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

Marco Aldinucci, Maurizio Drocco, Fabio Tordini, Mario Coppo
Computer Science Department
University of Torino, Italy
Email: {aldinuc, tordini, coppo}@di.unito.it, maurizio.drocco@gmail.com

Massimo Torquati
Computer Science Department
University of Pisa, Italy
Email: torquati@di.unipi.it

***Abstract*—The stochastic simulation of biological systems is an increasingly popular technique in Bioinformatics. It is often an enlightening technique, especially for multi-stable systems which 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 in model tuning. In this work we advocate the high-level design of simulators for stochastic systems as a vehicle for building efficient and portable parallel simulators. In particular, the Calculus of Wrapped Components (CWC) simulator, which is designed according to the FastFlow's pattern-based approach, is presented and discussed in this work. FastFlow has been extended to support also clusters of multi-cores with minimal coding effort, assessing the portability of the approach.**

***Keywords*-Parallel patterns; multi-core; distributed computing; stochastic simulation; systems biology.**

I. 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 [1], has been supported by a growing number of formalisms aiming to describe biological systems as stochastic models [2].

The stochastic modelling 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.

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 machines. This approach naturally couples with the distributed execution of a batch of tasks that requires large infrastructures (e.g. grids, clouds) and suffers from slow time-to-solution as the single experiment require to enqueue the simulations in shared environment, deploy initial data, simulate the model, gather results from a distributed environment, post-process them (often sequentially), then eventually access results. This process is typically repeated several time to fine-tune initial conditions and simulation parameters. In the meanwhile, the entire hardware industry has moved to multi-core, which nowadays equips the large majority of computing platforms, included ones constituting large computing grids and clouds. 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 multi-core 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. This 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 multi-core platforms, i.e. memory bandwidth and core synchronisations;
2) the "sequentialisation" 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 simulator is often specifically optimised for a specific parallel platform, either multi-core or distributed (or not optimised at all).

This paper presents a critical rethinking of the parallelisation of stochastic processes in the light of hybrid multi-core and distributed platforms and the tools that

are required to derive an efficient simulator from both performance and 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 multi-core platforms and large distributed execution environments.

In this work, the simulator for the Calculus of Wrapped Compartments (CWC) will be used as test-bed. CWC is a recently proposed formalism, based on term rewriting, for the representation of biological systems [3]. The CWC simulator [4], which currently targets multi-core platforms, has been designed exploiting a high-level methodology based on parallel patterns and considering the whole simulation workflow: from simulation to on-line data analysis and mining. This latter aspect provides bioinformatics scientists with immediate feedback on simulation results and their main statistic estimators while the simulation is still running, thus with an early feedback on simulation effectiveness.

In this paper, the very same design is extended to a hybrid distributed and multi-core execution environment. This work takes advantage from the extension to distributed platforms of the high-level programming framework used to develop the simulator itself, i.e. the FastFlow framework [5]. As result, the extension of the CWC simulator itself has required no substantial changes to the simulation and result analysis code, and involved a very limited coding effort. Notwithstanding, the new version keeps the same features and provides the user with same strict feedback on simulation results, by way of the on-line (parallel) analysis of (parallel) simulation results.

II. Background and related work

The related work is approached from two different perspectives: the engineering of tools for parallel computing (Sec. II-A) and the formalisms and tools for systems biology (Sec. II-B).

A. High-level development tools

Programming tools and frameworks are needed to efficiently target the architectures hosting inter networked – possibly heterogeneous – multi-core devices, which appear to be *the* reference architecture ferrying programmers from the mainly sequential to mainly parallel programming era [6]. The urgency is even more crucial given that both grids and clouds provide application programmers with the possibility to reserve collections of multi-cores to support parallel applications eventually presented as (or orchestrated by) web services.

Shared memory multi-cores and clusters/networks of processing elements, however, require quite different techniques and tools to support efficient parallelism exploitation. The *de facto* standard tools in the two cases are OpenMP [7] and MPI [8], 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 symmetric multiprocessing (SMP) systems [9]. 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. In order to reduce complexity and simultaneously providing the user with a higher level of abstraction, algorithmic skeletons have been proposed [10]. They provide predefined parallel computation and communication patterns, hiding the parallelism management to the user.

