean and standard deviation on the *Escherichia Coli* model. The figure reports also the raw simulation trajectories.

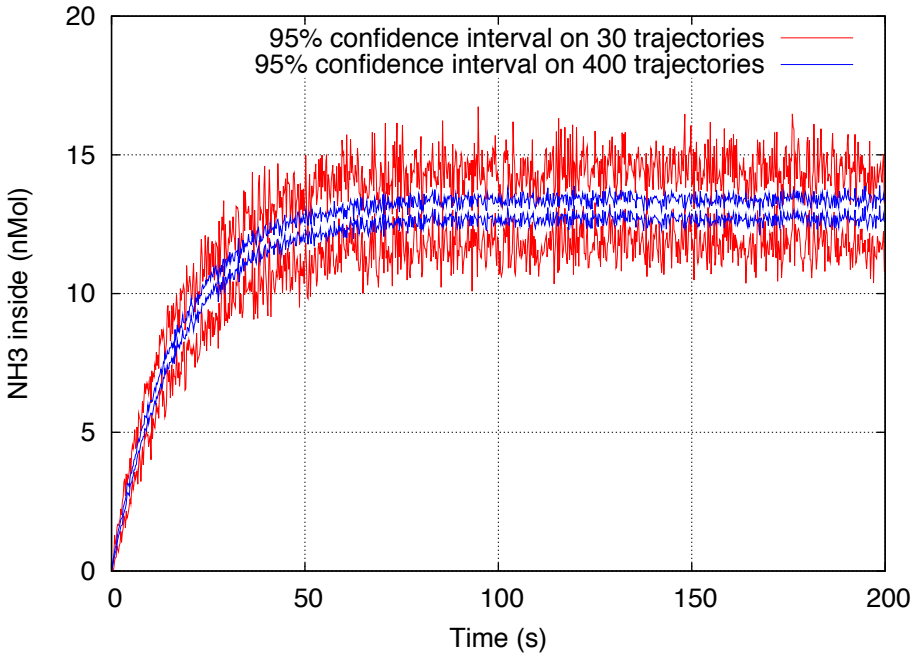


Figure 8.3: Convergence of the sample average on the *Escherichia Coli* model

8.1.2 Bi-stable biological systems

One of the most studied examples of bi-stabile model is the *Schlögl* model [115]. The simplicity of this network makes it an ideal prototype to show the effectiveness of the on-line clustering techniques on the filtered trajectories in the presence of bi-modality.

$$a\ a \overset{0.03}{\longmapsto} a\ a\ a$$
$$a\ a\ a \overset{0.0001}{\longmapsto} a\ a$$
$$b \overset{200}{\longmapsto} b\ a$$
$$a \overset{3.5}{\longmapsto} \bullet$$

Figure 8.4: CWC rules for the *Schlögl* dynamics

The set of CWC rules modelling this system is reported in Fig. 8.4 (in the following examples all reactions are assumed to take place in the external environment, thus we omit to explicitly write the compartment ($\top$) in which they take place).

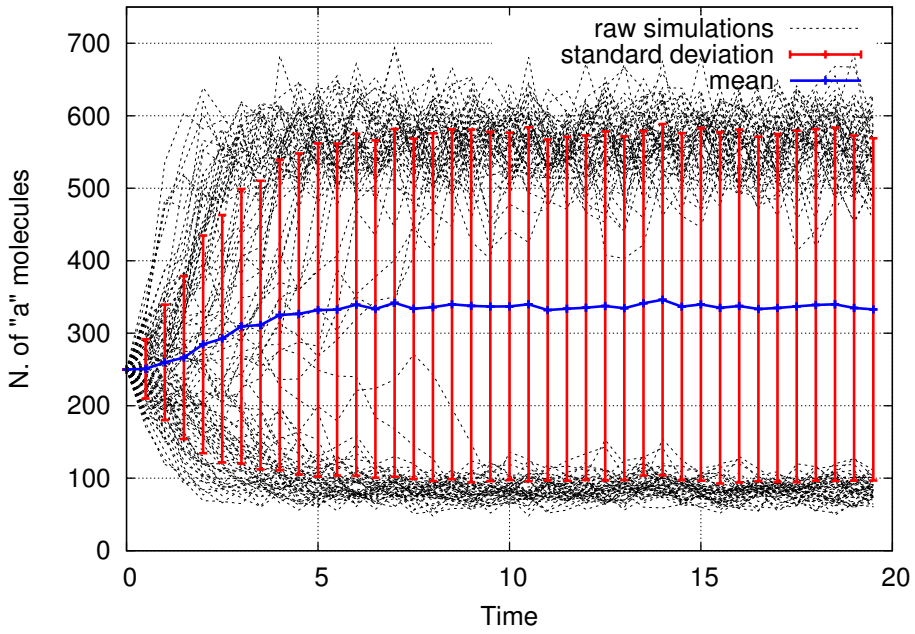
The number of molecules of the species b is kept constant (buffered) while, at equilibrium, the species a displays a noise-induced switching between the two stable steady states (see Fig. 8.5a). This case is paradigmatic to show that simple mean and standard deviation are not significant to summarise the overall behaviour and the mean is not representative of any simulation trajectory.

Fig. 8.5b shows the resulting clusters (colored circles) computed on-line using K-means on the *Schlögl* model for species a over 100 stochastic simulations starting with the term:

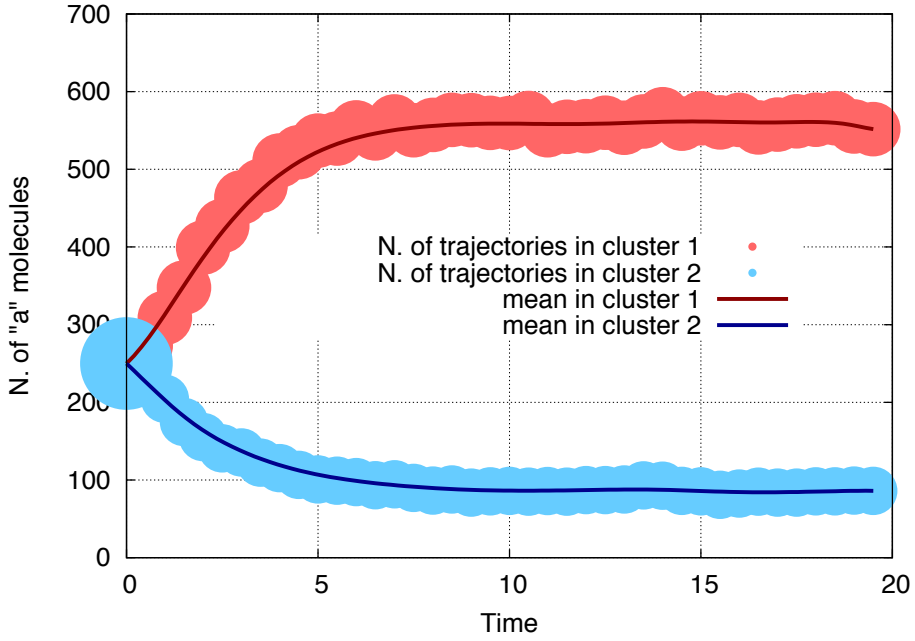
$$200b\ 250a.$$

Circles diameters are proportional to each cluster size.

Fig. 8.6 reports the analysis of convergence for the *Schlögl* model. In this case the confidence interval is computed (in pipeline) *after* the clustering of the results. Notice that, in this case (and in general in multi-stable outputs) the clustering of results is a key feature since confidence intervals of the average before the clustering of results generally do not decrease their amplitude with an increase of simulation trajectories.



(a) simulations, mean and standard deviation



(b) K-means clustering

Figure 8.5: Simulation results on the *Schlögl* model. Upper figure reports the raw simulation trajectories, the mean and standard deviation. Lower figure shows the on-line clustering results using K-means.

