Exercising high-level parallel programming on streams: a systems biology use case

Marco Aldinucci, Maurizio Drocco, Guilherme Peretti Pezzi, Claudia Misale, Fabio Tordini
Computer Science Department, University of Torino, Italy
Email: {aldinuc, drocco, peretti, tordini, misale}@di.unito.it

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

***Abstract*—The stochastic modelling of biological systems, coupled with Monte Carlo simulation of models, is an increasingly popular technique in Bioinformatics. The simulation-analysis workflow may result into a computationally expensive task reducing the interactivity required in the model tuning. In this work, we advocate high-level software design as a vehicle for building efficient and portable parallel simulators for a variety of platforms, ranging from multi-core platforms to GPGPUs to cloud. In particular, the Calculus of Wrapped Compartments (CWC) parallel simulator for systems biology equipped with on-line mining of results, which is designed according to the FastFlow pattern-based approach, is discussed as a running example. In this work, the CWC simulator is used as a paradigmatic example of a complex C++ application where the quality of results is correlated with both computation and I/O bounds, and where high-quality results might turn into big data. The FastFlow parallel programming framework, which advocates C++ pattern-based parallel programming makes it possible to develop portable parallel code without relinquish neither run-time efficiency nor performance tuning opportunities. Performance and effectiveness of the approach are validated on a variety of platforms, inter-alia cache-coherent multi-cores, cluster of multi-core (Ethernet and Infiniband) and the Amazon Elastic Compute Cloud.**

I. Introduction

Systems biology is a biology-based inter-disciplinary field that studies living organisms, considering them as systems whose evolution depends on how components interact among them. To study these complex interactions within biological systems, both mathematical and computational models are needed to describe and analyse results coming from tools that implement different simulation techniques. Typically, the most common simulation technique used is stochastic simulation [1]. This technique comes as a parallel approach to ordinary differential equations (ODEs) and is able to describe transient and multi-stable behaviours of biological systems that do not appear with ODEs modelling, but demanding more resources. The stochastic simulation uses Monte Carlo methods applied to a number of independent instances, in order to obtain a result that is statistically significant. Because of their independence, these instances make the simulation an embarrassingly parallel problem: thanks to this property, groups of instances can be executed independently. This does not trivially extends to the whole simulation-analysis pipeline, which exhibits data dependencies between the two stages of the processes. In particular, data from simulation stage should be collected and aligned to simulation time before being ready to be analysed. In the parallelisation of both stages a number of performance problems might arise: load-balancing among simulations (which are typically heavily unbalanced), overheads due to low computation-to-communication ratio, increased memory traffic due to need of storing simulation results, increased complexity in enforcing data dependencies along the pipeline. Implement general and efficient code for a variety of platforms is a complex task, especially when facing state-of-the-art techniques, which are required to build efficient tools. They are, for example, lock-free algorithms in shared memory [2], [3], zero-copy message passing, and asynchronous data feeding and non-coherent memory hierarchies in GPGPUs. To overcome this problem, it is of fundamental importance for programmers productivity to rely on a framework that rises the level of abstraction and provides them with parallel programming paradigms masking delicate tasks, such as orchestration, synchronisation, and data movements.

In this work, the simulator for the Calculus of Wrapped Compartments (CWC) is presented as a test bed for the FastFlow programming framework [4], which provides high-level programming constructs to help in simplifying the development of applications. Over the last several years, different versions of the CWC were implemented, including shared memory multi-core, clusters, and cloud. Here we want to address the data parallelism exploitable via GPGPU accelerators, thanks to the offloading facilities offered by the FastFlow framework. This work is organised as follows: related work on high level development frameworks and other parallel stochastic simulators are discussed in Section II and Section III focuses on the FastFlow framework. Section IV presents all implemented versions of the CWC simulator, discussing also the new target of GPGPU. Performances are shown in Section V, while Section VI concludes the paper.

