On Designing Multicore-Aware Simulators for Biological Systems

Marco Aldinucci*, Mario Coppo*, Ferruccio Damiani*, Maurizio Drocco*, Massimo Torquati†, and Angelo Troina*
*Computer Science Department, University of Torino, Italy
†Computer Science Department, University of Pisa, Italy

***Abstract*—The stochastic simulation of biological systems is an increasingly popular technique in bioinformatics. It often is an enlightening technique, which may however result in being computational expensive. We discuss the main opportunities to speed it up on multi-core platforms, which pose new challenges for parallelisation techniques. These opportunities are developed in two general families of solutions involving both the single simulation and a bulk of independent simulations (either replicas of derived from parameter sweep). Proposed solutions are tested on the parallelisation of the CWC simulator (Calculus of Wrapped Compartments) that is carried out according to proposed solutions by way of the FastFlow programming framework making possible fast development and efficient execution on multi-cores.**

***Index Terms*—multi-core; parallel simulation; stochastic simulation; SIMD; lock-free synchronisation;**

I. Introduction

Stochastic simulations are an increasingly popular technique to study biological systems. They – differently from other modelling approaches such as differential equations (ODEs) – are able to describe transient, and multi-stable behaviours of the systems. Different formalisms, based on automata models [1], process algebras [2], [3], or rewrite systems [4], [5] have either been applied to, or inspired from, biological systems. Quantitative simulations of biological models represented with these kinds of frameworks (e.g. [2], [6], [7]) are usually developed via a stochastic method derived by Gillespie's algorithm [8].

Among other formalisms, the Calculus of Wrapped Compartments (CWC) [9] is a recently proposed rewriting-based language for the representation and simulation of biological systems. It has been designed with the aim of simplifying the development of efficient implementations, while keeping the same expressiveness of other more complex languages.

Stochastic simulations are computationally more expensive than ODEs numerical solution. This is particularly true for the kind of systems that are better represented by stochastic models since, for their uneven nature, should be simulated at a very fine grain to spot possible spikes of the modelled phenomena along time, or to discriminate families of possible behaviour that are not revealed by the averaged behaviour described by ODEs. The high computational cost of stochastic simulation is well known and has led, in the last two decades, to a number of attempts to accelerate them up using several kind of techniques, such as approximate simulation algorithms and parallel computing [10]. In this work, this latter approach is taken into account.

Since stochastic simulations are basically Monte Carlo processes, many independent instances should be computed to achieve a statistically valid solution. These independent instances have been traditionally exploited in an *embarrassingly parallel* fashion, executing a partition of the instances in a different machine. This approach naturally couples with distributed computing (i.e. cluster, grid, clouds). However, the entire hardware industry has been moving to multi-core, which nowadays equips the large majority of computing platforms. 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 speed. Especially for I/O- and memory-bound applications since all the cores usually share the same memory and I/O subsystem.

The analysis of biological systems produces a large amount of data, often organised in streams coming from either analysis instruments or simulators. The management of these streams in 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 regards analysis of the simulation results, which requires the merging of results from different simulation instances and possibly their statistical filtering or mining. In distributed computing, this phase is often demoted to a secondary aspect in the computation and treated as with off-line post-processing tools. However, this approach is no longer realistic because of both 1) the ever-increasing size of produced data and, 2) it insists on the main weakness of multi-core platforms, i.e. memory bandwidth and core synchronisations.

In this paper we propose a critical rethinking of the parallelisation of stochastic processes in the light of emerging multi-core platforms and the tools that are required to derive an efficient simulator from both performance and easy engineering viewpoints. We believe that this latter aspect is of crucial importance for next generation biological tools because they will be largely designed by bioinformatic 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 multi-core platforms.

Research founded by the BioBITs project (Converging technologies 2007, area: Biotechnology-ICT), Regione Piemonte - Italy.