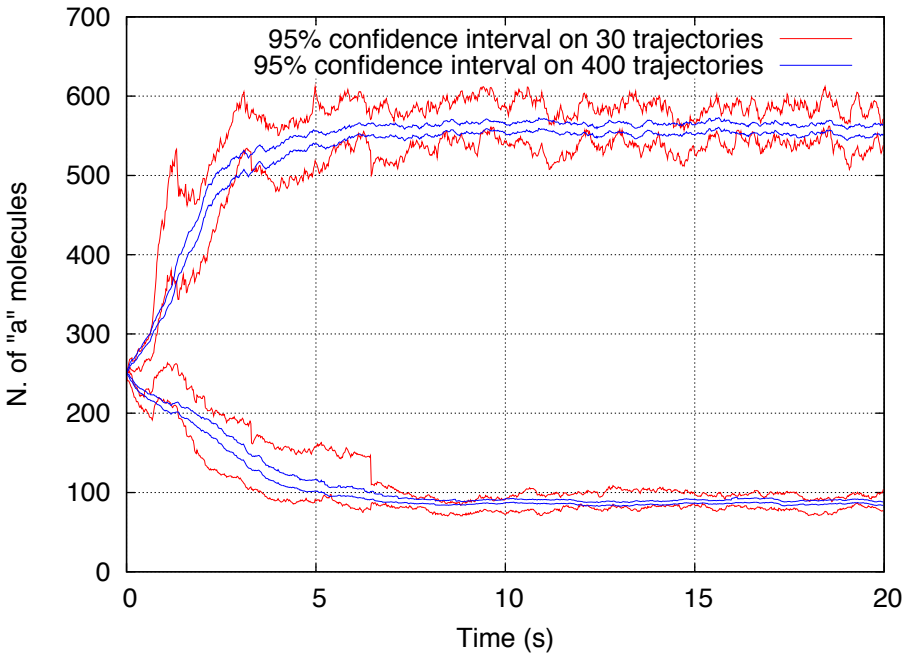


Figure 8.6: Per-cluster convergence of the sample average on the *Schlögl* model

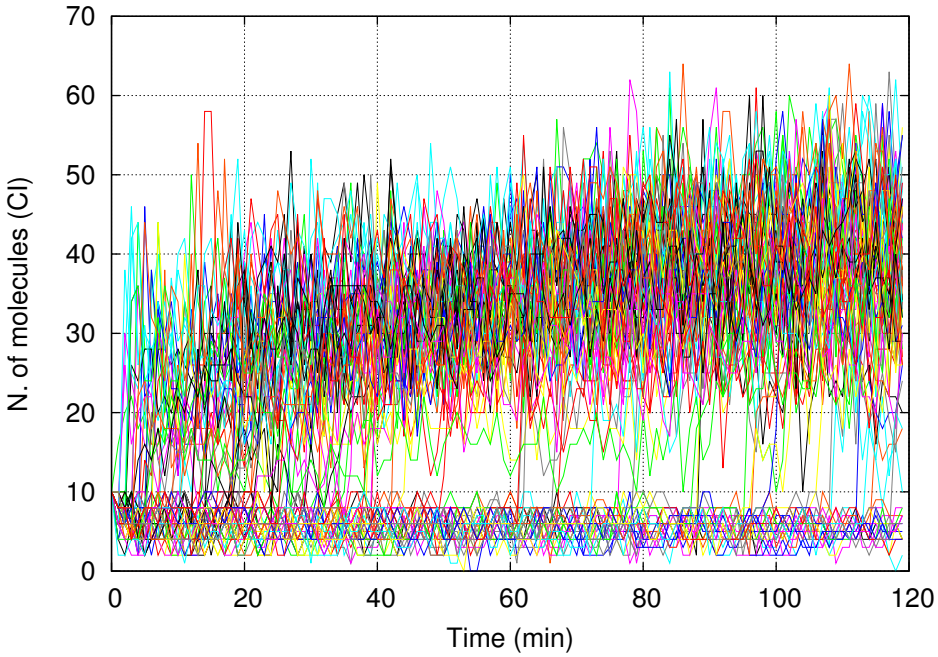
8.1.3 Multi-stable systems

A well-studied example of multi-stability in genetic systems is the *Bacteriophage* λ life cycle [26]. This process involves two different biological entities delimited by membranes, the phage and the bacterium. Lambda phage is a virus particle consisting of a head, containing a double-stranded linear DNA, and a tail. The phage particle recognises and binds to its host, *Escherichia Coli*, causing the DNA in the head of the phage to be ejected through the tail into the cytoplasm of the bacterial cell. After this, it can enter into one of two alternative stages called lysogeny and lysis. The lysogenic stage is a dormant stage in which the phage inserts its DNA into the host DNA and passively reproduces with the host. The only protein expressed in this phase is the λ repressor CI. When the host becomes stressed, the phage is more likely to go into lysis, in which case it reproduces more phages, kills the host and spreads to other bacteria. The decision between lysis and lysogeny can be thought of as a switching mechanism. A simplified model for the bacteriophage was proposed in [72]. In their model, the gene cI expresses the λ repressor (denoted by the symbol CI), which dimerises (denoted by $CI2$) and binds to DNA (denoted by D) as a transcription factor at either of two binding sites. The binding of the transcription factor to the site enhancing the transcription of CI (positive feedback) is denoted by D^+CI2. The phagic DNA in state D^+CI2 leads the lysogenic stage. The binding of the transcription factor to the site repressing the transcription of CI (negative feedback) is denoted by D^-CI2. The symbol D^+CI2D^-CI2 denotes the phagic DNA when both sites are bound ($CI2$ can bind to the repressing site also when another $CI2$ dimer is bound to the promoting site, with a global repressing effect).

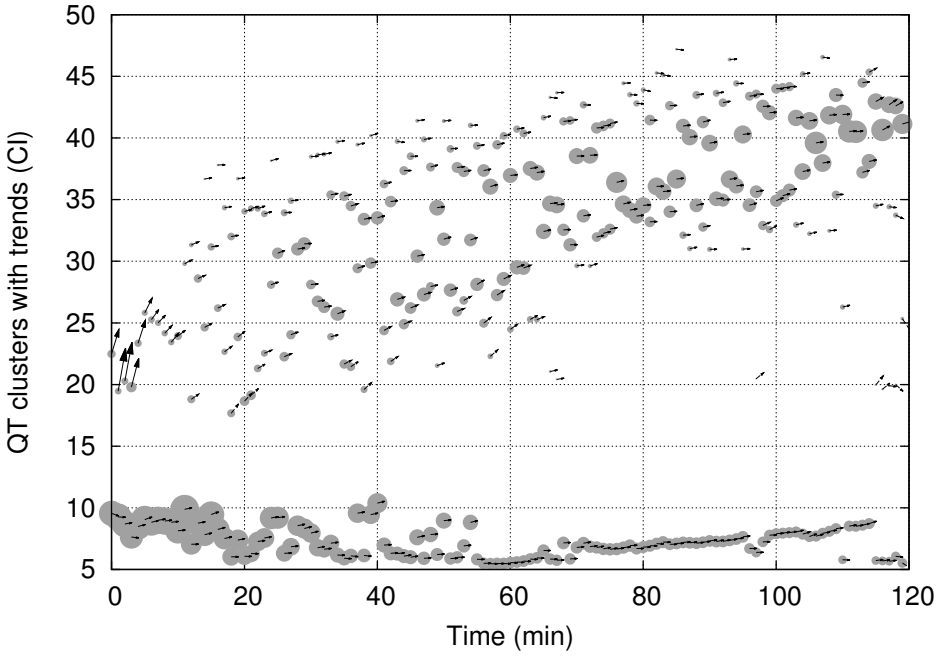
$$
\begin{aligned}
&CI \;\; CI \overset{0.05}{\longmapsto} CI2\\
&CI2 \overset{0.5}{\longmapsto} CI \;\; CI\\
&CI2 \;\; D \overset{0.026}{\longmapsto} D^+CI2\\
&D^+CI2 \overset{0.026}{\longmapsto} CI2 \;\; D\\
&CI2 \;\; D \overset{0.026}{\longmapsto} D^-CI2\\
&D^-CI2 \overset{0.026}{\longmapsto} CI2 \;\; D\\
&D^+CI2 \;\; CI2 \overset{0.13}{\longmapsto} D^+CI2D^-CI2\\
&D^+CI2D^-CI2 \overset{0.13}{\longmapsto} D^+CI2 \;\; CI2\\
&D^+CI2 \;\; P \overset{40}{\longmapsto} D^+CI2 \;\; P \;\; CI2 \;\; CI2\\
&CI \overset{0.0007}{\longmapsto} \bullet
\end{aligned}
$$

Figure 8.7: CWC rules for the *Bacteriophage* λ dynamics

The CWC rules in this system are reported in Fig. 8.7, where P represents the RNA polymerase, assumed here to be constant, and two proteins per mRNA transcript were considered.



(a) raw simulations



(b) QT clustering

Figure 8.8: Simulation results on the *Bacteriophage* λ model. Upper figure reports the 100 raw trajectories, lower figure shows the on-line QT clustering results.

In this model the stochastic time trajectories of *CI* switch between two stable equilibria if the noise amplitude is sufficient to drive the trajectories occasionally out of the basin of attraction of one equilibrium into the basin of attraction of the other equilibrium (Fig. 8.8a).

Fig. 8.8b shows the resulting clusters (gray circles) computed on-line using QT on the *Bacteriophage* λ model for species *CI* over 100 stochastic simulations starting with the term:

$$10\,CI\ \ D\ \ P.$$

Circles diameters are proportional to each cluster size and arrows display the local trends of the clustered trajectories.

K-means is suitable for stable switch systems where the number of clusters and their tendencies are known in advance, in the other cases QT, although more computationally expensive, can build accurate partitions of trajectories giving evidence of instabilities with a dynamic number of clusters.

8.1.4 Oscillatory systems

We examine here the theoretical model for circadian oscillations based on transcriptional regulation of the frequency (frq) gene in the fungus *Neurospora*. The model relies on the feedback exerted on the expression of the frq gene by its protein product FRQ. In this model, sustained rhythmic variations in protein and mRNA (M) levels occur, in the form of limit cycle oscillations [84].

$$\top : (x \rfloor FRQ\ \ X)^{\eta} \overset{f_{FRQ}(t)}{\longmapsto} (x \rfloor FRQ\ \ X)^{\eta}\ \ M \qquad \top : M \overset{0.5}{\longmapsto} M\ \ FRQ$$
$$\top : M \overset{f_M}{\longmapsto} \bullet \qquad \top : FRQ \overset{f_d}{\longmapsto} \bullet$$
$$\top : FRQ\ \ (x \rfloor X)^{\eta} \overset{0.5}{\longmapsto} (x \rfloor FRQ\ \ X)^{\eta} \qquad \top : (x \rfloor FRQ\ \ X)^{\eta} \overset{0.6}{\longmapsto} FRQ\ \ (x \rfloor X)^{\eta}$$

Figure 8.9: CWC rules for the *Neurospora* dynamics

The CWC rules modelling this case are reported in Fig. 8.9, where the cytosol is referred as the $\top$ compartment while η is the label of the compartment representing the nucleus. The model is based on the negative feedback exerted by the protein FRQ on the transcription of the frq gene; the rate of gene expression is enhanced by light. The model includes gene transcription in the nucleus, accumulation of the corresponding mRNA in the cytosol with the associated protein synthesis, protein transport into and out of the nucleus, and regulation of gene expression by the nuclear form of the FRQ protein. The function

$$f_{FRQ}(t) = v_s(t)\frac{K_I^n}{FRQ^n + K_I^n}$$

denotes the rate of frq transcription. The parameter $v_s(t)$ defined by:

$$v_s(t) = \begin{cases} 200 \text{ when } 2nT \leq t < (2n+1)T \\ 160 \text{ when } (2n+1)T \leq t < (2n+2)T \end{cases} \quad (n \geq 0)$$

increases in light conditions of the current time of the simulation, where T represents the period of the dark-light phases. The constant K_I is related to the threshold beyond which nuclear FRQ represses frq transcription; the Hill coefficient n, characterises the degree of cooperation of the repression process. In the functions, the occurrence of a symbol indicates its multiplicity. The mRNA degradation is given by the Michaelis rate function:

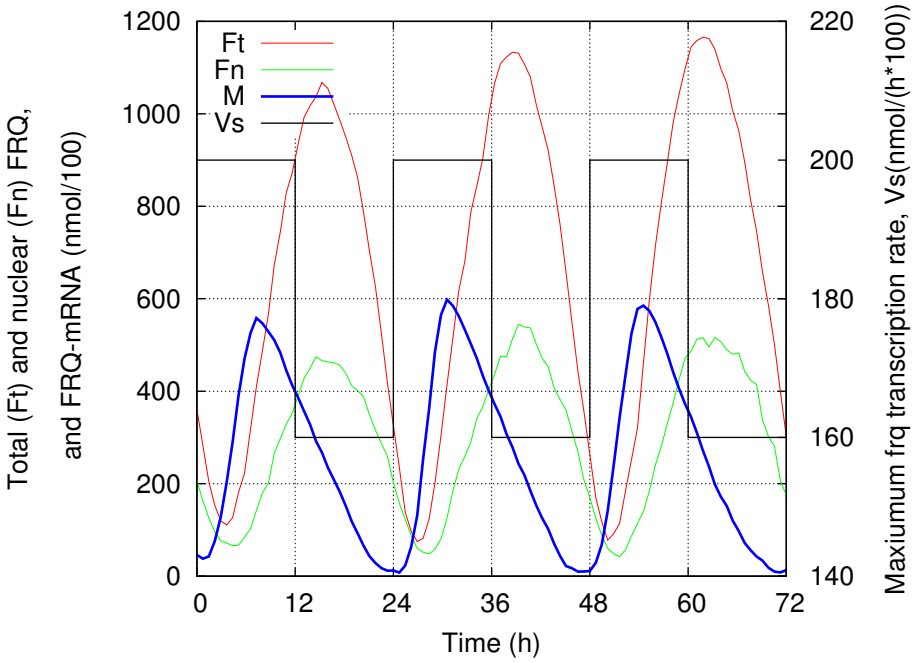
$$f_M = v_m\frac{M}{K_M + M}.$$

The FRQ degradation is given by the Michaelis rate function:

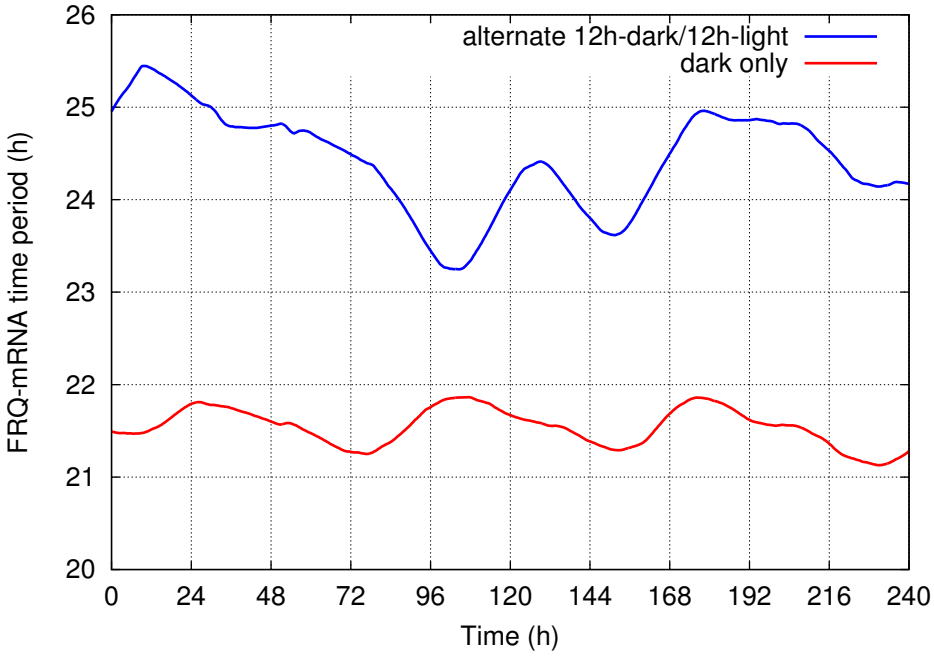
$$f_d = v_d\frac{FRQ}{K_d + FRQ}$$

where v_d is the maximum rate of FRQ degradation and the Michaelis constant related to this process is K_d.