II. Related Work

A. High-level development tools

Programming tools and frameworks are needed to efficiently target the architectures hosting homogeneous and heterogeneous multi-core platforms or cluster of workstations. The same necessity is present also when targeting grids and clouds. The standard tools for shared memory multi-cores and distributed systems are OpenMP and MPI that, unfortunately, are not always satisfactory solutions. OpenMP offers a high-level API for (cycle independent) loop parallelism, whereas

This work has been partially supported by the EC- FP7 STREP project Paraphrase (n. 288570), the Fondazione San Paolo IMPACT project (ID. ORTO11TPXK), and by the NVidia CUDA research center programme.

streaming and task parallelism can be approached only at low level of abstraction. MPI exhibits an even lower-level approach, where the programmer has to design domain decomposition, data serialisation, and deadlocks avoidance.

Pattern-based parallel programming, rooted in the algorithmic skeletons experience [5] and programming patterns [6], promotes parallel paradigms as first-class programming constructs. 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 above mentioned problems [5]. More recently, some of the existing frameworks have been extended to target clusters of multi-cores. 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 extended to target multi-core architectures and distributed multi-core workstations by generating OpenMP code. [7]. Some other frameworks provide a high level of abstraction, but are oriented to coarse grain computations (e.g. *StreamIt* [8], *Brook* [9]). SkePU specifically targets parallel patterns for GPGPUs [10].

B. Systems biology simulators

Bio-PEPA [11] is a language for modelling and analysis of biochemical networks. It is based on the PEPA process algebra for computer systems performance analysis and then extended to model biochemical networks. Supported analysis include stochastic simulation analysis based on ordinary differential equations (ODEs) and model checking. Two software tools are available for modelling with Bio-PEPA: the Bio-PEPA Workbench and the Bio-PEPA Eclipse Plugin. Efforts have been done to parallelise such simulators, focusing on distributed architectures. StochKit [12], developed in the C++ language, aims at making stochastic simulation accessible to biologists, while remaining open to extension via new stochastic and multi-scale algorithms. It implements the Gillespie algorithm and targets multi-core platforms but it does not implement the post-processing phase of on-line trajectory reduction. In [13], is proposed a GPGPUs version of the Gillespie's SSA by way of the NVidia CUDA framework aiming to compute in parallel different trajectories. StochSimGPU [14], a GPU-accelerated version of the Matlab framework, allows computation of averages and histograms of the molecular populations across the sampled realisations on the GPU. It can be noticed that GPGPUs are heavily used in scientific applications that need to exploit high-performance computing, despite the fact that their usage, if directly approached via CUDA/OpenCL programming, might needs a lot of work to tune the algorithm to the underlying architecture.

III. FastFlow

FastFlow is a general-purpose C++ programming framework for heterogenous parallel platforms (platforms consisting of CPUs, GPUs and possibly other processors) that simplifies the design and engineering of portable parallel applications [4]. As shown in Fig. 1, it is designed as a stack of layers that abstract out the shared memory parallelism at the level of cores up to the definition of useful programming abstractions and parallel patterns. The abstraction process has two main goals: 1) to promote high-level, platform-independent parallel programming, and in particular patter-based programming, and 2) to realise efficient implementation of applications for homogeneous and heterogeneous platforms, such as multi-core, many-core (GPGPU) and distributed clusters of them.

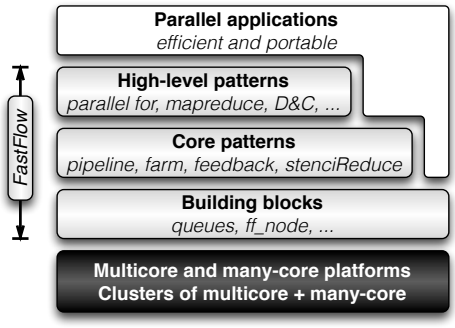

Fig. 1. FastFlow Layered Design.