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 [10], [11]. 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 multi-cores may also be exploited. SkeTo provides data parallel skeletons as proper C++ abstractions. It has been recently extended to target multi-core clusters using a two-step dynamic task scheduling strategy, which enables balancing of the load both between nodes of the cluster and cores of the single node [12]. The Muesli programming framework is provided as a C++ library offering both data and stream parallel skeletons. The original version of Muesli generated code for MPI platforms. Recently, Muesli has been extended in such a way that both multi-core architectures and distributed multi-core workstations may be targeted by generating OpenMP code, in addition to the MPI code [13].

As mentioned, data produced by a Monte Carlo simulator, as well as from other hardware tools from systems biology, can be regarded as a stream. Several languages and libraries are available for programming stream applications. Some of them, e.g. NVidia *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* [14], [15], *StreamIt* [16], *Brook* [17]). 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), and FastFlow, which will be discussed in Sec. III.

B. Calculi for Bioinformatics and Simulators

In the field of biological modelling, tools such as SPiM [18] and Dizzy [19] 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 [20] 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 distributed architectures. Our work differs from these efforts in three aspects: 1) it addresses multicore-specific parallelisation issues; 2) it advocates a general parallelisation schema rather than a specific simulator; 3) it addresses the on-line data analysis, thus it is designed to manage large streams of data. To the best of our knowledge, many related works cover some of these aspects, but few of them address all three aspects.

The Swarm algorithm [21], which is well suited for biochemical pathways optimisation, has been used in a distributed environment – e.g., in Grid Cellware [22], 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.

Parameter Sweep Applications (PSAs) execute the same piece of code multiple times, with unique sets of input parameters. However, since the instances of a PSA are independent, this type of application is extremely amenable to parallelisation in a distributed computing paradigm, that allows to sample a large space of independent instances. In [23], a grid-based version of a multi-volume stochastic simulator is presented for Parameter Sweep Applications (PSAs).

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 [24].

StochKit [25] is a C++ stochastic simulation framework. Among other methods, it implements the Gillespie algorithm and in its second version it targets multi-core 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 [26] 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 characterise the intrinsic noise of the network has been done as a post-processing step. We believe that our parallelisation framework could further improve those kinds of analyses.

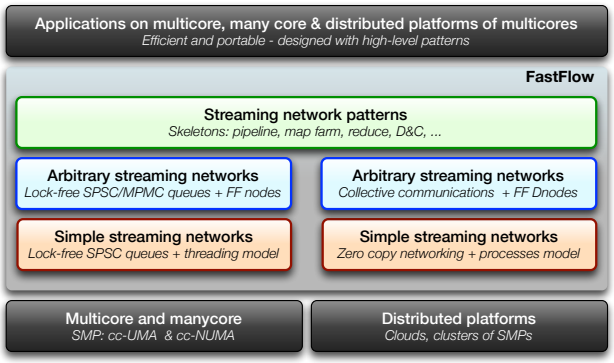

Figure 1. Layered FastFlow design.

SRSim [27] performs rule-based spatial modelling, which are embarrassingly parallelised with MPI. In this case, high latency and nuisances with the connection of the computing clusters could decrease the speed efficiency.

In [28], the authors propose an adaptation for GPGPUs of the Gillespie's 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-optimised 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 synchronisation mechanisms that are missing in GPGPUs.

StochSimGPU [29] 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.

As a matter of fact, GPGPUs are becoming more and more used in scientific and high-performance computing. GPGPUs can provide relevant speedups for applications that suit the SIMD class of parallelisation, but they still need a lot of work on tuning the algorithm to the underlying architecture, in order to achieve a proper exploitation of the memory hierarchy. These tasks require a high level of expertise, and the learning curve to master this technology is rather steep.

III. The FastFlow programming framework

FastFlow [30] has been originally designed as a structured parallel programming environment for shared-memory multi-core platforms, and it has been recently extended to support distributed systems [31]. FastFlow provides programmers with predefined and customisable parallel design patterns – i.e. *farms*, *pipelines* and other stream-oriented parallel programming patterns. It has been

initially designed and implemented to be very efficient in the execution of fine grain parallel applications [5], [32].

FastFlow design is layered (see Fig. 1). The lower layer, called *simple streaming networks*, basically provide two basic abstractions:

- *process-component*, i.e. a control flow realized 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) [33] and zero-copy *ZeroMQ* channels [34] 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 2GHz [33]. ZeroMQ is an LGPL open-source communication library [34]. 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 allows 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).
- *collective channel*, i.e. a collective communication channel, either among `ff_nodes` or many `ff_dnodes`.

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`.