As in [84] we modelled the oscillations under two different conditions:



(a) Cytosolic FRQ oscillations in light/dark conditions



(b) Peaks frequency of the cytosolic FRQ

Figure 8.10: Simulation results of the cytosolic FRQ protein of the *Neurospora* model.

1. constant dark condition;

2. alternate light and dark phases.

Following [84], the values of the parameters are set as: $v_m = 50.5$, $v_d = 140$, $k_s = 0.5$, $k_1 = 0.5$, $k_2 = 0.6$, $K_m = 50$, $K_I = 100$, $K_d = 13$, $n = 4$. Concentrations have been made discrete by scaling 1nM to 100 symbolic elements. In the constant dark condition, parameter v_s is equal to 160, in the alternate condition, v_s is equal to 160 during the dark phase and to 200 during the light phase. Fig. 8.10a shows an extract of a single stochastic simulation of the circadian oscillations in the dark/light alternate condition, plotting the number of FRQ proteins within the nucleus, the total number of FRQ proteins in the cell and the number of mRNA molecules leading the synthesis of FRQ. Fig. 8.10b shows the outcome of the peak detection tool, which is able to summarize the frequency of the peak events over time. The plot results after capturing the peaks in the curve of the cytosolic mRNA for the FRQ protein synthesis. Measuring the distance between two consecutive peaks, we compute the period of each oscillation and then plot the moving average, over 200 simulations, of the local periods. In the constant dark condition, the circadian period is close to 21 and half hours. It increases, producing damping oscillations with a period of approximately 24 hours, in the dark/light alternate condition.

8.2 Performance

In this section we report some experimental results about performances of the realised tool. Variants of the tool have been experimentally evaluated on different computing platforms, as summarised in Table 8.1.

Tool variant	**Computing platform**
SIMD-enriched	Intel workstation equipped with a quad-core Xeon E5520 Nehalem (16 HyperThreads) @2.26GHz with 8MB L3 cache and 24 GBytes of main memory with Linux x86_64. Each core is equipped with a SSE4.2 SIMD engine.
Multicore	Intel workstation equipped with 4 eight-core E7-4820 Nehalem (32 cores, 64 contexts) @2.0GHz with 18MB L3 cache and 64 GBytes of main memory with Linux x86_64.
Distributed	Infiniband connected cluster of Intel workstations, each of them equipped with 2 six-core Xeon-X5670 (12 HyperThreads) @3.0GHz with with 12MB L3 cache and 24 GBytes of main memory with Linux x86_64.
Distributed	8 virtual machines (VMs) each having 4 cores Intel E-2670 2.6 GHz with 20MB of L3 cache running in the Amazon EC2 with Linux x86_64.
Distributed	Heterogeneous environment including: 8 EC2 virtual machines with 4 virtualised cores, two workstations at University of Pisa, each having 16 cores Intel Sandy Bridge @2GH.z with 20MB of L3 shared cache and one workstation at University of Torino, having 32 cores Intel Nehalem @2.0GHz with 18MB of L3 shared cache. All machines run Linux x86_64.

Table 8.1: Tool variants and computing platforms for performance experiments

8.2.1 SIMD-enriched CWC Simulator

The SIMD-based approach discussed in Sec. 5.2.2 is experimentally evaluated in terms of speedup results.

All the reported experiments have been executed on an Intel workstation with 2 quad-core Xeon E5520 Nehalem (16 HyperThreads) @2.26GHz with 8MB L3 cache and 24 GBytes of main memory with Linux x86_64. Each core is equipped with a SSE4.2 SIMD engine.

Table 8.2 reports the achieved speedup on a single core of the workstation, for a n-species generalisation of the well-known *Lotka-Volterra* model (the 2-species case is the

# of species	Sequential (s)	SIMD (s)	Speedup	Ideal speedup
2	5.021	5.071	0.99	4
4	19.076	18.887	1.01	4
8	70.743	70.043	1.01	4
16	284.276	278.701	1.02	4
32	1121.231	1099.245	1.02	4

Table 8.2: Execution time (s) and speedup of the SIMD-enriched CWC Simulator against the sequential version on a n-species *Lotka-Volterra* model.

standard *Prey-Predator* model).

Despite SSE exhibits very low overhead, the achieved speedup is almost negligible because only a fraction of the whole simulation step has been actually parallelised (Amdahl's law applies [24]).

8.2.2 CWC Simulator on multicore

The CWC Simulator workflow is experimentally evaluated in terms of speedup and throughput results.

All reported experiments have been executed on an Intel workstation equipped with 4 eight-core E7- 4820 Nehalem (32 cores, 64 contexts) @2.0GHz with 18MB L3 cache and 64 GBytes of main memory with Linux x86_64.

As we shall see, the number of statistical engines could be chosen according to a simple but effective performance model, which is made possible by the high-level approach of the design. According to the same model, the most interesting sensitivity analysis under performance viewpoint concerns the number of simulation engines.

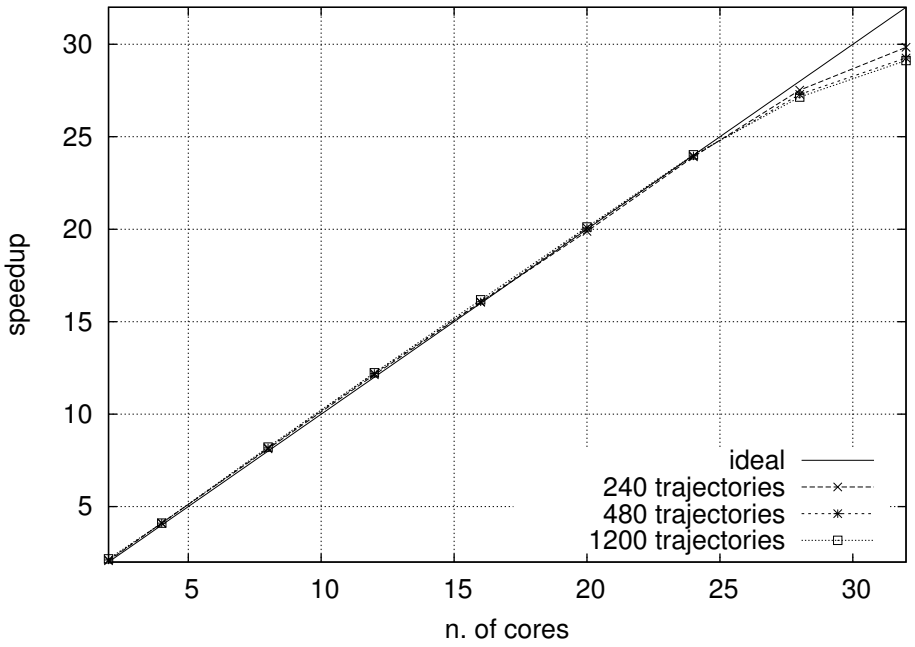
As case studies, we consider the simulation workflow for the *Neurospora* and *Bacteriophage* λ models. In the former case, we evaluate the speedup with different values of samples per trajectory. In the latter case, we evaluate the benefits provided by the parallelisation of the statistical engines.

Neurospora

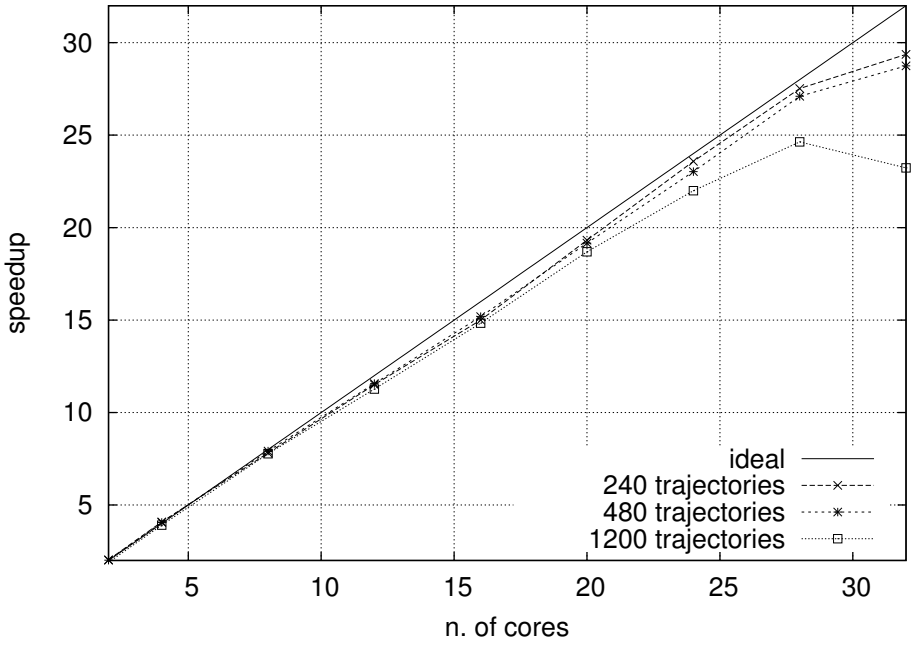
We consider the simulation workflow for the transcriptional regulation of the *Neurospora* model (8.1.4). The analysis pipeline is configured with 3 statistic engines executing mean, standard deviation, quantiles, K-means, QT and frequency detection filters.

Fig. 8.11a and Fig. 8.11b show the speedup obtained for the whole workflow on varying the number of concurrent simulation engines (one per core), where the simulation points (or *samples*) per trajectory, is set to be 10^4 and 10^5 simulation points, respectively.

The speedup on the total execution time achieved in the former case (Fig. 8.11a), scales ideally with respect to the number of simulation engines, whereas a performance



(a) 10^4 simulation points



(b) 10^5 simulation points

Figure 8.11: Speedup on the *Neurospora* model for different numbers of simulation points

penalty is paid in the latter case (Fig. 8.11b) for the highest degree of parallelism and number of produced trajectories.

Bacteriophage λ

We consider the simulation workflow for the transcriptional regulation of the *Bacteriophage* λ life cycle model (8.1.3).