FastFlow *building blocks* realises a (low-level) concurrent programming model as a CSP/Actor hybrid model: processing components are realised as POSIX threads or processes; channels are realised via either lock-free shared-memory Single-Producer-Single-Consumer FIFO queues or distributed zero-copy channels. These mechanisms are abstracted at the next level up to define few compositional *core patterns*, i.e. farm, pipeline and feedback. Also, core patterns include a single GPU-specific pattern called *stencilReduce*, which is general enough to model most of the interesting GPGPUs computations including iterative stencil computations.

Core patterns (or a composition of them) are further abstracted at the (*High-level patterns*) layer (via C++ class inheritance) to cover common parallel computing pattern in the whole spectrum of Task, Data, and Stream parallelism. Examples are parallel-for (likewise OpenMP *parallel*), Divide&Conquer, streaming pipelines and farm, map, reduce, MapReduce, Stencil, etc. The extensive description of FastFlow is beyond the scope of this paper. FastFlow programming model description and performance comparison with other programming frameworks (e.g. POSIX threads, Cilk, OpenMP, Intel TBB) can be found in related works [4], [15].

IV. CWC Implementations

The Calculus of Wrapped Compartments (CWC) is a formalism, based on term rewriting, for the representation of biological systems. Starting from an alphabet of atomic elements, CWC terms are defined as multisets of elements and compartments. Elements can be localised by compartmentalisation and the structure of a compartment can be specified by detailing the elements of interest on its membrane. For instance, a cell can be represented as a compartment and its nucleus with a separate, nested, compartment. The evolution of the system is driven by a set of rewrite rules modelling the reactions of interest that can be local to a single compartment, or involve different compartments. Compartments can be dynamically created or destroyed [16] In the following subsections, we present all the CWC simulator implementations, which implement the Gillespie algorithm on CWC terms. Notice that, due to the need of managing nested compartments,

any implementation of the CWC is significantly more complex than a plain Gillespie algorithm because terms should be represented by dynamic data structures (trees actually). The evolution of a single step of the system requires a number tree-matching functions. The Gillespie algorithm realises a Monte Carlo simulation on repeated random sampling to compute the result. Each individual simulation is called a trajectory, which tracks the state of the system at each time step.

A. Shared memory multi-core

The CWC simulator is designed in order to exploit both parallel simulation and data analysis in a single workflow: data distribution, parallel simulations, results collecting, parallel trajectory, data assembling and analysis are pipelined. Figure 2 shows the proposed three stage pipeline, composed by simulation, analysis and display of results.

1) Simulation pipeline: The simulation pipeline is composed of *i) generation of simulation tasks* stage, *ii) farm of simulation engines* stage and *iii) alignment of trajectories* stage [17]. 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 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 them according to the amount of simulation time. Once all simulation tasks overcome a given simulation time, an array of results is produced and streamed to the analysis pipeline.

2) Analysis pipeline: By design, each cut of simulation trajectories (i.e. an array containing the results of all simulations at a given simulation time), can be analysed immediately and independently (thus concurrently) from each other. More complex analysis, instead, require the access to the whole dataset, but it is difficult to do with an on-line process. In many cases it is approximated by way of sliding windows over the whole dataset. For this reason, the incoming stream is passed through sliding windows of trajectory cuts. Each sliding window can be processed in parallel. Results are collected and re-ordered (i.e. gathered) and streamed toward the user interface and permanent storage [18].

3) The graphical user interface: The CWC simulation-analysis pipeline can be controlled both from a command line and graphical user interface, which makes it possible to design the biological model, run simulations and analysis and to view partial results during the run.