The predefined pattern 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

```
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. FastFlow's `ff_node` class schema

Table I
COLLECTIVE COMMUNICATION PATTERNS AMONG `ff_nodes`.

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

features, different patterns may be implemented, such as *Divide&Conquer*, *map* and *reduce* patterns. FastFlow is being extended to also support data parallel patterns and to offload data parallel computations to GPUs, where such device(s) are available.

In the current version (see Fig. 1), which supports distributed platforms, many graphs of `ff_nodes` can be connected by way of `ff_dnodes`, which supports network collective channels. The set of communication collectives allows exchange of messages among a set of distributed nodes, using well-known predefined patterns. The semantics of each communication pattern currently implemented are summarised in Table I.

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_dnodes`. 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.

IV. THE CWC MULTI-CORE SIMULATOR

A. The Calculus of Wrapped Compartments

The Calculus of Wrapped Compartments (CWC) [3], [4] has been designed to describe biological entities (like cells and bacteria) by means of a nested structure of ambients delimited by membranes.

The terms of the calculus are built on a set of *atoms* (representing species, i.e. molecules, proteins or DNA strands), ranged over by a, b, ..., and on a set of *labels* (representing compartment types, i.e. cells or tissues), ranged over by ℓ, ... A *term* is a multiset $\overline{t}$ of *simple terms*, where a simple term is either an atom a or a compartment $(\overline{a} \rfloor \overline{t'})^{\ell}$, consisting of a *wrap* (a multiset of atoms $\overline{a}$), a *content* (a term $\overline{t'}$) and a *type* (a label ℓ).

Multisets are denoted by listing the elements separated by a space. As usual, the notation $n * a$ denotes n occurrences of the atom a. For instance, the term $2 * a \ \ (b \ \ c \rfloor d \ \ e)^{\ell}$ represents a multiset containing two occurrences of the atom a and an ℓ-type compartment $(b \ \ c \rfloor d \ \ e)^{\ell}$ which consists of a wrap with two atoms b and c on its surface, and containing the atoms d and e[1].

Interactions between biological entities are described by rewriting rules written as $\ell : P \mapsto O$ where P and O are terms built on an extended set of atomic elements which includes variables (ranged over by X, Y, ...) and ℓ represents the compartment type to which the rule can be applied. An example of rewrite rule is $\ell : \ a \ b \ X \mapsto c \ X$, that is often written as $\ell : \ \ a \ \ b \mapsto c$, giving X for understood to simplify notations.[2] The application of a rule $\ell : P \mapsto O$ to a term $\overline{t}$ consists in finding (if it exists) a subterm $\overline{u}$ in a compartment of type ℓ such that $\overline{u} = \sigma(P)$ for a ground substitution σ, and replacing it with $\sigma(O)$ in $\overline{t}$. We write $\overline{t} \mapsto \overline{t'}$ to mean that $\overline{t'}$ can be obtained from $\overline{t}$ by applying a rewrite rule.