The speedup of the workflow is shown in Fig. 8.12, against the number of simulation engines (one per core) in two configurations. Fig. 8.12a uses a single statistical engine in the analysis pipeline, while Fig. 8.12b uses a farm with 4 statistical engines. Observe that, in the case of a single statistical engine, the simulator succeeds to effectively use all the simulation engines only up to 512 independent simulations (producing 512 independent trajectories). The speedup decrease as soon as the output data (linear in size with the number of trajectories) become significant, due to the cost of on-line data filtering and analysis, i.e. as soon as the simulation pipeline and the analysis pipeline become unbalanced. As discussed in the previous chapters, the filtering and analysis of data has not a negligible cost for large data sets.

Discussion

The very same speedup behaviour is achieved for other test cases, and it is worth a detailed discussion.

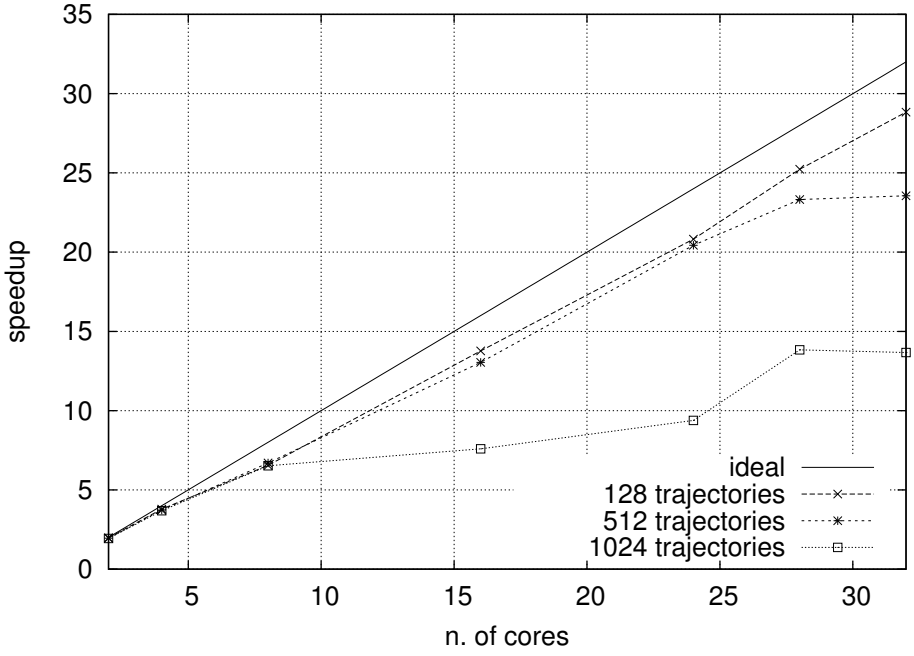
For each experiment the total number of FastFlow nodes, i.e. the boxes depicted with solid lines in Fig. 6.2, is

$$\begin{gathered}\#(\text{simulation engines}) + \#(\text{statistical engines}) + \\ \#(\text{other nodes}) + \#(\text{FastFlow support nodes}) = \\ \#(\text{simulation engines}) + \#(\text{statistical engines}) + \\ 3 + 4\end{gathered}$$

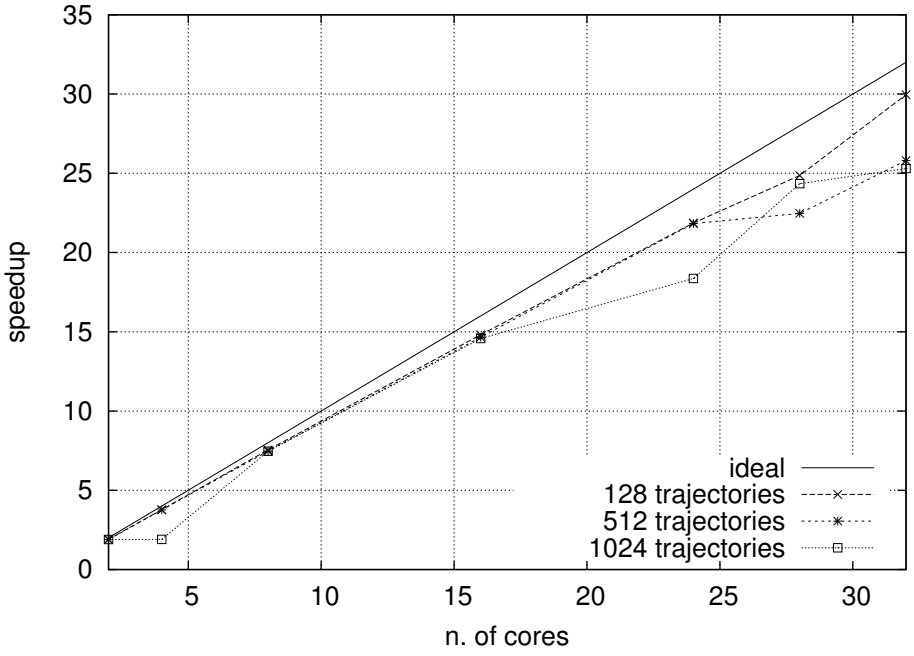
where "other nodes" are "generation of simulation tasks", "alignment of trajectories" and "generation of sliding windows of trajectories" nodes, whereas "FastFlow support nodes" are the two couples of dispatch-gather nodes in Fig. 6.2.

Observe that for each performance experiment all the runs are executed by fixing random seeds (see Appendix A). Thus, given a set of simulation parameters, it can be verified that the stochastic simulation of the single trajectory requires exactly the same number of iterations and the simulated time progress identically across random walks irrespectively of the number of simulation/statistic engines and *observed* simulation points, which can be considered a (synchronised) sampling at fixed simulation times of trajectories. These observations imply that:

- the parallelism strategy does not break determinism and reproducibility of results (correctness);
- as reflected in the speedup results, the design of the simulator ensures an effective load balancing and low synchronisation overheads;



(a) sequential statistics



(b) 4 parallel statistical engines

Figure 8.12: Speedup on the *Bacteriophage* λ model with parallel statistics

Model	Single trajectory information			Overall data (20 sim eng, 3 stat eng)		
	N. samples	Avg sim step	Sample time	Inter-arrival time	Throughput	Output size
Neurospora	10^4	7.80 μs	517.24 μs	25.86 μs	2.70 MB/s	82.40 MB
Neurospora	10^5	8.37 μs	55.51 μs	2.78 μs	28.59 MB/s	823.98 MB
Neurospora	10^6	75.63 μs	4.65 μs	232.68 ns	303.86 MB/s	8.24 GB
Escherichia Coli	10^6	173.64 μs	0.58 μs	28.81 ns	476.30 MB/s	9.15 GB
Lotka-Volterra	10^6	22.86 μs	0.69 μs	34.68 ns	237.05 MB/s	7.32 GB

Table 8.3: Throughput supported by the CWC Simulator

- the efficiency of parallel execution depends on the order of magnitude of the observed simulation points, and by the number of produced trajectories.

This latter point specifically exploits the working hypothesis: stochastic methods are particularly informative when used to simulate the model at high resolution, i.e. high number of samples and trajectories. In this case, the main bottleneck of simulation software is data movement and management since the computation/data-movement ratio may easily reach the limits of modern multicore platforms.

In multicore platforms, "observing" the phenomena at a high frequency has non-trivial effects on simulation/analysis process since the frequency of observation determines both the quality of results and, inversely, the overall speedup. As shown in Fig. 8.11a, Fig. 8.11b and Table 8.3, the proposed design and implementation effectively cope with this trade-off and succeed to exploit high rates of data movement. Thanks to merging many independent trajectories happening in the simulation pipeline, the output size, and thus the required disk throughput is greatly reduced (unless the storage of raw simulation results, happening among the two pipelines, is requested by the user).

The proposed simulation architecture is not only fast but also highly predictable in terms of performance. This latter aspect is mainly due to the high-level structured design [10]. The whole workflow is a pipeline of two pipelines (i.e. a pipeline), whose performance can be modelled by means of the *service time* (Ts) of each stage S_i. In particular:

$$Ts(\mathsf{pipeline}(S_1, S_2, \dots S_k)) = \max\{\, Ts(S_1),\, Ts(S_2), \dots\, Ts(S_k)\}$$

where $Ts(S_i)$ models the average inter-departure time of stream items of the stage i of the pipeline, which actually matches the average computation time of S_i to produce one stream item. In turn, some of the stages are farm, which exploit n independent replicas of a (sequential or parallel) worker, e.g. simulation and static engines. Its service time can be modelled as:

$$Ts(\mathsf{farm}(W, n)) = Ts(W)/n$$

Given the service time of each sequential stage, e.g. measured in the sequential code, these equations can be exploited to tune the optimal number of workers n for any new

simulation problem and to understand its upper bound in terms of speedup. As an example, in the Neurospora with 10^5 samples test case the sequential code exhibits the following timing per trajectory: $Ts(\mathsf{generation}) \sim 0$, $Ts(\mathsf{sim\ eng}) = 5.3$ s, $Ts(\mathsf{alignment}) = 0.11$ s, $Ts(\mathsf{windows\ generation}) = 0.02$ s, $Ts(\mathsf{stat\ eng}) = 0.33$ s, with a total execution time for each trajectory of ~ 5.8 s (~ 120 minutes for 1200 trajectories). Among those, sim eng and stat eng are used within a farm, thus their service time can be reduced by increasing the number of workers. Therefore, the maximum performance and efficiency of the whole workflow is reached when the two farms are tuned to match the service time of the slowest sequential stage, i.e. the alignment stage 0.11s). For this, the farm in the simulation pipeline should be configured with $n = 5.3/0.11 \approx 48$ workers, whereas the farm in the analysis pipeline with $n = 0.33/0.11 = 3$ workers. The overall speedup upper bound can be obtained using the total execution time and the slowest stage service time, i.e. maximum speedup achievable for this test case is $\approx 5.8/0.11 = 53$, which includes the contributes from both the pipeline and the farm. The analysis, despite being approximated since does not include synchronisation overheads and memory bandwidth limits, is adherent of results depicted in Fig. 8.11a since the speedup linearly grows with the number of simulation engines in the $n = [1..32]$ range, where the primary reasons of the slight performance drop in the right end of the plots is due to the fact that more virtual cores (i.e. hyper-thread contexts) than physical core are used, and the increased memory traffic for high numbers of trajectories. Furthermore, the performance analysis highlights that the bottleneck of the architecture for high throughput problems, is in the alignment of trajectory stage. Its parallelisation, which can be addressed by pipelining simulation engines and a partial alignment stage within the farm, is among future works.