We use the CWC calculus and its sequential simulator (Sec. II) as paradigmatic example to discuss the key features required to derive an easy porting on a multi-core platform (Sec. III). In particular we will argument on the parallelisation of a single simulation instance (Sec. III-A1), many independent instances (Sec. III-A2), and the technical challenges they require. Among these, parallel programming tools for multi-core are discussed in Sec. IV, in particular we will focus on stream oriented pattern-based parallel programming supported by the FastFlow framework (Sec. IV-A). The key features discussed in Sec. II are turned into a family of solutions to speed up both the single simulation instance and many independent instances. The former issue is approached using SIMD hardware accelerators (Sec. V-A), the latter advocating a novel simulation schema based on FastFlow accelerator that guarantees both easy development and efficient execution (Sec. V-B). The proposed solutions are experimentally evaluated.

II. The Calculus of Wrapped Compartments

The Calculus of labelled Wrapped Compartments (CWC) (see [9], [11]) is based on a nested structure of ambients delimited by membranes with specific proprieties. Biological entities like cells, bacteria and their interactions can be easily described in CWC.

A. CWC

Let $\mathcal{A}$ be a set of *atomic elements* (*atoms* for short), ranged over by a, b, ..., and $\mathcal{L}$ a set of *compartment types* represented as *labels* ranged over by $\ell, \ldots$. A *term* of CWC 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* (represented by the multiset of atoms $\overline{a}$), a *content* (represented by the term $\overline{t'}$) and a *type* (represented by the label ℓ).

An example of term is $a \;\; b \;\; (c \;\; d \rfloor e \;\; f)^\ell$ representing a multiset (multisets are denoted by listing the elements separated by a space) consisting of two atoms a and b (e.g. two 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 (e.g. two proteins) on its surface, and containing the atoms e (e.g. a molecule) and f (e.g. a DNA strand).

System transformations are defined by rewriting rules. A rewriting rule is defined as a pair of terms (on an extended set of atomic elements which includes variables), which represent the *patterns*, ranged over by P, O, together with a label ℓ representing the compartment type to which the rule can be applied. Rules are represented as expression of the form $\ell : P \mapsto O$. A simple example of a rewrite rule is $\ell : \; a \; b \; X \mapsto c \; X$ meaning that in all compartments of type ℓ an occurrence of a, b (X can match with all the remaining part of the compartment content) can be replaced by c. The application of a rule $\ell : P \mapsto O$ to a term $\overline{t}$ is performed in the following way: 1) find (if it exists) the content (or the wrap) $\overline{u}$ of a compartment of type ℓ in $\overline{t}$ and an substitution σ of variables by terms such that $\overline{u} = \sigma(P)$. 2) replace in $\overline{t}$ the subterm $\overline{u}$ with $\sigma(O)$. We write $\overline{t} \mapsto \overline{t'}$ if $\overline{t'}$ is obtained by applying a rewrite rule to $\overline{t}$.

B. Stochastic Simulation

A stochastic simulation model for biological systems can be defined by incorporating a collision-based stochastic framework along the line of the one presented by Gillespie in [8], which is, *de facto*, the standard way to model quantitative aspects of biological systems. The idea of Gillespie's algorithm is that a rate constant is associated with each considered chemical reaction. Such a constant is obtained by multiplying the kinetic constant of the reaction by the number of possible combinations of reactants that may occur in the system (thus modelling the law of mass action, but more flexible approaches are also considered in the literature [9]). The resulting rate is then used as the parameter of an exponential distribution modelling the time spent between two occurrences of the considered chemical reaction.

Each reduction rule is enriched by the kinetic constant k of the reaction that it represents (notation $\ell : P \stackrel{k}{\longmapsto} O$). The number of reactants in a reaction represented by a rewrite rule is evaluated considering the number of distinct occurrences, in the same context, of sub-terms matching with the considered rule. For instance in evaluating the application rate of the stochastic rewrite rule $R = \ell : a \;\; b \;\; X \stackrel{k}{\longmapsto} c \;\; X$ to the term $\overline{t} = a \;\; a \;\; b \;\; b$ in a compartment of type ℓ we must consider the number of the possible combinations of reactants of the form $a \;\; b$ in $\overline{t}$. Since each occurrence of a can react with each occurrence of b, this number is 4. So the application rate of R is $k \cdot 4$. This number can be evaluated by specific algorithms (we refer to [9] for a more detailed account). The stochastic simulation algorithm is essentially a *Continuous Time Markov Chain* (CTMC). Given a term $\overline{t}$, a set $\mathcal{R}$ of reduction rules, a global time δ and all the reductions $e_1, \ldots, e_M$ applicable to $\overline{t}$, with rates $r_1, \ldots, r_M$, Gillespie's "direct method" allows to determine:

- The exponential probability distribution (with parameter $r = \sum_{i=1}^{M} r_i$) of the time interval τ after which the next reaction will occur;
- The probability r_i/r that the reaction occurring at time $\delta + \tau$ will be e_i.