The standard way to model the time evolution of biological systems is that presented by Gillespie [1]. In Gillespie's algorithm a numerical rate is associated with each considered chemical reaction, which is used as the parameter of an exponential distribution modelling the probability that the reaction takes place. In the standard approach, this reaction rate is obtained by multiplying the kinetic constant of the reaction by the number of possible combinations of reactants that may occur in the region in which the reaction takes place, thus modelling the law of mass action. In this case, a stochastic rule is written as $\ell : P \stackrel{k}{\longmapsto} O$, where k represents the kinetic constant of the corresponding reaction. In CWC, a rewrite rule is associated with a *rate function* 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 Michaelis-Menten kinetics.

[1]For uniformity we assume that the term representing the whole system is always a single compartment labelled $\top$ with an empty wrap.

[2]At the left-hand side of a rule, 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.

B. The CWC simulator

The CWC simulator [35] is an open source tool that implements Gillespie's algorithm on CWC terms. It handles CWC models with different rating semantics (law of mass action, Michaelis-Menten kinetics, Hill equation) and it can run independent stochastic simulations. The CWC simulator was designed using the FastFlow high-level methodology and targets multi-core platforms. It exploits both parallel simulation and data analysis in a single workflow. To make it possible, all the logical phases of the process (i.e. data distribution, parallel simulations, result gathering, parallel trajectory, data assembling and analysis) must be effectively pipelined. This implies that all phases work on a data streams.

The simulation workflow, sketched in Fig. 3, is composed of a three-stage pipeline: simulation, analysis, and display of results. The former two stages are in turn pipelines, whereas the display of results is realised by way of a Graphical User Interface (GUI). In the picture, all 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 term of FastFlow objects (i.e. farm and pipelines), and filling white boxes with sequential code. All data is passed through memory references (no data copy).

1) The simulation pipeline: The simulation 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 [4].

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. This not necessarily represents the current status (at a given point in wall-clock time) of all running simulations. Stochastic processes 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 toward the next stage which sorts out all received results and aligns

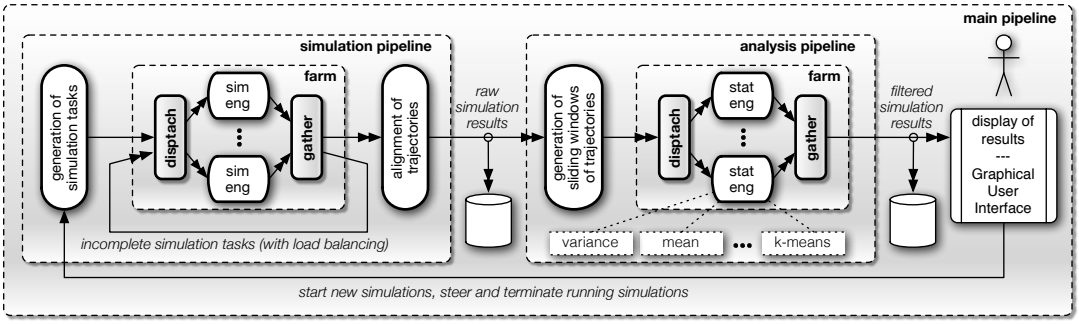

Figure 3. Architecture of the CWC multi-core simulator with on-line parallel analysis, where solid boxes are `ff_nodes` and dashed boxes are FastFlow patterns. Double-headed arrows are streams; grey boxes are automatically generated by the FastFlow framework; white boxes hold user-defined code.

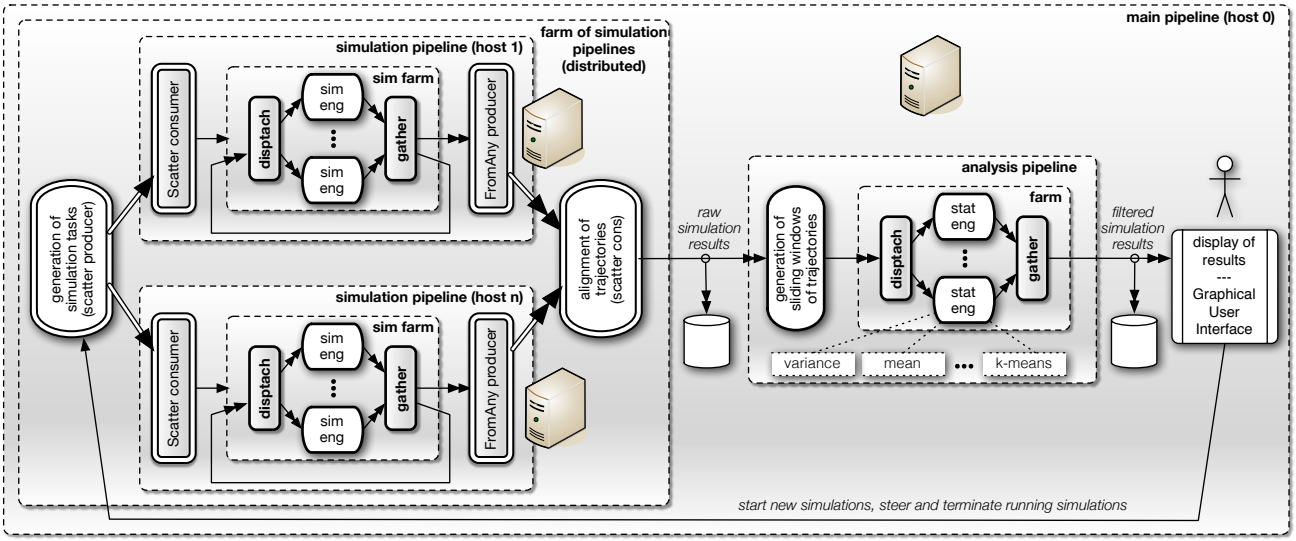

Figure 4. Architecture of the CWC distributed and multi-core simulator with on-line parallel analysis, where solid boxes are `ff_nodes`, double-stroked boxes are `ff_dnodes`, and dashed boxes are FastFlow patterns. Double-headed arrows are streams; grey boxes are automatically generated by the FastFlow framework; white boxes hold user-defined code. The distributed workflow is obtained by replicating the simulation pipeline of the multi-core version in Fig. 3 via the farm pattern.

them according to the simulation time. Once all simulation tasks overcome a given simulation time, an arrays 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.