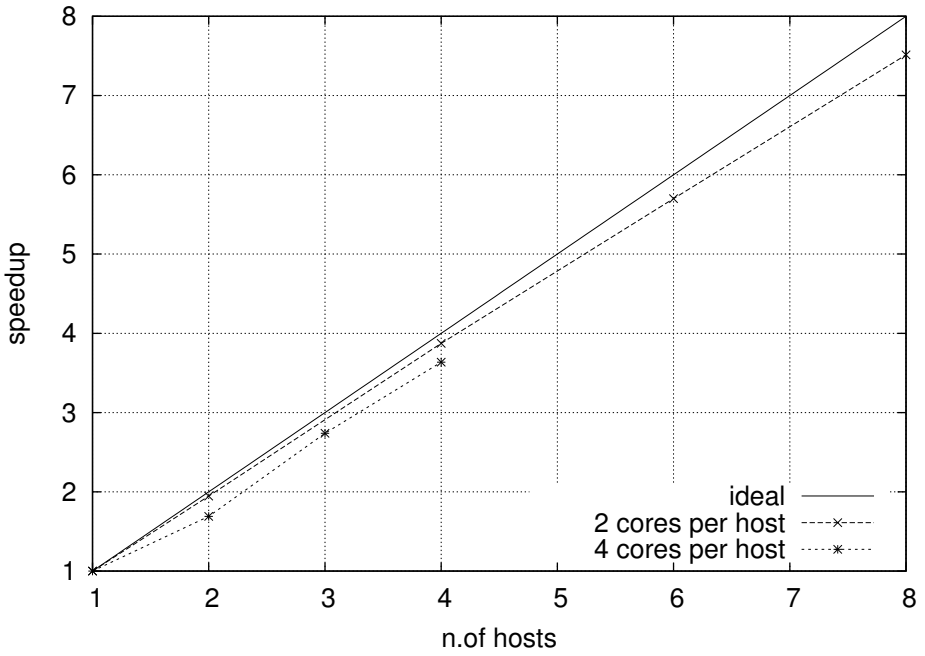
However, this simple reasoning, does not apply when a big number of trajectories is modelled. In fact, in such cases, the main architecture bottleneck when using an high number of simulation engines, is the memory bandwidth limit of the underlying platform. Such effect can be seen in the plot of Fig. 8.11b for the case 1200 trajectories.

8.2.3 CWC Simulator on multicore cluster

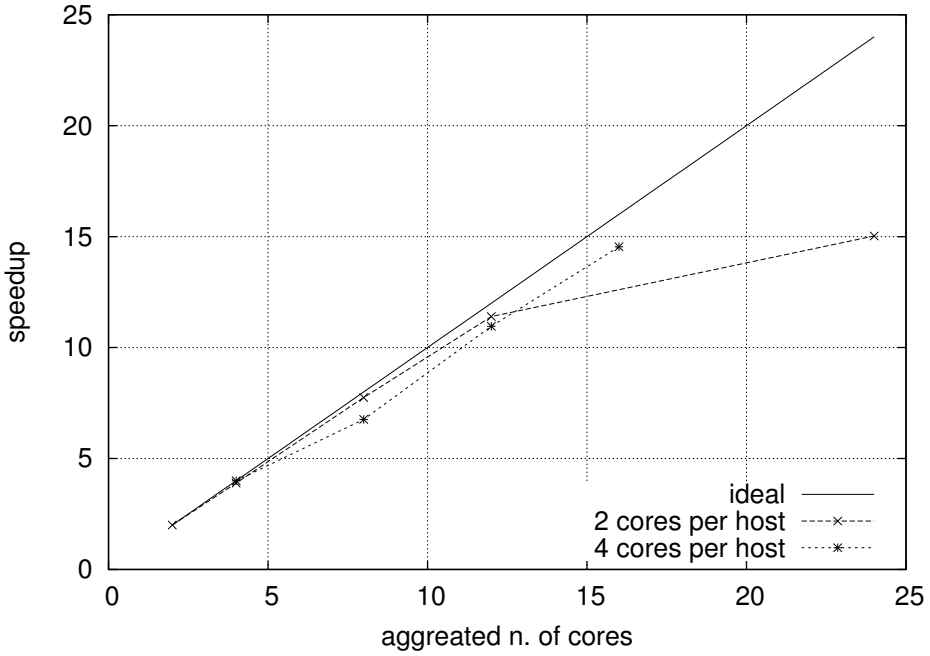
All reported experiments have been executed on an Infiniband connected cluster of Intel workstations, each of them equipped with 2 six-core Xeon-X5670 (12 HyperThreads) @3.0GHz with with 12MB L3 cache and 24 GBytes of main memory with Linux x86_64. The Infiniband network is used via the TCP/IP stack (IPoIB).

The experiments have been executed over the *Bacteriophage* λ use case.

The speedup of the distributed version of the simulator is shown in the preliminary experiments in Fig. 8.13a and Fig. 8.13b, against the number of hosts and the aggregated count of cores in different hosts, respectively. In both cases 4 statistical engines are used. As shown in Fig. 8.13a, the simulator exhibits a reasonable scalability with respect to the number of hosts. In this case, the speedup is also influenced by the number of simulation engines per hosts since the kind of latency and bandwidth involved in data streaming depend on the kind of channel (shared-memory or network). Despite the architecture exhibits a reasonable neutrality with respect to engines-to-cores-to-host mapping (as shown in Fig. 8.13b), further experimentation is needed to tune message



(a) w.r.t. the number of hosts



(b) w.r.t. aggregated number of cores

Figure 8.13: Speedup of the distributed simulator on the *Bacteriophage* λ life cycle on the Intel multicore cluster with two different usage of cores per node.

sizes (via message coalescing) and study the bottlenecks of the design on larger configurations and different simulations. It can be expected, for example, that alignment of trajectories stage can be a bottleneck, as it receives the full pressure of data produced by many hosts. Lifting data-alignment to the simulation pipeline can help in mitigating this effect, as this stage aggregates output items (i.e. coalesces many small data items in fewer larger data items), thus reducing the impact of network latency in distributed communications.

8.2.4 CWC Simulator on cloud

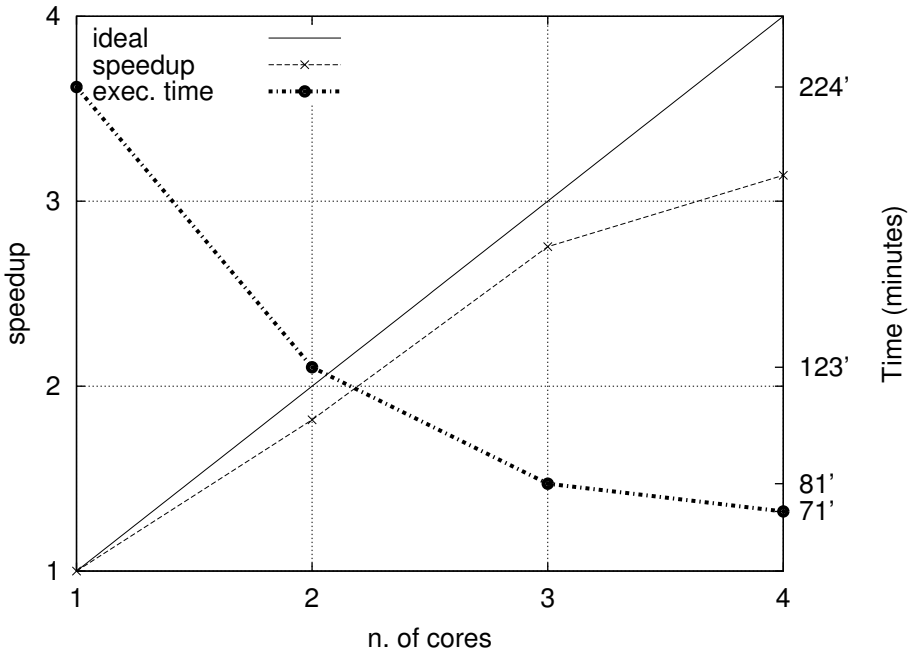
The distributed CWC Simulator have been tested for performance on the Amazon Elastic Compute Cloud (Amazon EC2) [23] and on a heterogeneous private/public cloud.

The performance of the simulator was tested on the *Neurospora* model, described in 8.1.4. We ran two set of experiments: the first one considering 8 virtual machines (VMs) each having 4 cores Intel E-2670 2.6 GHz with 20MB of L3 cache running in the Amazon EC2; the second set considering an heterogeneous environment of virtual and physical machines which allowed to scale the core count up to 96. The heterogeneous environment, which can be considered a private cloud including a public cloud comprises: 8 EC2 virtual machines with 4 virtualised cores, two workstations at University of Pisa, each having 16 cores Intel Sandy Bridge @2GH.z with 20MB of L3 shared cache, and one workstation at University of Torino, having 32 cores Intel Nehalem @2.0GHz with 18MB of L3 shared cache. Virtual and physical machines run Linux x86_64.

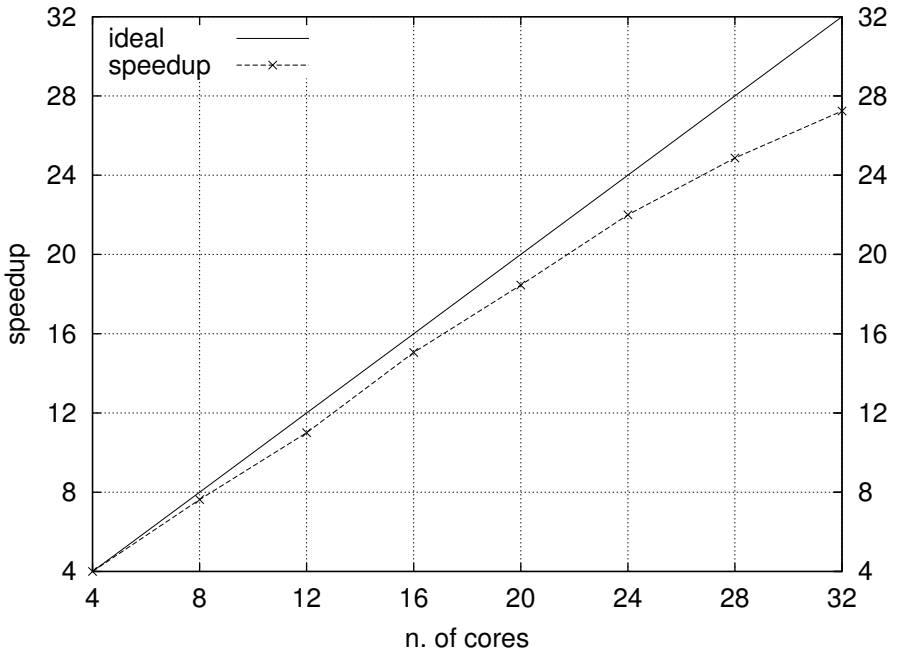
In the first test we measured the speedup and the execution time of the simulator when running 96 days of simulation time on a single quad-core VM. The results obtained are shown in Fig. 8.14a. In this case, the maximum speedup using all available cores is 3.15 out of 4 so that the execution time decreases from about 224 min of the sequential run down to about 71 min.

Next, we executed the same test using 8 quad-core VMs. Figure 8.14b reports the speedup for the same simulation time varying the number of virtual cores used. The trend is almost ideal. With 32 virtual cores we obtained a completion time of 10.5 min, with a gain of about 27x with respect to the sequential execution time of the simulator on a single-core VM of the same clock frequency and a gain of $\sim$7x with respect to the execution time obtained on the single quad-core VM.

In the second set of experiments, we executed the simulation using different platforms. Initially we ran the simulator on the 32 cores Nehalem workstation using the shared memory implementation of the simulator. The minimum execution time obtained on that machine using all cores available is 67.3 min, i.e. almost the same time obtained using the 8 Amazon VMs (having an overall number of 32 virtualised cores). This result confirms the quality of the distributed implementation of the simulator. Next, in order to further decrease the simulation time, we used together the Nehalem workstation, the Amazon VMs, and the two 16 cores Sandy Bridge workstations. In this case, since the machines were not homogeneous in terms of number of cores and computational power, we used a weighted dispatching policy for the distribution of the simulations, where the weights used are the number of virtualised or physical cores of the target



(a) single quad-core VM: speedup and execution time.