C. The CWC simulator

The CWC simulator is a tool strictly based on Gillespie's direct method algorithm [8]. Starting from an initial term it iterates the following three logical steps:

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 (Monte Carlo):* it stochastically decides the *time* offset at which the next reaction will occur and the *rule* that will activate it.
3) *Update:* if effectively applied the selected reaction, affecting both the system and the clock, moving forward the simulation process.

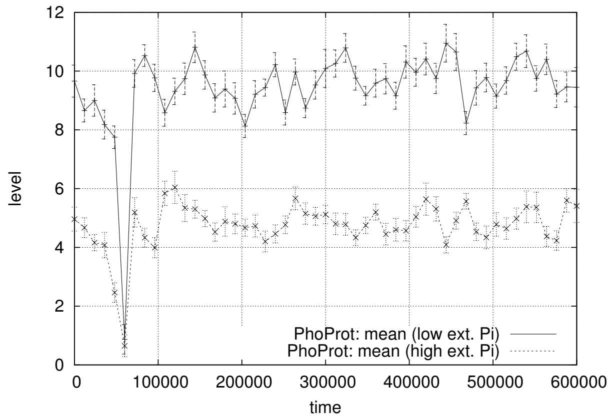

Fig. 1. Output of the CWC simulator for gene regulation in E. Coli model: average of 100 independent instances with variance (90% confidence) computed a fixed simulation time steps.

III. Exploiting Parallelism in Simulations

Gillespie algorithm realises 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. A typical output of the CWC simulator for gene regulation in E. Coli is reported in Fig. 1, where each curve is obtained by averaging 100 trajectories (90% confidence intervals are also indicated).

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 [10]. 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 multi-core 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 [1].

A. What can speeded up? Where parallelism can be found?

1) Speeding up a single simulation: Parallelising 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 parallelising 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. V). 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 parallelisation: 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 parallelisation opportunity has to be searched in data parallelism to be exploited via a hardware accelerator, such as internal SSE/MMX or external GPGPU accelerators (General-Purpose GPU). In both cases, the simulator code should be deeply redesigned in a contiguous sequence of SIMD instructions. As we shall see in Sec. V, this generally may lead to very modest advantages with respect to the required effort.

2) Speeding up independent simulation instances: The intrinsic complexity in the parallelisation 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. Notwithstanding that the problem has been often approached in a simplified form, often assuming that output data has a negligible size, as it happens in Monte Carlo Pi computation; this is not likely to happen in this and next generation biological simulations.

In particular, simulation distribution, result gathering, trajectory data assembling and analysis phases are neither considered as a part of the problems to be accelerated nor considered in the performance evaluation. As matter of a fact, parallel simulations is often considered an “embarrassingly parallel” problem, whereas it is – if and only if – data distribution, gathering, filtering, and analysis are not considered as part of the whole process. Unfortunately, it happens that they may result as expensive as the simulation itself. As an example, a simulation of the HIV diffusion problem (computed using the StochKit toolkit for 4 years of simulation time) produces about 5 GBytes of data per instance [12]. As clear, the data size is n-folded when n instances are considered. During post-processing phase, this data should be gathered and often reduced to a single trajectory via statistical methods.

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 prevalently an I/O-bound problem on a multi-core 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 characterised as I/O-bound.

B. How to parallelise? 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 parallelisation issues that, we believe, can be used as pragmatic "guidelines" for the efficient parallelisation of this kind of algorithms on multi-core. 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. V as "instruments" to evaluate the quality of the parallelisation work for the execution of independent instances of the CWC simulator.

1) 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 multi-core platforms via a suitable *high-level programming tools*.

2) 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. We believe that, among many, one of the reasons for such failure is the fact that programming multi-core 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 multi-core is required to be graceful in the short term: existing applications should be ported to multi-core 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.

3) Cache-friendly synchronisation for data streams: Current commodity multi-core and many-core platforms exhibit a cache-coherent shared memory since it makes 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. 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.