2) The analysis pipeline: By design, each cut of simulation trajectories (i.e. an array of simulation results), can be analysed immediately and independently (thus concurrently) from 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 analysis, 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 it is possible to derive reasonable approximation of these analysis from a sliding window of the whole dataset. For this reason, stream incoming 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 eventually be 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 user interface and permanent storage [36].

3) The graphical user interface: The CWC simulation-analysis pipeline is wrapped in a back-end tool that can be steered either via command line tools or a graphical user interface, which makes it possible to design the biological model, run simulations and analysis and to view partial results during the run. Also, the front-end allows to control the simulation workflow from a remote machine.

C. The CWC distributed and multi-core simulator

Thanks to the high-level design of the CWC simulator and the extension of the FastFlow to distributed platforms [31], the porting of the CWC simulator to

distributed platforms has been possible with a very limited coding effort. The architecture of the CWC distributed simulator is shown in Fig. 4. It is clear that the design directly derives from the existing multi-core version, shown in Fig. 3, 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_nodes` to `ff_dnodes` to support network channels. Also, the simulation pipeline is extended with two fringe `ff_dnodes` 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.

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

V. Experimental evaluation

The evaluation of the CWC simulator takes into account the performance on both multi-cores and cluster of multi-cores. The ability of the CWC formalism to describe simple but significant biological systems, together with the effectiveness of the proposed on-line analysis to capture the behaviour of the system has been discussed in previous works [3], [36]. A single example of the output of the simulator is reported for the sake of completeness.

A. Simulation and analysis of multi-stable systems

One of the best studied examples of multi-stability in genetic systems is the bacteriophage λ life cycle [37]: the CWC framework can represent this kind of biological events. This process involves two different biological entities, the *phage* and the *bacterium*, which represent nested structures of ambient delimited by membranes. Moreover, a step of the process includes the integration of a strand of DNA in the circular molecule of DNA, which is the genome of E. Coli and, even if it is not delimited by physical membranes, it can be considered a functional compartment. 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, E. Coli, causing DNA in the head of the phage to be ejected through the tail into the cytoplasm of the bacterial cell. When this happens, it can enter into one of two alternative stages, called *lysogeny* and *lysis*. The lysogeny stage is a dormant stage, in which the phage inserts its DNA into the host's DNA and passively reproduces with the host. 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 cells. The decision between lysis and lysogeny can be thought of as a switching mechanism.

