Automatic Parallelization of Simulations on Manycore Architectures Using Scala: A Case Study

***Abstract*—At the manycore era, every simulation practitioner wants to take advantage of the computing horsepower delivered by the high performance computing devices available to him. From multicore CPUs (Central Processing Unit) to thousand-thread GPUs (Graphics Processing Unit), several architectures are now able to offer great speed-ups to simulations. However, it is often tricky to harness them properly, and even more complicated to implement a few declinations of the same model in order to compare their efficiency. Thus, simulation practitioners would mostly benefit of a simple and unified Application Programming Interface (API) that charges itself to parallelize their simulations. In this work, we study the ability of the Scala programming language to fulfill this need, thanks to third-party frameworks. We compare the features of two of them in this study: Scala Parallel Collections and ScalaCL, and challenge them with a well-known stochastic simulation model: the Ising Model. Although still in their infancy, according to what we noticed from our benchmark, these two Scala frameworks should be highly considered by the simulation community as soon as some enhancements validate their use in a scientific domain.**

***Keywords*—Parallelization of Simulation, Threads, OpenCL, Scala, Automatic Parallelization**

I. Introduction

Simulations tend to become more and more accurate, but also more and more and complex. In the same time, manycore architectures and hardware accelerators are now widespread, and allow great speedups to applications that can be parallelized in a way to make the most of their overwhelming power. Anyway, several problems arise when one is trying to parallelize a simulation. First, you need to decide which hardware accelerator will best fit your application, second, you must master its architecture's characteristics, and last but not least, you need to choose the programming language and Application Programming Interface (API) that will best suit this particular hardware.

Depending on the model and algorithms they are using, simulations will display greater speedups when parallelized on some architectures than others. For example, model relying on cellular automata algorithms will scale smoothly on vector architectures like Graphics Processing Units (GPUs).

Nowadays, even a desktop computer can be turned into an efficient computing machine when applications are able to exploit its four or soon eight CPU cores and hundreds of GPU cores. The question is how to program them correctly, easily, and in portable way so that the parallelization will still be efficient when the hardware changes. The answer to this question might be called OpenCL (Open Computing Language) [1]. OpenCL is a standard proposed by the Khronos group that aims to unify developments on various kinds of architectures like CPUs, GPUs and even Field Programmable Gate Arrays (FPGAs). It provides programming constructs based upon C99 to write the actual parallel code (called the kernel). They are enhanced by APIs (Application Programming Interface) used to control the device and the execution. OpenCL programs execution relies on specific drivers issued by the manufacturer of the hardware they run on. The point is OpenCL kernels are not compiled with the rest of the application, but on the fly at runtime. This allows specific tuning of the binary for the current platform.

OpenCL solves the three aforementioned obstacles: as a cross-platform standard, it allows developing simulations once and for all for any supported architecture. It is also a great abstraction layer that lets clients concentrate on the parallelization of their algorithm, and leave the device specific mechanics to the driver. Still, OpenCL is not a silver bullet; designing parallel applications might still appear complicated to scientists from many domains. Indeed, OpenCL development can be a real hindrance to parallelization. Thus, we need high-level programming APIs to hide this complexity to scientists, while being able to automatically generate OpenCL kernels and initializations.

Many attempts to generate parallel code from sequential constructs can be found in the literature. For the sole case of GPU programming with CUDA (Compute Unified Device Architecture), a programming paradigm developed by NVIDIA that specifically runs on this manufacturer's hardware, we can cite great tools like HMPP [2], FCUDA [3] and Par4all [4]. Other studies [5], as well as our own experience, show that CUDA displays far better performance than OpenCL, since it is optimized for NVIDIA devices. However, automatically generated CUDA cannot benefit of the same tuning quality, given that one has to consider the target device of his application to optimize a CUDA source code. Conversely, OpenCL has been designed as a cross-platform technology, and highly relies on the vendor implementation to accommodate to its host. Hence, it is better suited for generic and automatic code production than CUDA.

Another widespread cross-platform tool is Java and especially its virtual machine execution platform. Although the Java Virtual Machine (JVM) tends to become more and more efficient, the Java language itself evolves more slowly and several new languages running on top of the JVM appeared during the last few years, including Clojure, Groovy and Scala. These languages reduce code bloating and offer solutions to better handle concurrency. Among JVM languages, we have chosen to focus on Scala in this study because of its hybrid aspect: mixing both functional and object oriented paradigms. By doing so, Scala allows object-oriented developers to gently migrate to powerful functional constructs.

This study benchmarks Scala solutions to automate parallelization of simulations on manycore and GPU architectures. Indeed, Scala presents many advantages that make it the ideal candidate to perform automatic parallelization operations. In this study we will:

- Introduce Scala frameworks designed for automatic parallelization;
- Present our benchmark and its results;
- Discuss Scala limitations concerning automatic parallelization.