4) Load balancing of irregular workloads: Stochastic processes exhibit an irregular behaviour in space and time by their very nature since different simulations may cover the same simulation timespan following many different, randomly chosen, paths and number of iterations. Therefore, parallelisation tools should support the dynamic and active balancing of workload across the involved cores.

IV. Pattern-based high-level stream parallelism

Stream parallelism is a programming paradigm supporting the parallel execution of a stream of tasks by using a series of *sequential* or *parallel* stages. A stream program can be naturally represented as a graph of independent *stages* (kernels or filters) that 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. As with all kinds of parallel program, stream programs can be expressed as a graph of concurrent activities, and directly programmed using a low-level shared memory or message passing programming framework. Although this is still a common approach, writing a correct, efficient and portable program in this way is a non-trivial activity. Attempts to reduce the programming effort by raising the level of abstraction through the provision of parallel programming frameworks date back at least three decades and have resulted in a number of significant contributions. Notable among these is the *skeletal* approach [13] (a.k.a. *pattern-based* parallel programming), which appears to be becoming increasingly popular after being revamped by several successful parallel programming frameworks [14], [15], [16].

Skeletons (a.k.a. patterns) 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 [17].

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.

The *FastFlow* implementation of the loop and farm patterns will be exploited in Sec. V to parallelise the CWC simulator.

A. The FastFlow skeleton-based programming framework

FastFlow is a C++ parallel programming framework aimed at simplifying the development of applications for multi-core platforms. 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 [18].

FastFlow is conceptually designed as a stack of layers that progressively abstract the shared memory parallelism at the level of cores up to the definition of useful programming constructs supporting structured parallel programming on cache-coherent shared memory multi- and many-core architectures (see Fig. 2), including commodity, multi-core systems such as Intel core and AMD K10. FastFlow natively supports stream parallelism since it implements parallelism patterns as data-flow graphs – so-called *linear streaming networks*. The core of the FastFlow framework (i.e. *run-time support* tier) provides an efficient implementation of Single-Producer-Single-Consumer (SPSC) FIFO queues. FastFlow SPSC queues are lock-free, wait-free, and do not use interlocked operations [19]. The SPSC queue is primarily used as synchronisation mechanism for memory pointers in a consumer-producer fashion. The next tier up extends one-to-one queues (SPSC) to one-to-many (SPMC), many-to-one (MPSC) synchronisations and data flows, which are implemented using only SPSC queues and arbiter threads, thus providing lock-free and wait-free arbitrary data-flow graphs (*arbitrary streaming networks*) that requires few or no memory barriers, and thus few cache invalidations. The upper layer, i.e. *high-level programming*, provides a programming framework based on parallel patterns. In particular, FastFlow provides *farm*, *farm-with-feedback* (i.e. Divide&Conquer) and *pipeline* patterns, and supports their arbitrary nesting and composition. The FastFlow pattern set can be further extended by building new C++ templates.

FastFlow is available as an open source software under LGPLv3 [18]. A performance comparison against other programming tools such as POSIX, Cilk, OpenMP, and Intel TBB has been reported in [18], [20].

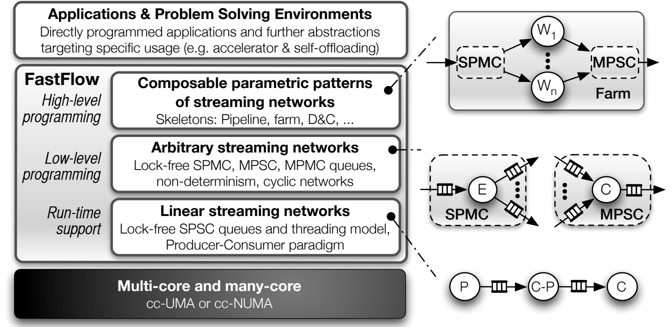

Fig. 2. FastFlow layered architecture with abstraction examples at the different layers of the stack.

```
Simulation_Step {
  // 1. Match
    foreach r ∈ ruleset {
      Match(r, T, TOP_LEVEL); // [non−SIMD parallelism]
  // 2. Resolve (Monte Carlo)
    (tau, mu) = Gillespie(matchset);
    context = stochastic_choice on matchset[mu];
  // 3. Update
    (P,O) = left_and_right_side (mu);
    delete P_sigma from T at context; // SIMD
    put deleted_elements in sigma;
    add O_sigma to T at context; // SIMD
    simclock += tau;
}
```

Fig. 3. CWC simulator pseudo-code (see also Sec. II-C) with possible sources of fine-grain parallelism.