A simplified model for the bacteriophage was proposed in [38]. In their model, the gene cI expresses the λ repressor CI which dimerises (atom $CI2$) and binds to DNA (atom D) as a transcription factor at either of two binding sites. Binding of this transcription factor to one site enhances the transcription of CI (positive feedback), while binding to the other site represses transcription of CI (negative feedback). The CWC rules in this system are:

$$\top : CI\ CI \xmapsto{K_1} CI2$$
$$\top : CI2 \xmapsto{K_1} CI\ CI$$
$$\top : CI2\ D \xmapsto{K_2} DCI2$$
$$\top : DCI2 \xmapsto{K_2} CI2\ D$$
$$\top : CI2\ D \xmapsto{K_3} DCI2^*$$
$$\top : DCI2^* \xmapsto{K_3} CI2\ D$$
$$\top : DCI2\ CI2 \xmapsto{K_4} DCI2CI2$$
$$\top : DCI2CI2 \xmapsto{K_4} DCI2\ CI2$$
$$\top : DCI2\ P \xmapsto{k_t} DCI2\ P\ CI2\ CI2CI \xmapsto{k_d} \bullet$$

where the $DCI2$ and $DCI2^*$ complexes denote the binding to the two sites, and $DCI2CI2$ denotes binding to both sites. K_i are forward equilibrium constants, k_t is the protein synthesis rate, and k_d is the degradation rate. P represents the RNA polymerase assumed here to be constant, and two proteins per mRNA transcript were considered. 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 (see Fig. 5 left).

Figure 5 (right) shows the resulting clusters (gray circles) computed on-line using QT on the λ-phage 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.

B. Performance on multi-core and cluster of multi-cores

The performance of the simulator is tested on the bacteriophage λ life cycle, described in Sec. V-A. Experiments have been executed on two platforms:

- *Intel 32 cores*, an Intel workstation, equipped with 4 eight-core E7-4820 Nehalem (64 HyperThreads) @2.0GHz with 18MB L3 cache and 64 GBytes of main memory with Linux x86_64. The Nehalem processor uses HyperThreading with 2 contexts per core that share execution units.

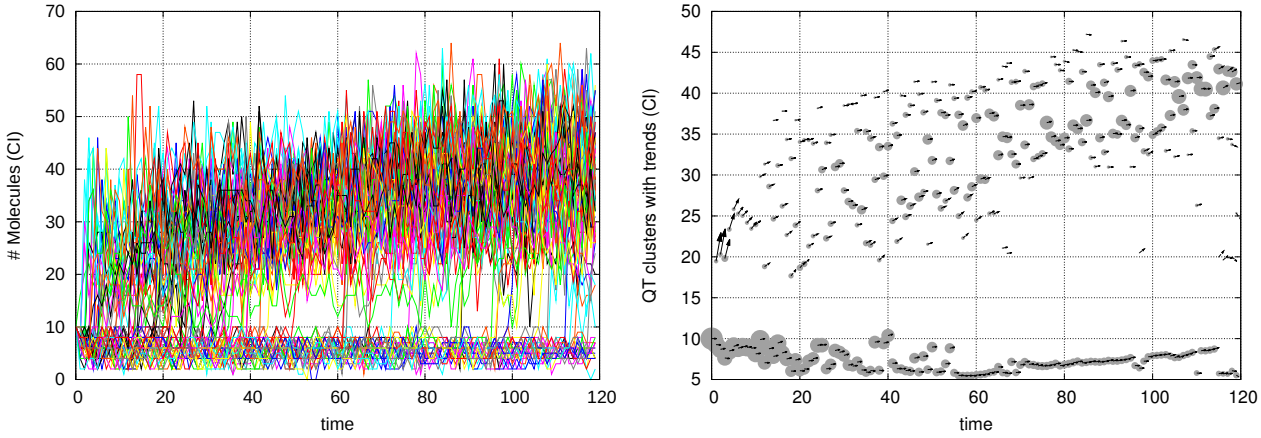

Figure 5. Simulation results on the λ-phage model. Left figure reports the 100 raw trajectories, right figure shows the on-line QT clustering results.

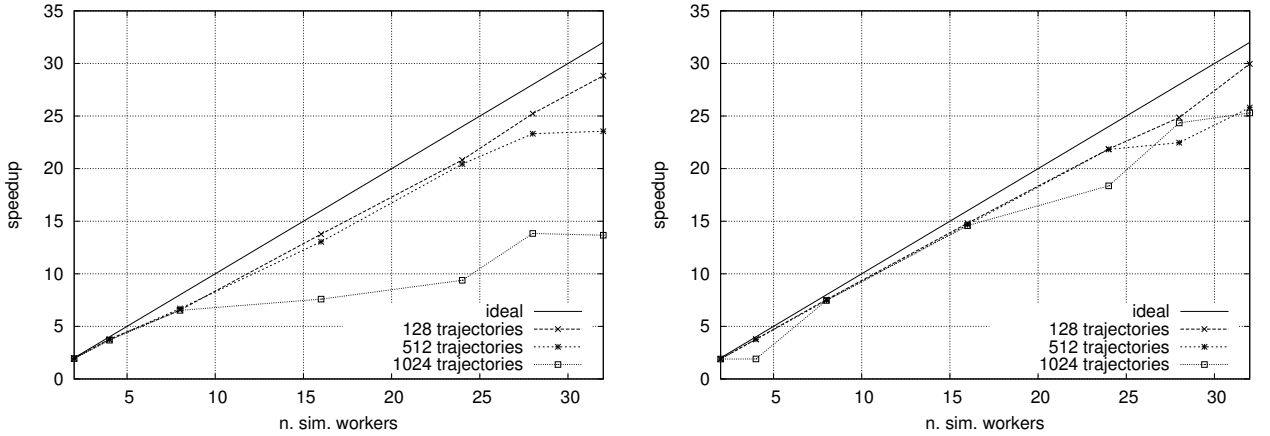

Figure 6. Speedup of the multi-core simulator on the bacteriophage λ life cycle on the Intel 32 cores platform with different number of simulations (trajectories). Left) using a single statistical engine. Right) using 4 statistical engines.

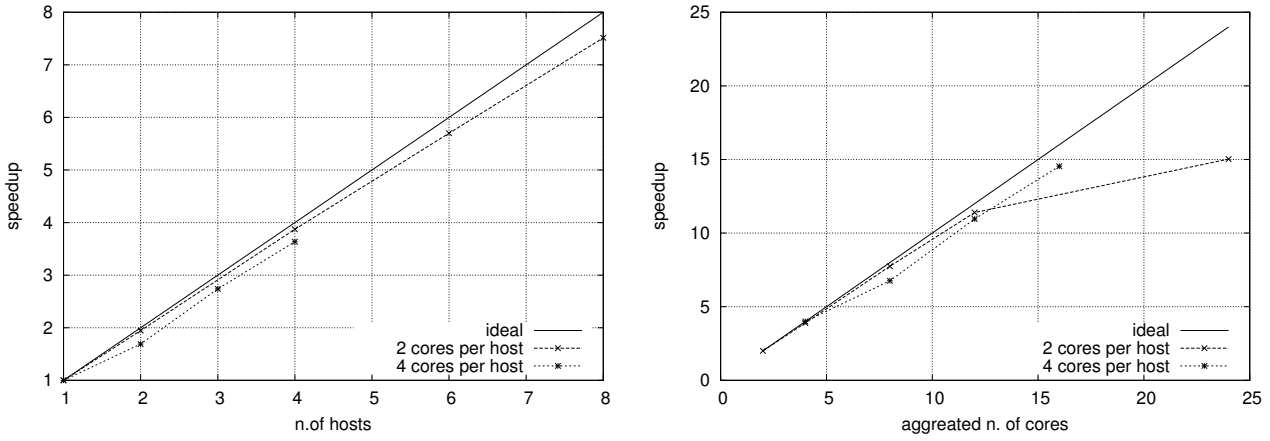

Figure 7. Speedup of the multi-core simulator on the bacteriophage λ life cycle on the Intel cluster of multi-cores with two different usage of cores per node. Left) w.r.t. the number of hosts. Right) w.r.t. aggregated number of cores.

- *Intel cluster of multi-cores*, a 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 speedup of the multi-core version of the simulator is shown in Fig. 6, against the number of simulation engines (one per core) in two configurations. Fig. 6 (left) uses a single statistical engine in the analysis pipeline, and Fig. 6 (right) 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 simulator exhibits an almost ideal speedup for limited data sets. 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 sections, the filtering and analysis of data has not a negligible cost for large data sets.