B. Distributed CWC Simulator and Cloud

Thanks to the extension of the FastFlow to distributed platforms [19], the CWC simulator porting has been possible with very limited code modifications. In this version, the simulation pipeline was changed to a farm of simulation pipelines that can be run on different platforms. Each farm receives simulation parameters from the node in charge of the generation of simulation tasks, and feeds the *alignment of trajectories* node with a stream of results. The pipeline was also extended to implement de-serialising and serialising activities without modifying the existing code. By considering an IaaS cloud platform as a virtual cluster of shared memory multi-core platforms, the distributed CWC Simulator can be easily fit to run on this kind of platforms [18]. The FastFlow framework generates the connections among each node with streams, which are implemented with the shared-memory model within a virtual machine and in the message-passing model among them.

C. GPGPUs

Thanks to the FastFlow support for execution offloading onto NVidia CUDA GPGPUs – based on the recently released CUDA Unified Memory feature, which provides the automatic C++ object serialisation from host and GPGPU and vice-versa – the porting of CWC for GPGPUs requires very limited effort. Notice that the Unified Memory together with support for recursion in GPGPU kernel code are enabling features for this kind of code due to the need to move C++ objects (i.e. simulations) including linked-lists and dynamic data structures from host memory to GPU and vice-versa. Prior to the advent of Unified Memory, data to be moved to GPGPUs was required to be continuous in memory. The manual serialisation of all data structures that should be moved from host to GPGPUs, despite being theoretically possible, will involve profound and complex extensions to the code base.

Abstractly, the execution of simulation quanta over different instances (i.e. different CWC terms) can be regarded as a data parallel pattern. The user intervention would amount to writing the CUDA code for a CUDA kernel which runs a simulation quantum for a single instance, then wrapping it into `ff_mapCUDA` nodes (one for each GPGPU available). The `ff_mapCUDA` node can be regarded as an execution environment that allows the user to seamlessly interact with a CUDA device. FastFlow takes care of memory management (via either Unified Memory or explicit copy to device and vice-versa for older CUDA runtime versions) and work segmentation for load balancing. Note that, due to the atomic nature of the CUDA kernel execution model, collection of outcomes for a simulation quantum could not start until all the instances have completed the quantum. Even if this partially mitigates the depicted on-line fashion of data collection and analysis, simulation quanta could be kept very short thanks to low overhead introduced by CUDA kernel calls.

V. Performance

In this section, the evaluation of the CWC Simulator is presented, considering all versions shown in the previous section.

A. Performance on multi-core and a cluster of multi-core

The CWC Simulator has been tested with a model for circadian oscillations based on transcriptional regulation of the frequency gene in the fungus Neurospora. The model relies on the feedback exerted on the expression of the frequency gene by its protein product [18], [20]. The performance of the simulator is tested on two platforms: *1)* Intel workstation, equipped with 4 eight-core E7-4820 Nehalem (64 HyperThreads) @2.0GHz with 18MB L3 cache and 64 GBytes of main memory running Linux CentOS 6.5 x86_64.

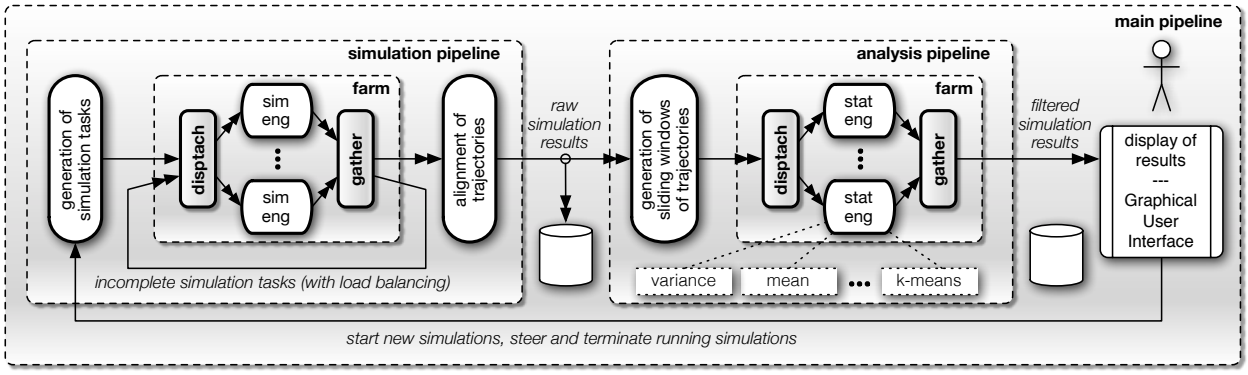

Fig. 2. Architecture of the CWC multi-core simulator with on-line parallel analysis. Double-headed arrows are streams; grey boxes are automatically generated by the FastFlow framework; white boxes hold user-defined code.