V. 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 (exploiting the C++ boost library), then it has been parallelised for multi-core. 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 [21] to speed up a single simulation, and the FastFlow parallel programming framework [18] to speed up independent simulation instances, which provides the basic facilities described in Sec. III-A2 and that is briefly recapped in Sec. IV-A.

All 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. The Nehalem processor uses Simultaneous MultiThreading (SMT, a.k.a. HyperThreading) with 2 contexts per core that share execution units. Each core is equipped with a SSE4.2 SIMD engine.

A. Speeding up a single simulation

As discussed in Sec. III-A1, the parallelisation of the single CWC simulation step is theoretically feasible via the SSE accelerator. The pseudo-code of the simulation step is sketched in Fig. 3. In the figure, the phases of the code that can be

# 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

Fig. 4. Execution time (S) and speedup of the SIMD CWC simulator against the sequential version on the n-species *Lotka-Volterra*.

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; Fig. 4 reports the achieved speedup on a single core for n-species of the *Lotka-Volterra* models (the 2-species case is the 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's applies [22]). 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 Gillespies first reaction method on NVIDIA CUDA [23].

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. 3) 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 an example the parallelisation via POSIX threads (tested with FastFlow and Intel TBB) is, in our the reference platform, 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 if it might require, for this class of algorithms, a coding effort that easily overcomes the potential benefits.

B. 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 multi-core. Its design aims to address all the issues discussed in Sec. III-A2: it is realised by means of the FastFlow framework (see Sec. IV-A) 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 multi-core for the execution of multiple simulation instances (either replicas or the parameter sweeping of a simulation), and the on-line synthesis of their trajectories, which can be made according to one or more associative reduction functions, e.g. average, variance, confidence.

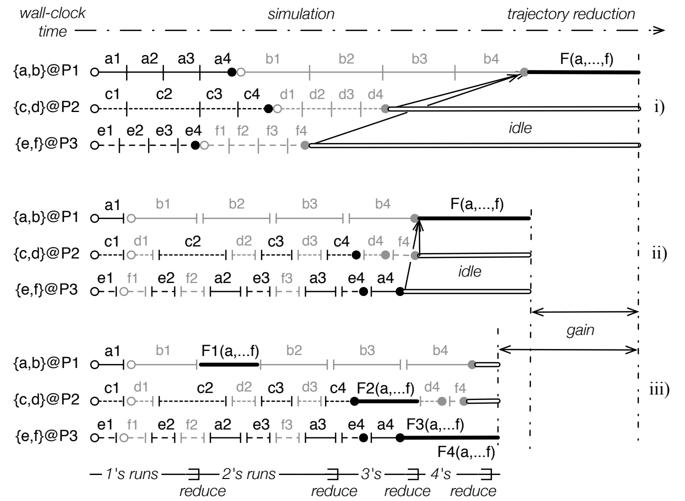

Fig. 5. Three alternative parallelisation schemas exemplified on 6 simulation instances and 3 processors. *i)* Round-robin execution of simulations followed by a reduction phase. *ii)* Auto-balancing schema with time-slicing at constant simulation time (variable wall-clock time) followed by a reduction phase. *iii)* Previous schema with on-line pipelined reduction.

The schema supports three main behaviours, which are exemplified in Fig. 5:

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 simplify the reduction process even if it is not strictly required since data could be on-line re-aligned during the buffering [12]. 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-*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 time 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

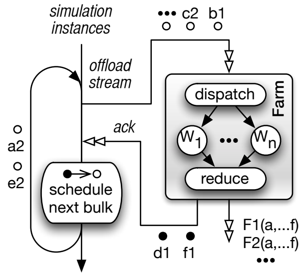

Fig. 6. Architecture of the FastFlow-based CWC parallel simulator.

workers provided slices of the same instances are sequenced (possibly on different workers). Thanks to cache-coherent shared memory the scheduling can be efficiently realised via pointer management. The idea is exemplified in Fig. 5-*ii*. Also, scheduling and dispatching to workers can be equipped with predictive heuristics based on instance history in order to characterise 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 thread (i.e. the farm collector), can be either run on an additional processor or interleaved with the execution of simulation instances (see Fig. 5-*iii*. The solution also significantly reduces the amount of data to be kept in memory because: 1) thanks to interleaving all the trajectories advances almost aligned with respect to simulation time; 2) the already reduced parts of the trajectories can be deleted from main memory (and stored in secondary memory if needed).