The speedup of the distributed version of the simulator is shown in the preliminary experiments in Fig. 7, against the number of hosts (left) and the aggregated count of cores in different hosts (right). In both cases 4 statistical engines are used. As shown in Fig. 7 (left), 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. 7 right), further experimentation is needed to tune message 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.

VI. Conclusions

We presented the design and the implementation of a distributed version of the CWC simulator, which is obtained with low engineering and coding efforts from the previous multi-core version. The distributed version maintains the good features of the previous one – i.e. scalability, automatic load balancing and fast feedback to the bioinformatics scientists – and can be executed in a cluster of multi-cores platforms. Preliminary experimental evaluations show that the design is flexible and robust with respect to target platform, and it is able to squeeze the full power of heterogeneous parallel platforms.

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.

VII. Acknowledgments

This work has been partially supported by the EC-FP7 STREP project "Paraphrase" (n. 288570) and the Fondazione San Paolo IMPACT project (id. ORTO11TPXK).

References

[1] D. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *J. Phys. Chem.*, vol. 81, pp. 2340–2361, 1977.

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

[3] M. Coppo, F. Damiani, M. Drocco, E. Grassi, E. Sciacca, S. Spinella, and A. Troina, "Simulation Techniques for the Calculus of Wrapped Compartments," *Theoretical Computer Science*, vol. 431, pp. 75–95, 2012.

[4] M. Aldinucci, M. Coppo, F. Damiani, M. Drocco, M. Torquati, and A. Troina, "On designing multicore-aware simulators for biological systems," in *Proc. of Intl. Euromicro PDP 2011: Parallel Distributed and network-based Processing*. Ayia Napa, Cyprus: IEEE, Feb. 2011, pp. 318–325.

[5] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing. Wiley, 2013, ch. 13.

[6] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

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

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

[9] F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks," in *Proc. of the 2000 ACM/IEEE conference on Supercomputing (SC)*. IEEE, 2000.

[10] M. Cole, "Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[11] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers," *Software: Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[12] Y. Karasawa and H. Iwasaki, "A parallel skeleton library for multi-core clusters," in *Proc. of the Intl. Conference on Parallel Processing (ICPP)*. Washington, DC, USA: IEEE, 2009, pp. 84–91.

[13] P. Ciechanowicz and H. Kuchen, "Enhancing muesli's data parallel skeletons for multi-core computer architectures," in *Proc. of the IEEE Intl Conference on High Performance Computing and Communications (HPCC)*. Washington, DC, USA: IEEE, 2010, pp. 108–113.

[14] M. Vanneschi, "The programming model of ASSIST, an environment for parallel and distributed portable applications," *Parallel Computing*, vol. 28, no. 12, pp. 1709–1732, Dec. 2002.

[15] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo, "A framework for experimenting with structure parallel programming environment design," in *Parallel Computing: Software Technology, Algorithms, Architectures and Applications*, ser. Advances in Parallel Computing, vol. 13. Elsevier, 2004, pp. 617–624.

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

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