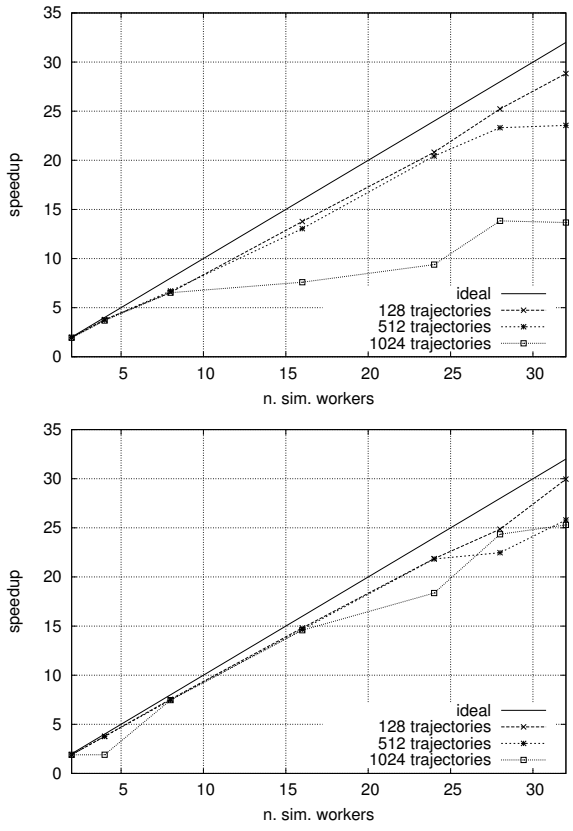

Fig. 3. Speedup of the multi-core simulator on the Neurospora model on the Intel 32-cores platform with different numbers of simulations (trajectories). 1) using a single statistical engine. 2) using 4 statistical engines.

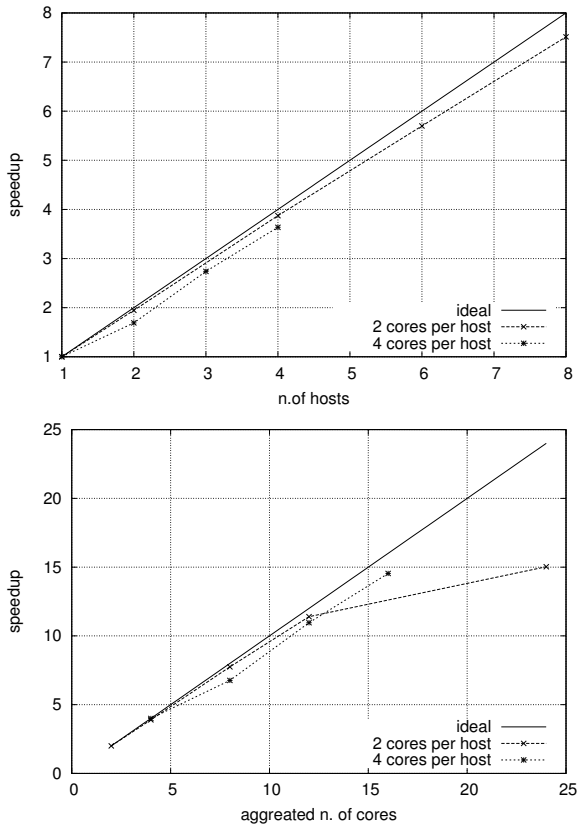

Fig. 4. Speedup of the multi-core simulator on the Neurospora model on the Intel cluster of multi-cores with two different usage of cores per node. First) w.r.t. the number of hosts. Second) w.r.t. aggregated number of cores.