The three schemas can be effectively implemented using FastFlow as sketched in Fig. 6. In particular, the FastFlow *farm accelerator* feature [18] 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 simulator, and to implement user-defined dispatching and reduction functions via standard Object Oriented subclassing. As discussed in Sec. IV-A, 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 multi-core 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.

Figure 7 reports the achieved speedup for schema *iii)*, evaluated on multiple instances of simulation over the Lotka-Volterra model. In the speedup plot, both the parallel and

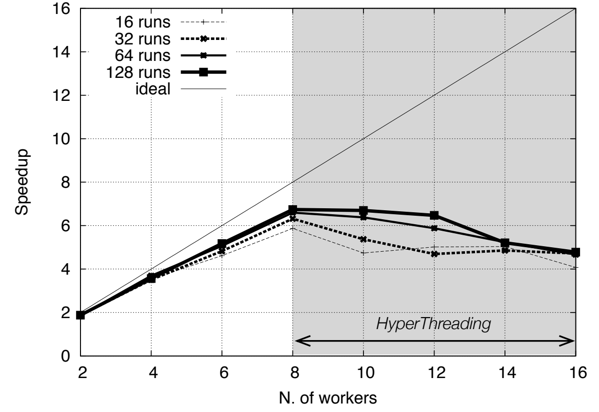

Fig. 7. Speedup of the parallel CWC simulator, see Sec. V-B *iii)*.

the sequential versions of the code include the time spent for reducing multiple trajectories with mean, variance and their confidence intervals. Observe that, the usage of multiple contexts via HyperThreading not only does not bring any additional benefit, as it can expected in CPU-intensive workloads, but it worsens the execution time due to the increased memory and cache invalidation traffic, as it can be expected in memory-bound workloads.

VI. Related Works

The parallelisation of stochastic simulators has been extensively studied in the last two decades. Many of these efforts focus on distributed architecture and specific simulators. Our work differs from these efforts in three main aspects: 1) it mainly address multicore-specific parallelisation issues; 2) it advocates a general parallelisation schema rather than a specific simulator, 3) it specifically address the on-line reduction of simulation trajectories, thus it is designed to manage large streams of data. To the best of our knowledge, many related works covers some of these aspects, but very few of them (if any) address all three aspects. Among related works, some are worth to be explicitly mentioned.

The Swarm algorithm [24], which is well suited for biochemical pathway optimisation has been used in a distributed environment, e.g., in Grid Cellware [25], 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) exploit that aim must involve making the problem very time consuming. However, since the instances of a PSA are independent, the distributed computing paradigm to to sample a large space of independent instances. In [26], a grid-based version of a multi-volume stochastic simulator is presented.

DiVinE is a general distributed verification environment meant to support the development of distributed enumerative model checking algorithms. It includes probabilistic analysis

features and it has been extensively used for the analysis of biological systems [27].

StochKit [28] is an extensible stochastic simulation framework developed in the C++ language. 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. It does not implement any kind of SIMD parallelism nor on-line trajectory reduction that is performed in a post-processing phase. A first form of on-line reduction of simulation trajectories has been experimented within the StochKit-FF framework [12], which is an extension of StochKit using the FastFlow runtime.

VII. Concluding remarks

Starting from the Calculus of Wrapped Compartments we have discussed the main parallelisation issues for its simulator, and in general for the stochastic simulation of biological systems, on commodity multi-core 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 multi-cores. These guidelines include both the programming language abstractions (streams and high-level programming patterns), the run-time mechanisms (SIMD parallelism, 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.