(b) virtual cluster of eight quad-core VMs: speedup.

Figure 8.14: Performance of the simulator for the *Neurospora* model on two configurations of the Amazon EC2 cloud, varying the number of virtualized cores.

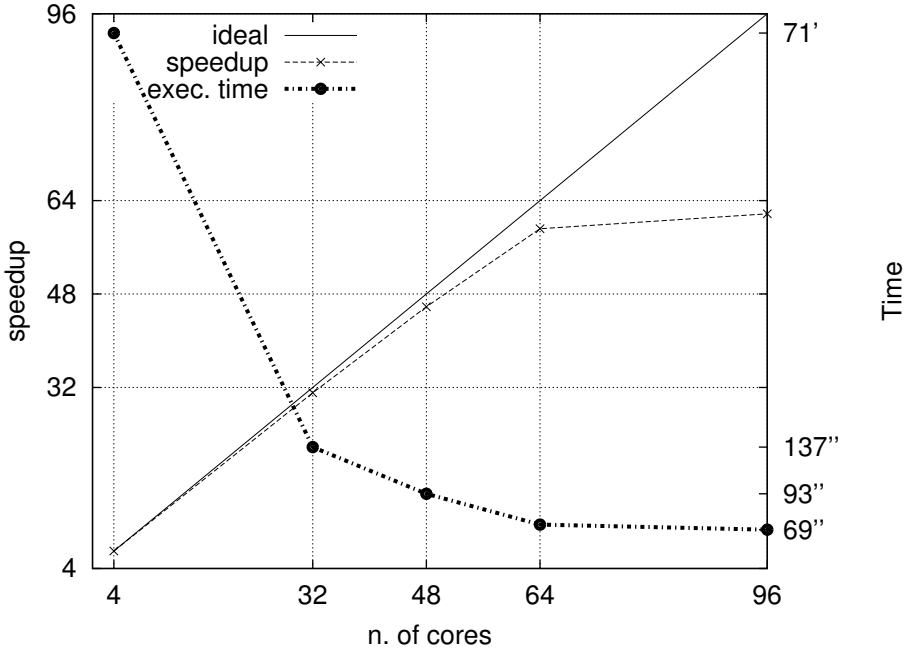


Figure 8.15: Performance of the simulator for the *Neurospora* model on an heterogeneous environment of virtualized and physical machines (eight quad-core Amazon EC2 VMs, 1 32-core Nehalem workstation and 2 16-core Sandy Bridge workstations): speedup and execution time varying the number of cores.

platform. The results obtained for the execution time and the speedup (the speedup is computed w.r.t. the execution time obtained on single-core Amazon VM), are shown in Figure 8.15. For this test, the analysis pipeline was mapped on the 32 cores Nehalem workstation. The minimum execution time obtained using 96 cores (32 cores in the 8 quad-core VMs, 32 cores in the Nehalem workstation and 2x16 cores in the 2 Sandy Bridge workstations) is 69.3 s, carrying a gain of ∼62x in the execution time, which a remarkable result considering the low computation granularity (∼20 ms) of the single worker thread and the high frequency of communication (30 – 80 ms) for collecting results computed by remote machines running the simulation pipeline. As a general rule, the lower the communication/computation ratio (i.e. the coarser the grain), the higher the speedup obtained. The test considered has a not optimal communication/computation ratio (i.e. it is fine grained) and for this reason we were not able to obtain a performance improvement with more than 64 cores.

9 Conclusions

In this work we presented a novel approach for the design of portable stochastic simulators (especially for biological systems), based on regarding the whole simulation/analysis process as a single stream-processing workflow and accelerating it up by way of high-level programming frameworks for parallel computing. In particular, the stream-oriented design of the simulation/analysis workflow make it possible to perform the statistical analysis and data mining of simulation results as an on-line process starting together with simulation and immediately starting to provide the user with a stream of final results, thus enforcing a fast feedback to the bioinformatics scientists.

We have discussed the main issues arising from the parallelisation of the simulation stage (supporting on-line analysis) on commodity multicore platforms. In particular, we distinguished two different approaches to parallelisation, i.e. the parallelisation of the single simulation instance and many simulation instances. For each class we have defined a number of design guidelines, which may support the easy and efficient porting of this class of algorithms on multicores. These guidelines include both the programming language abstractions (streams and high-level programming patterns), the run-time mechanisms (lock-free cache-friendly inter-core synchronisations here provided by the FastFlow framework), and basic simulator architectural schema (simulation "objectification", interleaved execution and pipelined reduction), which can be gracefully optimised with limited effort to experiment different parallel execution behaviours.

We have also discussed the problem of the effective analysis of stochastic simulation outcomes, which can be complex to interpret also due to intrinsic stochastic "noise" and the overlapping of the many required experiments by the Monte Carlo method. At this aim, we characterised some patterns of behaviour for biological system dynamics, e.g. monostable, multi-stable, and oscillatory systems, and we exemplified them with minimal yet paradigmatic examples from the literature. For these, we identified data filters able to provide statistically meaningful information to the biological scientists in order to simplify the data analysis. We defined some guidelines to design an effective and expressive analysis stage, supporting the parallel execution of multiple user-definable filters on large (even big) data streams.

In order to demonstrate the validity of the presented approach, we used CWC and its simulator as a test bed. The presented guidelines have been used to develop a multicore-aware CWC Simulator on top of the high-level FastFlow programming framework, then we have extended it to support distributed (and public cloud) platforms.

We think the issues for the portable and efficient design of the workflow are paradigmatic for a broad class of algorithms for bioinformatics, and more generally for the implementation of other Monte Carlo methods. Both the simulations and the on-line statistic filters, which are both parallel and pipelined, can be easily extended with new

simulation algorithms and filters thanks to FastFlow-based parallel infrastructure that exempt the programmer from synchronisation and orchestration of concurrent activities.

Experimentation on various execution environments demonstrates that its high-level design via the FastFlow framework provides the application designer with easy engineering, seamless portability on distributed and multicore platforms (physical or virtualised), and automatic load balancing. Moreover, experimental evaluations show that the design is flexible and robust with respect to target platform, and it is able to provide performance scalability on heterogeneous parallel platforms, also for fine-grained problems.

We believe that the design has the potentiality to survive in the hostile environment populated by platform heterogeneity, coding complexity, high-performance and performance portability. In this regard, we believe it is an evolution of the species of simulators for systems biology.

Bibliography

[1] Aldinucci, M. eskimo: experimenting with skeletons in the shared address model. *Parallel Processing Letters 13*, 3 (sep 2003), 449–460.

[2] Aldinucci, M., Bracciali, A., Liò, P., Sorathiya, A., and Torquati, M. StochKit-FF: Efficient systems biology on multicore architectures. In *Euro-Par 2010 Workshops, Proc. of the 1st Workshop on High Performance Bioinformatics and Biomedicine (HiBB)* (Ischia, Italy, aug 2011), M. R. Guarracino, F. Vivien, J. L. Träff, M. Cannataro, M. Danelutto, A. Hast, F. Perla, A. Knüpfer, B. D. Martino, and M. Alexander, Eds., vol. 6586 of *LNCS*, Springer, pp. 167–175.

[3] Aldinucci, M., Calcagno, C., Coppo, M., Damiani, F., Drocco, M., Sciacca, E., Spinella, S., Torquati, M., and Troina, A. On designing multicore-aware simulators for systems biology endowed with on-line statistics. *BMC Bioinformatics* (2013). Submitted for pubblication.

[4] Aldinucci, M., Campa, S., Ciullo, P., Coppola, M., Danelutto, M., Pesciullesi, P., Ravazzolo, R., Torquati, M., Vanneschi, M., and Zoccolo, C. A framework for experimenting with structure parallel programming environment design. In *Parallel Computing: Software Technology, Algorithms, Architectures and Applications (Proc. of PARCO 2003, Dresden, Germany)* (2004), G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, Eds., vol. 13 of *Advances in Parallel Computing*, Elsevier, pp. 617–624.

[5] Aldinucci, M., Campa, S., Ciullo, P., Coppola, M., Magini, S., Pesciullesi, P., Potiti, L., Ravazzolo, R., Torquati, M., Vanneschi, M., and Zoccolo, C. The implementation of ASSIST, an environment for parallel and distributed programming. In *Proc. of 9th Intl Euro-Par 2003 Parallel Processing* (Klagenfurt, Austria, aug 2003), H. Kosch, L. Böszörményi, and H. Hellwagner, Eds., vol. 2790 of *LNCS*, Springer, pp. 712–721.

[6] Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., and Torquati, M. Targeting distributed systems in FastFlow. In *Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing* (2013), vol. 7640 of *LNCS*, Springer, pp. 47—56.

[7] Aldinucci, M., Coppo, M., Damiani, F., Drocco, M., Sciacca, E., Spinella, S., Torquati, M., and Troina, A. On parallelizing on-line statistics for stochastic biological simulations. In *Euro-Par 2011 Workshops, Proc. of the 2st Workshop on High Performance Bioinformatics and Biomedicine (HiBB)*

(Bordeaux, France, 2012), M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. D. Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. L. Scott, J. L. Träff, G. Vallée, and J. Weidendorfer, Eds., vol. 7156 of *LNCS*, Springer, pp. 3–12.

[8] Aldinucci, M., Coppo, M., Damiani, F., Drocco, M., Torquati, M., and Troina, A. On designing multicore-aware simulators for biological systems. In *Proc. of Intl. Euromicro PDP 2011: Parallel Distributed and network-based Processing* (Ayia Napa, Cyprus, feb 2011), Y. Cotronis, M. Danelutto, and G. A. Papadopoulos, Eds., IEEE, pp. 318–325.

[9] Aldinucci, M., Coppola, M., Danelutto, M., Vanneschi, M., and Zoccolo, C. ASSIST as a research framework for high-performance grid programming environments. In *Grid Computing: Software environments and Tools*, J. C. Cunha and O. F. Rana, Eds. Springer, jan 2006, ch. 10, pp. 230–256.

[10] Aldinucci, M., and Danelutto, M. Stream parallel skeleton optimization. In *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems* (Cambridge, Massachusetts, USA, nov 1999), IASTED, ACTA press, pp. 955–962.

[11] Aldinucci, M., and Danelutto, M. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures 33*, 3-4 (oct 2007), 179–192.

[12] Aldinucci, M., Danelutto, M., and Kilpatrick, P. Towards hierarchical management of autonomic components: a case study. In *Proc. of Intl. Euromicro PDP 2009: Parallel Distributed and network-based Processing* (Weimar, Germany, feb 2009), F. S. Didier El Baz, Tom Gross, Ed., IEEE, pp. 3–10.

[13] Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., and Torquati, M. Accelerating sequential programs using FastFlow and self-offloading. Tech. Rep. TR-10-03, Università di Pisa, Dipartimento di Informatica, Italy, feb 2010.

[14] Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., and Torquati, M. An efficient unbounded lock-free queue for multi-core systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing* (Rhodes Island, Greece, aug 2012), vol. 7484 of *LNCS*, Springer, pp. 662–673.

[15] Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. Fastflow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*, S. Pllana and F. Xhafa, Eds., Parallel and Distributed Computing. Wiley, 2013, ch. 13.

[16] Aldinucci, M., Danelutto, M., and Teti, P. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems 19*, 5 (jul 2003), 611–626.

[17] Aldinucci, M., Meneghin, M., and Torquati, M. Efficient Smith-Waterman on multi-core with fastflow. In *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing* (Pisa, Italy, feb 2010), M. Danelutto, T. Gross, and J. Bourgeois, Eds., IEEE, pp. 195–199.