The Nehalem processor uses HyperThreading with 2 contexts per core, and *2)* an Infiniband connected cluster of Intel workstations, each of them equipped with 2 six-core Xeon-X5670 (12 HyperThreads) @3.0GHz with 12MB L3 cache and 24 GBytes of main memory running Linux x86_64. The Infiniband network is used via the TCP/IP stack (IPoIB).

The speedup of the multi-core version is reported in Fig. 3. The upper figure uses a single statistical engine in the analysis pipeline, while the second uses a farm with 4 statistical engines. In the first case, the simulator succeeds to effectively use all the simulation engines only up to 512 independent simulations (producing 512 independent trajectories), reaching close to ideal speedup for limited data sets. The speedup decreases with the dimension increasing of the dataset, because of the on-line data filtering and analysis.

The speedup of the distributed version of the simulator is shown in Fig. 5, against the number of hosts (top) and the aggregated count of cores in different hosts (bottom). In both cases 4 statistical engines are used. Here, speedup is also influenced by the number of simulation engines per host since

the kind of latency and bandwidth involved in data streaming depend on the kind of channel (shared-memory or network).

B. Performance on cloud

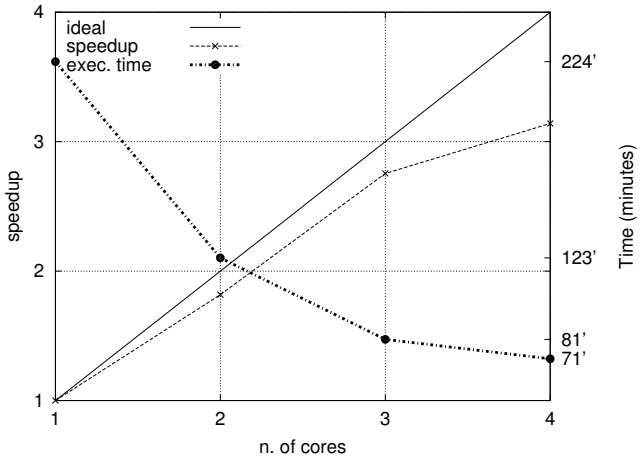

Fig. 5. Execution of the Simulator in the single quad-core VM in the Amazon EC2: speedup and execution time on the variation of virtualized cores.

Within this experiment, the *Neurospora* model for circadian oscillations based on transcriptional regulation of frequency is considered [20]. We compute the period of each oscillation and plot the moving average of more than 200 simulations of the local period. The first experiment was on eight virtual machines (VMs) each equipped with four Intel E-2670@2.6GHz cores with 20MB L3 cache, running on the Amazon Elastic Compute Cloud (Amazon EC2). Before testing on VMs, the speedup and the execution time of the simulator on a 96-days run on a single quad-core VM was measured. Data in Figure 5 show that the maximum speedup is next to the ideal (3.15 out of 4), with an execution time of ~ 71 minutes from the initial time of ~ 224 minutes. The speedup is not linear because of the additional work done by the on-line alignment of trajectories during the simulation. Next, the model was tested on eight quad-core VMs. Figure 6 (top) shows the speedup with respect to the variation of the number of virtual cores used. Also in this test, the speedup is almost ideal, reaching a maximum speedup of nearly 28 using 32 virtual cores. The gain, with respect to the sequential execution on a single-core VM is of $\sim 21\times$ and of $\sim 7\times$ with respect to the single quad-core VM. The same experiment has been executed on an heterogeneous environment composed of the Amazon VMs, one 32-core Intel Nehalem workstation and the 2×16-cores Intel Sandy Bridge workstations. The minimum execution time obtained using 96 cores (32 from VMs, 32 from Nehalem and 2×16 from Sandy Bride) is 69.3 seconds with a gain of $\sim 62\times$ (Figure 6 bottom). It can be considered a good result taking into account the high frequency of communications needed to collect results computed by simulation pipelines.