The presented guidelines have been used to develop a multicore-aware porting of the CWC simulator, which have been experimented over classic simulation problems. The experimental evidence obtained in the design and utilisation of the parallel simulator are convincing both in term of the achieved performance and the moderate porting effort for the parallelisation of multiple instances whereas it appears disappointing for the parallelisation of the single instance.

Both the FastFlow framework and the CWC simulator are open source software under LGPL licence [18], [29].

Acknowledgements

We wish to thank Gianfranco Balbo for the many fruitful discussions on simulation techniques.

References

[1] R. Alur, C. Belta, and F. Ivancic, "Hybrid modeling and simulation of biomolecular networks," in *Hybrid Systems: Computation and Control (HSCC)*, ser. LNCS, vol. 2034. Springer, 2001, pp. 19–32.

[2] C. Priami, A. Regev, E. Y. Shapiro, and W. Silverman, "Application of a stochastic name-passing calculus to representation and simulation of molecular processes," *Inf. Process. Lett.*, vol. 80, no. 1, pp. 25–31, 2001.

[3] L. Cardelli, "Brane calculi," in *Proc. of the Workshop on Computational Methods in Systems Biology (CMSB)*, ser. LNCS, vol. 3082. Springer, 2005, pp. 257–278.

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

[5] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, and A. Troina, "A calculus of looping sequences for modelling microbiological systems," *Fundam. Inform.*, vol. 72, no. 1-3, pp. 21–35, 2006.

[6] J. Krivine, R. Milner, and A. Troina, "Stochastic bigraphs," in *Proc. of Conference on the Mathematical Foundations of Programming Semantics*, ser. ENTCS, vol. 218. Elsevier, 2008, pp. 73–96.

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

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

[9] M. Coppo, F. Damiani, M. Drocco, E. Grassi, and A. Troina, "Stochastic Calculus of Wrapped Compatnents," in *Proc. of Workshop on Quantitative Aspects of Programming Languages (QAPL)*, vol. 28. EPTCS, 2010, pp. 82–98.

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

[11] M. Coppo, F. Damiani, M. Drocco, E. Grassi, M. Guether, and A. Troina, "Modelling ammonium transporters in arbuscular mycorrhiza symbiosis," *Transactions on Computational Systems Biology*, to appear.

[12] M. Aldinucci, A. Bracciali, P. Liò, A. Sorathiya, and M. Torquati, "StochKit-FF: Efficient systems biology on multicore architectures," in *Proc. of the 1st Workshop on High Performance Bioinformatics and Biomedicine (HiBB)*, ser. LNCS. Ischia, Italy: Springer, Sep. 2010. [Online]. Available: http://arxiv.org/abs/1007.1768v1

[13] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computations*, ser. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

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

[15] *Threading Building Blocks*, Intel Corp., 2009, http://www.threadingbuildingblocks.org/.

[16] 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," *CACM*, vol. 52, no. 10, pp. 56–67, 2009.

[17] M. Aldinucci and M. Danelutto, "Skeleton based parallel programming: functional and parallel semantic in a single shot," *Computer Languages, Systems and Structures*, vol. 33, no. 3-4, pp. 179–192, Oct. 2007.

[18] *FastFlow website*, Oct. 2009, http://mc-fastflow.sourceforge.net/.

[19] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[20] M. Aldinucci, M. Meneghin, and M. Torquati, "Efficient Smith-Waterman on multi-core with FastFlow," in *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*. Pisa, Italy: IEEE, Feb. 2010, pp. 195–199.

[21] *Intel ® C++ Intrinsics Reference*, Intel, 2010. [Online]. Available: http://software.intel.com/en-us/avx/

[22] G. M. Amdahl, "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: ACM, 1967, pp. 483–485.

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

[24] 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.

[25] P. K. Dhar, T. C. Meng, S. Somani, L. Ye, K. Sakharkar, A. Krishnan, A. B. Ridwan, S. H. Wah, M. Chitre, and Z. Hao, "Grid cellware: the first grid-enabled tool for modelling and simulating cellular processes," *Bioinformatics*, vol. 7, pp. 1284–1287, 2005.

[26] 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," *Proc. of the Intl. Workshop on High Performance Computational Systems Biology*, pp. 33–42, 2009.

[27] 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.

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

[29] *CWC Simulator*, 2010, http://sourceforge.net/projects/cwcsimulator/.