[18] Aldinucci, M., Tordini, F., Drocco, M., Torquati, M., and Coppo, M. Parallel stochastic simulators in system biology: the evolution of the species. In *Proc. of Intl. Euromicro PDP 2013: Parallel Distributed and network-based Processing* (Belfast, Nothern Ireland, U.K., feb 2013), IEEE.

[19] Aldinucci, M., and Torquati, M. *FastFlow website*, June 2013 (last accessed). `http://mc-fastflow.sourceforge.net/`.

[20] Aldinucci, M., Torquati, M., Spampinato, C., Drocco, M., Calcagno, C., and Coppo, M. Parallel stochastic systems biology in the cloud. *Briefings in Bioinformatics* (2013). Submitted for pubblication.

[21] Alur, R., Belta, C., and Ivancic, F. Hybrid modeling and simulation of biomolecular networks. In *Proc. of the 4th International Workshop on Hybrid Systems: Computation and Control (HSCC), Rome, Italy* (2001), vol. 2034 of *LNCS*, Springer, pp. 19–32.

[22] Aman, B., Dezani-Ciancaglini, M., and Troina, A. Type Disciplines for Analysing Biologically Relevant Properties. *Electr. Notes Theor. Comput. Sci. 227* (2009), 97–111.

[23] Amazon.com, Inc. *Amazon Elastic Compute Cloud website*, June 2013 (last accessed). `http://aws.amazon.com/ec2/`.

[24] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proc. of the April 18-20, 1967, spring joint computer conference* (New York, NY, USA, 1967), ACM, pp. 483–485.

[25] Apache Software Foundation. *Hadoop*, June 2013 (last accessed). `http://hadoop.apache.org`.

[26] Arkin, A., Ross, J., and McAdams, H. Stochastic kinetic analysis of developmental pathway bifurcation in phage λ-infected escherichia coli cells. *Genetics 149*, 4 (1998), 1633.

[27] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. A view of the parallel computing landscape. *Communications of the ACM 52*, 10 (2009), 56–67.

[28] Barbuti, R., Maggiolo-Schettini, A., and Milazzo, P. Extending the Calculus of Looping Sequences to Model Protein Interaction at the Domain Level. In *ISBRA* (2007), vol. 4463 of *LNCS*, Springer, pp. 638–649.

[29] Barbuti, R., Maggiolo-Schettini, A., Milazzo, P., Tiberi, P., and Troina, A. Stochastic calculus of looping sequences for the modelling and simulation of cellular pathways. *Transactions on Computational Systems Biology IX* (2008), 86–113.

[30] Barbuti, R., Maggiolo-Schettini, A., Milazzo, P., and Troina, A. A calculus of looping sequences for modelling microbiological systems. *Fundam. Inform. 72*, 1-3 (2006), 21–35.

[31] Barnat, J., Brim, L., Cerna, I., and et al. Biodivine: a framework for parallel analysis of biological models. *Electronic Proceedings in Theoretical Computer Science 6* (2009), 31–45.

[32] Barnat, J., Brim, L., and Safránek, D. High-performance analysis of biological systems dynamics with the divine model checker. *Briefings in Bioinformatics 11*, 3 (2010), 301–312.

[33] Bayer. *Computational Systems Biology Software Suite website*, June 2013 (last accessed). `http://www.systems-biology.com/products.html`.

[34] Bischof, H., Gorlatch, S., and Leshchinskiy, R. DatTel: A data-parallel C++ template library. *Parallel Processing Letters 13*, 3 (2003), 461–472.

[35] Boost. *Boost website*, June 2013 (last accessed). `http://www.boost.org/`.

[36] Bortolussi, L., and Vigliotti, M. G. CoBiC: Context-dependent Bioambient Calculus. *Electr. Notes Theor. Comput. Sci. 253*, 3 (2009), 187–201.

[37] Brijder, R., Cavaliere, M., Riscos-Núñez, A., Rozenberg, G., and Sburlan, D. Membrane systems with proteins embedded in membranes. *Theor. Comput. Sci. 404*, 1-2 (2008), 26–39.

[38] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. Brook for gpus: stream computing on graphics hardware. In *ACM SIGGRAPH '04 Papers* (New York, NY, USA, 2004), ACM Press, pp. 777–786.

[39] Cappello, F., and Etiemble, D. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Proc. of the 2000 ACM/IEEE conference on Supercomputing (CDROM)* (2000), Supercomputing '00, IEEE Computer Society.

[40] Cardelli, L. Brane Calculi. In *CMSB* (2004), vol. 3082 of *LNCS*, Springer, pp. 257–278.

[41] Cavaliere, M., and Sedwards, S. Decision problems in membrane systems with peripheral proteins, transport and evolution. *Theor. Comput. Sci. 404*, 1-2 (2008), 40–51.

[42] Chabrier-Rivier, N., Chiaverini, M., Danos, V., Fages, F., and Schächter, V. Modeling and querying biomolecular interaction networks. *Theor. Comput. Sci. 325*, 1 (2004), 25–44.

[43] Chen, Y., Lawless, C., Gillespie, C. S., Wu, J., Boys, R. J., and Wilkinson, D. J. Calibayes and basis: integrated tools for the calibration, simulation and storage of biological simulation models. *Briefings in bioinformatics 11*, 3 (2010), 278–289.

[44] Ciocchetta, F., and Hillston, J. Bio-PEPA: An Extension of the Process Algebra PEPA for Biochemical Networks. *ENTCS 194*, 3 (2008), 103–117.

[45] Ciocchetta, F., and Hillston, J. Bio-pepa: An extension of the process algebra pepa for biochemical networks. In *Proc. of 1st Workshop "From Biology To Concurrency and back (FBTC), Lisbon, Portugal* (2008), vol. 194 of *Electr. Notes Theor. Comput. Sci.*, Elsevier, pp. 103–117.

[46] Cole, C., and Herlihy, M. Snapshots and software transactional memory. *Sci. Comput. Program. 58*, 3 (2005), 310–324.

[47] Cole, M. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing 30*, 3 (2004), 389–406.

[48] Cole, M. *Skeletal Parallelism home page*, June 2013 (last accessed). `http://homepages.inf.ed.ac.uk/mic/Skeletons/`.

[49] Coppo, M., Damiani, F., Drocco, M., Grassi, E., Sciacca, E., Spinella, S., and Troina, A. Hybrid Calculus of Wrapped Compartments. In *4th International Meeting on Membrane Computing and Biologically Inspired Process Calculi (MeCBIC'10)* (2010), vol. 40, EPTCS, pp. 102–120.

[50] Coppo, M., Damiani, F., Drocco, M., Grassi, E., Sciacca, E., Spinella, S., and Troina, A. Simulation techniques for the calculus of wrapped compartments. *Theoretical Computer Science 431*, 0 (2012), 75 – 95. Modelling and Analysis of Biological Systems Based on papers presented at the Workshop on Membrane Computing and Bio-logically Inspired Process Calculi (MeCBIC) held in 2008 (Iasi), 2009 (Bologna) and 2010 (Jena).

[51] Coppo, M., Damiani, F., Drocco, M., Grassi, E., and Troina, A. Stochastic Calculus of Wrapped Compartments. In *Proceedings Eighth Workshop on Quantitative Aspects of Programming Languages (QAPL), Paphos, Cyprus* (2010), vol. 28, EPTCS, pp. 82–98.

[52] Curti, M., Degano, P., Priami, C., and Baldari, C. T. Modelling biochemical pathways through enhanced pi-calculus. *Theor. Comput. Sci. 325*, 1 (2004), 111–140.

[53] Danos, V., Feret, J., Fontana, W., and Krivine, J. Scalable Simulation of Cellular Signaling Networks. In *APLAS* (2007), vol. 4807 of *LNCS*, Springer, pp. 139–157.

[54] Danos, V., and Laneve, C. Formal molecular biology. *Theor. Comput. Sci. 325*, 1 (2004), 69–110.

[55] Darlington, J., Field, A. J., Harrison, P., Kelly, P. H. J., Sharp, D. W. N., While, R. L., and Wu, Q. Parallel programming using skeleton functions. In *Proc. of Parallel Architectures and Langauges Europe (PARLE'93)* (Munich, Germany, June 1993), vol. 694 of *LNCS*, Springer, pp. 146–160.

[56] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *Usenix OSDI '04* (Dec. 2004), pp. 137–150.

[57] Degano, P., Prandi, D., Priami, C., and Quaglia, P. Beta-binders for biological quantitative experiments. *Electr. Notes Theor. Comput. Sci. 164*, 3 (2006), 101–117.

[58] Dematté, L., Priami, C., and Romanel, A. The beta workbench: a computational tool to study the dynamics of biological systems. *Briefings in Bioinformatics 9*, 5 (2008), 437–449.

[59] Dezani-Ciancaglini, M., Giannini, P., and Troina, A. A Type System for a Stochastic CLS. In *Proc. of 4th Workshop on Membrane Computing and Biologically Inspired Process Calculi (MeCBIC), Bologna, Italy* (2009), vol. 11, EPTCS, pp. 91–105.

[60] Dhar, P. K., and et al. Grid cellware: the first grid-enabled tool for modelling and simulating cellular processes. *Bioinformatics 7* (2005), 1284–1287.

[61] Dittamo, C., and Cangelosi, D. Optimized parallel implementation of Gillespie's first reaction method on graphics processing units. In *Proc. of Intl. Conference on Computer Modeling and Simulation, (ICCMS)* (Macau, China, Feb. 2009), IEEE, pp. 156–161.

[62] Drawert, B., Engblom, S., and Hellander, A. Urdme: a modular framework for stochastic simulation of reaction-transport processes in complex geometries. *BMC systems biology 6*, 1 (2012), 76.

[63] Eissing, T., Kuepfer, L., Becker, C., Block, M., Coboeken, K., Gaub, T., Goerlitz, L., Jaeger, J., Loosen, R., Ludewig, B., et al. A computational systems biology software platform for multiscale modeling and simulation: integrating whole-body physiology, disease biology, and molecular reaction networks. *Frontiers in Physiology 2* (2011).

[64] Elowitz, M., Levine, A., Siggia, E., and Swain, P. Stochastic gene expression in a single cell. *Science 297*, 5584 (2002).

[65] Ferscha, A. *Performance Models for Discrete Event Systems with Synchronisations: Formalisms and Analysis Techniques*, vol. 2 of *MATCH Advanced Schools*. Jaca, Spain, Sept. 1998, ch. VII – Simulation.

[66] Flynn, M. J. Very high-speed computing systems. *Proceedings of the IEEE 54*, 12 (1966), 1901–1909.

[67] Giavitto, J.-L. Invited Talk: Topological Collections, Transformations and Their Application to the Modeling and the Simulation of Dynamical Systems. In *RTA* (2003), vol. 2706 of *Lecture Notes in Computer Science*, Springer, pp. 208–233.

[68] Gillespie, D. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem. 81* (1977), 2340–2361.

[69] González-Vélez, H., and Leyton, M. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software: Practice and Experience 40*, 12 (2010), 1135–1160.

[70] Gruenert, G., Ibrahim, B., Lenser, T., Lohel, M., Hinze, T., and Dittrich, P. Rule-based spatial modeling with diffusing, geometrically constrained molecules. *BMC bioinformatics 11*, 307 (2010).

[71] Hartigan, J., and Wong, M. A k-means clustering algorithm. *JR Stat. Soc., Ser. C 28* (1979), 100–108.