C. Performances on GPGPU

GPGPUs (and CUDA in particular) utilises the so-called Single-Instruction-Multiple-Thread (SIMT) execution model. Typically, independent parallel activities are mapped to threads, which are grouped in blocks in a kernel to be

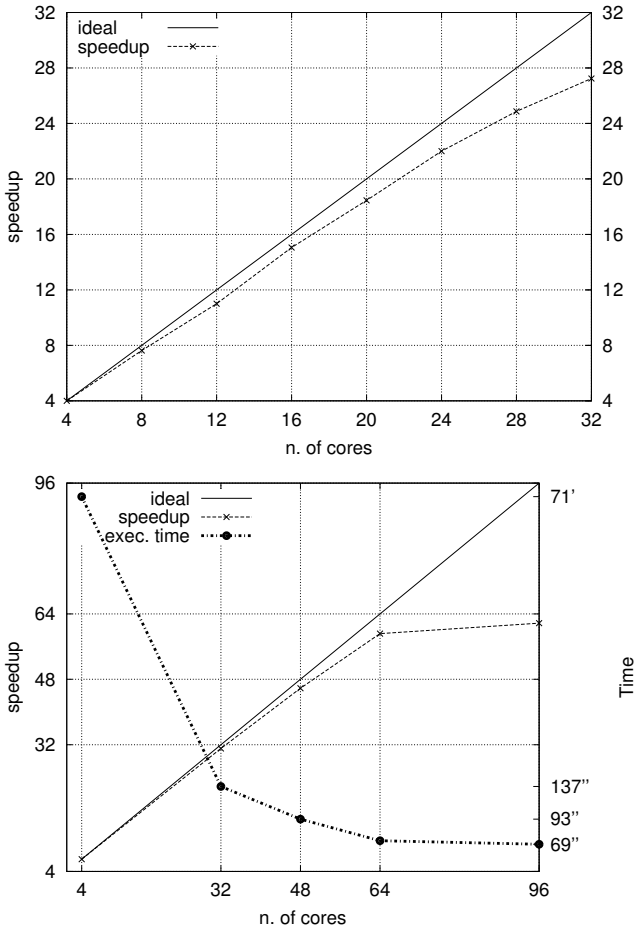

Fig. 6. Performances of the simulator on the virtual cluster of eight quad-core VMs (top) and on the heterogeneous platform (bottom) with eight quad-core Amazon EC2 VMs, one 32-core Nehalem and two 16 cores Sandy Bridge

TABLE I. EXECUTION TIME ON MULTI-CORE (INTEL) AND GPGPU (NVIDIA K40). $\mathcal{Q}/\tau$ DENOTE QUANTUM/SAMPLES RATIO.

N. sims	Execution time (s)			
	CPU (32 Intel cores)		GPGPU (2880 SMX cores)	
	$\mathcal{Q}/\tau = 10$	$\mathcal{Q}/\tau = 1$	$\mathcal{Q}/\tau = 10$	$\mathcal{Q}/\tau = 1$
128	22	22	32	39
512	83	82	47	50
1024	166	164	70	63
2048	332	328	165	104

offloaded onto the GPGPU. Threads within a block can synchronise whereas global synchronisation of threads on different blocks may happen only at kernel end. FastFlow makes it possible to build kernels according to a data-parallel pattern, significantly simplifying application development against plain CUDA/OpenCL programming, in particular with respect to all aspects involving GPGPU data feeding, kernel synchronisation, host-device interaction (which also involves the management of independent memory hierarchies), and the execution of a number of common global operations (e.g. global data reduction) [21]. However, application performance either directly using CUDA/OpenCL or FastFlow is very sensible to

a number of low-level optimisations in both kernel code and data management.