[18] A. Phillips and L. Cardelli, "A correct abstract machine for the stochastic pi-calculus," in *Proc. of BIOCONCUR, London, England*, ser. Electr. Notes Theor. Comput. Sci., 2004.

[19] S. Ramsey, D. Orrell, and H. Bolouri, "Dizzy: Stochastic simulation of large-scale genetic regulatory networks (supplementary material)," *J. Bioinformatics and Computational Biology*, vol. 3, no. 2, pp. 437–454, 2005.

[20] F. Ciocchetta and J. Hillston, "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*, ser. Electr. Notes Theor. Comput. Sci., vol. 194, no. 3. Elsevier, 2008, pp. 103–117.

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

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

[23] E. Mosca, P. Cazzaniga, I. Merelli, D. Pescini, G. Mauri, and L. Milanesi, "Stochastic simulations on a grid framework for parameter sweep applications in biological models," *High Performance Computational Systems Biology, International Workshop on*, pp. 33–42, 2009.

[24] J. Barnat, L. Brim, and D. Safránek, "High-performance analysis of biological systems dynamics with the divine model checker," *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 301–312, 2010.

[25] L. Petzold, *StochKit: stochastic simulation kit web page*, 2009, http://www.engineering.ucsb.edu/~cse/StochKit/index.html.

[26] J. Intosalmi, T. Manninen, K. Ruohonen, and M. Linne, "Computational study of noise in a large signal transduction network," *BMC bioinformatics*, vol. 12, no. 252, 2011.

[27] G. Gruenert, B. Ibrahim, T. Lenser, M. Lohel, T. Hinze, and P. Dittrich, "Rule-based spatial modeling with diffusing, geometrically constrained molecules," *BMC bioinformatics*, vol. 11, no. 307, 2010.

[28] H. Li and L. Petzold, "Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on the graphics processing unit," *Int. J. High Perform. Comput. Appl.*, vol. 24, no. 2, pp. 107–116, May 2010.

[29] G. Klingbeil, R. Erban, M. Giles, and P. Maini, "Stochsimgpu: parallel stochastic simulation for the systems biology toolbox 2 for matlab," *Bioinformatics*, vol. 27, no. 8, p. 1170, 2011.

[30] *FastFlow website*, 2009. [Online]. Available: http://mc-fastflow.sourceforge.net/

[31] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, "Targeting distributed systems in fastflow," in *Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing*, ser. LNCS. Springer, 2013.

[32] M. Aldinucci, S. Ruggieri, and M. Torquati, "Porting decision tree algorithms to multicore using FastFlow," in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, ser. LNAI, vol. 6321. Barcelona, Spain: Springer, Sep. 2010, pp. 7–23.

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

[34] *ZeroMQ website*, 2012. [Online]. Available: http://www.zeromq.org/

[35] *CWC Simulator project*, Sourceforge website, 2010. [Online]. Available: http://sourceforge.net/projects/cwcsimulator/

[36] M. Aldinucci, M. Coppo, F. Damiani, M. Drocco, E. Sciacca, S. Spinella, M. Torquati, and A. Troina, "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)*, ser. LNCS, vol. 7155. Springer, 2012, pp. 3–12.

[37] A. Arkin, J. Ross, and H. McAdams, "Stochastic kinetic analysis of developmental pathway bifurcation in phage λ-infected escherichia coli cells," *Genetics*, vol. 149, no. 4, p. 1633, 1998.

[38] J. Hasty, J. Pradines, M. Dolnik, and J. Collins, "Noise-based switches and amplifiers for gene expression," *Proceedings of the National Academy of Sciences*, vol. 97, no. 5, p. 2075, 2000.