[72] Hasty, J., Pradines, J., Dolnik, M., and Collins, J. Noise-based switches and amplifiers for gene expression. *Proceedings of the National Academy of Sciences 97*, 5 (2000), 2075.

[73] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, fifth ed. Elsevier, 2011.

[74] Heyer, L., Kruglyak, S., and Yooseph, S. Exploring expression data: identification and analysis of coexpressed genes. *Genome research 9*, 11 (1999), 1106.

[75] Intel Corp. *Intel ® C++ Intrinsics Reference*, June 2013 (last accessed). `http://software.intel.com/en-us/avx/`.

[76] Intel Corp. *Threading Building Blocks website*, June 2013 (last accessed). `http://www.threadingbuildingblocks.org/`.

[77] Intosalmi, J., Manninen, T., Ruohonen, K., and Linne, M. Computational study of noise in a large signal transduction network. *BMC bioinformatics 12*, 252 (2011).

[78] Khronos Compute Working Group. *OpenCL website*, June 2013 (last accessed). `http://www.khronos.org/opencl/`.

[79] Klingbeil, G., Erban, R., Giles, M., and Maini, P. Stochsimgpu: parallel stochastic simulation for the systems biology toolbox 2 for matlab. *Bioinformatics 27*, 8 (2011), 1170.

[80] Krivine, J., Milner, R., and Troina, A. Stochastic Bigraphs. *Electron. Notes Theor. Comput. Sci. 218* (2008), 73–96.

[81] Lamport, L. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst. 5*, 2 (1983), 190–222.

[82] Langmead, B., Hansen, K. D., Leek, J. T., et al. Cloud-scale rna-sequencing differential expression analysis with myrna. *Genome Biol 11*, 8 (2010), R83.

[83] Langmead, B., Schatz, M. C., Lin, J., Pop, M., and Salzberg, S. L. Searching for snps with cloud computing. *Genome Biol 10*, 11 (2009), R134.

[84] Leloup, J., Gonze, D., and Goldbeter, A. Limit cycle models for circadian rhythms based on transcriptional regulation in drosophila and neurospora. *Journal of Biological Rhythms 14*, 6 (1999), 433.

[85] Li, H., and Homer, N. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics 11*, 5 (2010), 473–483.

[86] Li, H., and Petzold, L. Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on the graphics processing unit. *Int. J. High Perform. Comput. Appl. 24*, 2 (May 2010), 107–116.

[87] Ma, H., Boogerd, F., and Goryanin, I. Modelling nitrogen assimilation of escherichia coli at low ammonium concentration. *Journal of biotechnology 144*, 3 (2009), 175–183.

[88] Matsumoto, M., and Nishimura, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 1 (1998), 3–30.

[89] Matsunaga, A., Tsugawa, M., and Fortes, J. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on* (2008), IEEE, pp. 222–229.

[90] Matsuno, H., Doi, A., Nagasaki, M., and Miyano, S. Hybrid Petri net representation of gene regulatory network. In *Prooceedings of Pacific Symposium on Biocomputing* (2000), World Scientific Press, pp. 341–352.

[91] Michel, O., Giavitto, J.-L., Cohen, J., and Spicher, A. MGS home page. `http://mgs.spatial-computing.org/`, June 2013 (last accessed).

[92] Michel, O., Spicher, A., and Giavitto, J.-L. Rule-based programming for integrative biological modeling. *Natural Computing 8*, 4 (2009), 865–889.

[93] Miras, H., Jiménez, R., Miras, C., and Gomà, C. Cloudmc: a cloud computing application for monte carlo simulation. *Physics in medicine and biology 58*, 8 (2013), N125.

[94] NVIDIA Corp. *CUDA website*, June 2013 (last accessed). `http://www.nvidia.com/object/cuda_home_new.html`.

[95] Oury, N., and Plotkin, G. Multi-Level Modelling via Stochastic Multi-Level Multiset Rewriting. Draft submitted to MSCS, 2011.

[96] Pacheco, P. S. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[97] Park, I., Voss, M. J., Kim, S. W., and Eigenmann, R. Parallel programming environment for OpenMP. *Scientific Programming 9* (2001), 143–161.

[98] Parzen, E. *Stochastic Processes*. Holden-Day, 1962.

[99] Pescini, D., Besozzi, D., Mauri, G., and Zandron, C. Dynamical probabilistic P systems. *Int. J. Found. Comput. Sci. 17*, 1 (2006), 183–204.

[100] Petzold, L. *StochKit: stochastic simulation kit web page*, June 2013 (last accessed). `http://www.engineering.ucsb.edu/~cse/StochKit/index.html`.

[101] Phillips, A., and Cardelli, L. A correct abstract machine for the stochastic pi-calculus. In *Proc. of BIOCONCUR, London, England* (2004), Electr. Notes Theor. Comput. Sci.

[102] Poldner, M., and Kuchen, H. Scalable farms. In *Proc. of Intl. PARCO 2005: Parallel Computing* (Malaga, Spain, Sept. 2005).

[103] Priami, C., Regev, A., Shapiro, E. Y., and Silverman, W. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Inf. Process. Lett. 80*, 1 (2001), 25–31.

[104] Păun, G. *Membrane computing. An introduction*. Springer, 2002.

[105] Ramsey, S., Orrell, D., and Bolouri, H. Dizzy: Stochastic simulation of large-scale genetic regulatory networks (supplementary material). *J. Bioinformatics and Computational Biology 3*, 2 (2005), 437–454.

[106] Ray, T., and Saini, P. Engineering design optimization using a swarm with an intelligent information sharing among individuals. *Eng. Opt. 33* (2001), 735–748.

[107] Regev, A., Panina, E. M., Silverman, W., Cardelli, L., and Shapiro, E. Y. BioAmbients: an abstraction for biological compartments. *Theor. Comput. Sci. 325*, 1 (2004), 141–167.

[108] Regev, A., and Shapiro, E. Cells as computation. *Nature 419* (2002), 343.

[109] Regev, A., Silverman, W., and Shapiro, E. Y. Representation and Simulation of Biochemical Processes Using the pi-Calculus Process Algebra. In *Pacific Symposium on Biocomputing* (2001), pp. 459–470.

[110] Ropella, G., and Hunt, C. A. Cloud computing and validation of expandable in silico livers. *BMC systems biology 4*, 1 (2010), 168.

[111] Salis, H., Sotiropoulos, V., and Kaznessis, Y. Multiscale hy3s: hybrid stochastic simulation for supercomputers. *BMC bioinformatics 7*, 93 (2006).

[112] Savitzky, A., and Golay, M. Smoothing and differentiation of data by simplified least squares procedures. *Analytical chemistry 36*, 8 (1964), 1627–1639.

[113] Schatz, M. C. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics 25*, 11 (2009), 1363–1369.

[114] Schatz, M. C., Langmead, B., and Salzberg, S. L. Cloud computing and the dna data race. *Nature biotechnology 28*, 7 (2010), 691.

[115] Schlögl, F. Chemical reaction models for non-equilibrium phase transitions. *Zeitschrift für Physik A Hadrons and Nuclei 253*, 2 (1972), 147–161.

[116] Serot, J. Tagged-token data-flow for skeletons. *Parallel Processing Letters 11*, 4 (2001), 377–392.

[117] Sevior, M., Fifield, T., and Katayama, N. Belle monte-carlo production on the amazon ec2 cloud. In *Journal of Physics: Conference Series* (2010), vol. 219, IOP Publishing, p. 012003.

[118] Shendure, J., and Ji, H. Next-generation dna sequencing. *Nature biotechnology 26*, 10 (2008), 1135–1145.

[119] Singer, S., and Nicolson, G. The fluid mosaic model of the structure of cell membranes. *Science 175* (1972), 720–731.

[120] Sourceforge website. *CWC Simulator project*, June 2013 (last accessed). `http://sourceforge.net/projects/cwcsimulator/`.

[121] Spicher, A., Michel, O., Cieslak, M., Giavitto, J.-L., and Prusinkiewicz, P. Stochastic P systems and the simulation of biochemical processes with dynamic compartments. *Biosystems 91*, 3 (2008), 458–472.

[122] Stanford Univeristy. *Folding@home website*, June 2013 (last accessed). `http://folding.stanford.edu/English/HomePage`.

[123] Stein, L. D., et al. The case for cloud computing in genome informatics. *Genome Biol 11*, 5 (2010), 207.

[124] Thies, W., Karczmarek, M., and Amarasinghe, S. P. StreamIt: A language for streaming applications. In *Proc. of the 11th Intl. Conference on Compiler Construction (CC)* (London, UK, 2002), Springer, pp. 179–196.

[125] University of California, Santa Barbara. *StochSS: Stochastic Simulation Service. A Cloud Computing Framework for Modeling and Simulation of Stochastic Biochemical Systems website*, June 2013 (last accessed). `http://iguana.cs.ucsb.edu/wordpress/`.

[126] Vanneschi, M. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing 28*, 12 (dec 2002), 1709–1732.

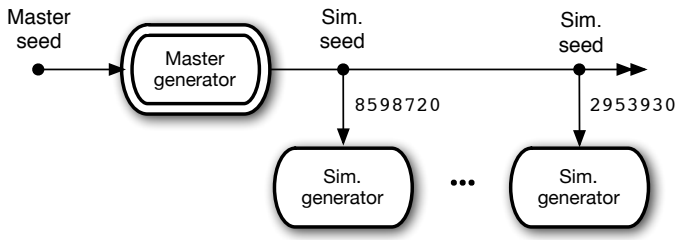
[127] ZeroMQ. *ZeroMQ website*, June 2013 (last accessed). `http://www.zeromq.org/`.

[128] Zou, Q., Li, X.-B., Jiang, W.-R., Lin, Z.-Y., Li, G.-L., and Chen, K. Survey of mapreduce frame operation in bioinformatics. *Briefings in bioinformatics* (2013).

A Randomness

Gillespie SSA, as any Monte Carlo method, is based on the generation of approximately-random sequences of numbers, fitting known probability distributions. Moreover, simulation instances should be *repricable* (this is crucial for accurate performance evaluation). This aspects are usually addressed by using a PRNG (Pseudo-Random Number Generator), an algorithm that, given an initial state (a.k.a. the *seed*), outputs a *deterministic* sequence, completely identified by the seed. Requirements for a suitable PRNG include good statistical properties and a long *period* (the maximum length of the generated sequence after which the initial state is re-encountered, so the sequence starts to repeat).

In order to obtain statistically meaningful results from the reduction of the outcomes of different simulation instances, the instances must be *statistically independent*. Making the instances sharing the same PRNG is not a feasible solution for efficient parallel computing, thus each instance needs a private generator. In the implemented tool, the generators are initialised according to the following schema:



Clearly the master generator should be different – i.e. it should generate different sequences – from the other generators, otherwise the sequences would overlap. This both guarantees the instances are statistically independent (if generators are properly chosen) and makes the experiments fully replicable, since a master seed induces a unique set of simulation instances.

With the above considerations in mind, we used the MT19937 Mersenne Twister PRNG [88] for the simulation instances, which exhibits some nice features:

- it's fast
- it passes various tests for statistical randomness
- it has a (very long) period of $2^{19937} - 1$.

In particular, we used the Boost [35] implementation of MT19937, now included in C++11. This implementation is "parallel-safe", in the sense that different instances of the generator don't share global structures in memory – actually they act as if they were "in insulation" – making it suitable to support concurrent simulation instances.