In the case of the CWC simulator, the most critical aspect involves so-called *thread divergence*. Unless in the SIMD model, in the SIMT model all threads in a block not necessarily should execute the same instruction, however any divergence turns into a performance penalty (thread stall). Due to very uneven execution time of different trajectories (due to random walks of simulation time), thread divergence turns into load balancing and eventually into performance degradation. The best performances can be expected form those models in which the structure (nested CWC compartments) of the terms is the same over all the simulation instances at a given time. In this category are biochemical and homogeneous mono-stable systems. Unfortunately they are ones that can be better studied with not stochastic approaches (e.g. ODEs). Symmetrically, multi-stable and oscillatory systems (as the presented Neurospora model) are in the worst case scenario.

Table I reports execution time of the Neurospora model on the NVidia Tesla K40 (in a Intel quad-core i3 system) against multi-core version running on 32 cores. Due to divergence, the GPGPU succeed to exploit only a fraction of its peak power, being anyway two-fold faster with respect to multi-core version. Observe that the result is possible thanks to the CWC design which manages blocks of simulations a FastFlow stream, splitting them in successive quantum and implementing a load re-balancing strategy after the computation of each quantum. As can be noticed in Table I, quantum size negligibly affects multi-core performance whereas significantly affects GPGPU performance. It eventually makes it possible to tune the same code to platforms with quite different hardware execution models.

VI. Conclusion

Seamless code portability *and* performance portability will be of paramount importance for next generation parallel programming frameworks. FastFlow approaches the problem with a pattern-based approach and low-overhead run-time support.

In this work we described how a quite complex application, the CWC simulation-analysis pipeline, once designed with the high-level FastFlow pattern-based methodology, has been ported from multi-core to clusters/cloud to GPGPU with limited source code changes. In all cases the application exploits both pipeline parallelism between simulation an analysis stages and either stream or data parallelism within both stages. Analysis of results is performed by evaluating statistical estimators and data mining on streams, thus are computed while simulation are still running.

Notably, the high-level approach supports code portability and performance exploitation on a wide range of platforms but also provides the designer with a number of knobs supporting optimisation and performance tuning the configuration level, thus significantly improving applications time-to-market.

References

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

[2] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1–26, 1998.

[3] 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.

[4] 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, Mar. 2014, ch. 13.

[5] 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.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley, 1995.

[7] 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.

[8] 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)*. Springer, 2002, pp. 179–196.

[9] 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*. New York, USA: ACM, 2004, pp. 777–786.

[10] J. Enmyren and C. W. Kessler, "SkePU: a multi-backend skeleton programming library for multi-GPU systems," in *Proc. of the fourth international workshop on High-level parallel programming and applications (HLPP '10)*. New York, USA: ACM, 2010, pp. 5–14.

[11] 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. ENTCS, vol. 194, no. 3. Elsevier, 2008, pp. 103–117.

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

[13] 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.

[14] 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.

[15] *FastFlow website*, 2014, http://calvados.di.unipi.it/fastflow/.

[16] 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.

[17] 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.

[18] M. Aldinucci, M. Torquati, C. Spampinato, M. Drocco, C. Misale, C. Calcagno, and M. Coppo, "Parallel stochastic systems biology in the cloud," *Briefings in Bioinformatics*, Jun. 2013.

[19] 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, vol. 7640. Springer, 2013, pp. 47—56.

[20] J. Leloup, D. Gonze, and A. Goldbeter, "Limit cycle models for circadian rhythms based on transcriptional regulation in drosophila and neurospora," *Journal of Biological Rhythms*, vol. 14, no. 6, p. 433, 1999.

[21] M. Aldinucci, M. Torquati, M. Drocco, G. Peretti Pezzi, and C. Spampinato, "Fastflow: Combining pattern-level abstraction and efficiency in GPGPUs," in *GPU Technology Conference (GTC 2014)*, San Jose, CA, USA, 2014.