II. Automatic Parallelization with Scala

A. Why is Scala a good candidate for parallelization of simulations?

First of all, let us make a brief recall on what Scala is. Scala is a programming language mixing two paradigms: object oriented and functional programming [6]. Its main feature is that it runs on top of the JVM. In our case, it means that Scala developments can interoperate like clockwork with Java, thus allowing the wide range of simulations developed in Java to integrate Scala parts without being modified.

The second asset of Scala is its compiler. In fact, Scalac (for Scala Compiler) offers the possibility to enhance its behavior through plugins. ScalaCL, which we will study later in this section, uses a compiler plugin to transform Scala code to OpenCL at compile time. This mechanism offers great opportunities to generate code and the OpenCL generator studied in this work is just a concrete example of what can be achieved when extending the Scala compiler.

Finally, Scala presents a collection framework [7] that intrinsically facilitates parallelization. As a matter of fact, Scala default collections are immutable: each time a function is applied to an immutable collection, this one remains unchanged and the result is a modified copy returned by the function. On the other hand, mutable collections are also available when explicitly summoned, albeit the Scala specification does not ensure any thread-safe access on these collections. Such an approach appears to be very interesting and efficient, when trying to parallelize an application, since no lock mechanism is involved anymore. Thus, concurrent accesses to the collection elements are not a bottleneck anymore as long as no superfluous action induces overhead. Still, classical functional programming is used to issuing a new collection as the result of applying a function to an immutable collection. Although this procedure might appear costly, works have been done to optimize the way it is handled, and efficient implementations do not copy the entire immutable collection [8].

B. Scala Parallel Collections

Scala 2.9 release introduced a new set of Parallel Collections [9] mirroring the classical ones. These parallel collections offer the same methods than their sequential equivalents, but the method execution will be automatically parallelized by a framework implementing a divide and conquer algorithm.

The point is they integrate seamlessly in already existing source codes because the parallel operations have the same names as their sequential variants. As the parallel operations are implemented in separate classes they can be invoked if their data are in a parallel collection class. This is made possible thanks to a method named *par* that returns a parallel equivalent of the sequential implementation of the collection still pointing to the same data. Any subsequent operation invoked by an instance of the collection will benefit of a parallel execution without any other add from the client. Instead of applying the selected operation to each member of the collection sequentially, it is applied on each element in parallel. Such a function is referred to as a closure in functional programming jargon. It commonly designates an anonymous function embedded in the body of another function. A closure can also access the variables from the calling host function.

Scala Parallel Collections rely on the Fork/Join framework proposed by Doug Lea [10]. This framework based upon the divide and conquer paradigm was released with the latest Java 7 Software Development Kit (SDK). Fork/Join introduces the notion of tasks, which are basically a unit of work to be performed. Tasks are then assigned to worker threads waiting in a sleeping state in a Thread Pool. The pool of worker threads is created once and only once when the framework is initialized. Thus, creating a new task does not suffer of thread creation and initialization that could, depending on the task content, be slower than processing the task itself.

Fork/Join is implemented using work stealing. This adaptive scheduling technique offers efficient load balancing features to the Java framework, provided that work is split into tasks of a small enough granularity. In order to determine the ideal task granularity, ScalaCL implements an exponential task splitting technique detailed in [11].

C. ScalaCL

ScalaCL is a project, part of the free and open-source Nativelibs4java initiative, led by Olivier Chafik. This project is an ambitious bundle of libraries trying to allow

users to take advantage of various native binaries in a Java environment.

From ScalaCL itself, two projects have recently emerged. The first one is named Scalaxy. It is a plugin for the Scala compiler that intends to optimize Scala code at compile time. Indeed, Scala functional constructions might run slower than their classical Java equivalents. Scalaxy deals with this problem by pre-processing Scala code to replace some constructions by more efficient ones. Basically, this plugin intends to transform Scala's loop-like calls such as map or foreach by their while loops equivalents. The main advantage of this tool is that it is applicable to any Scala code, without relying on any hardware.

The second element resulting of this fork is the ScalaCL collections. It consists in a set of collections that support a restricted amount of Scala functions. However, these functions can be mapped at compile time to their OpenCL equivalents. A compiler plugin dedicated to OpenCL generation is called at compile time to normalize the code of the closure applied to the collection. The resulting source is then converted to an OpenCL kernel. At runtime, another part of ScalaCL comes into play, since the rest of the OpenCL code, like the initializations, are coupled to the previously generated kernel to form the whole parallel application. The body of the closure will be computed by an OpenCL Processing Element (PE), which can be a thread or a core depending on the host where the program is being run.

The two parallelization approaches introduced in this section are compared in Figure 1:

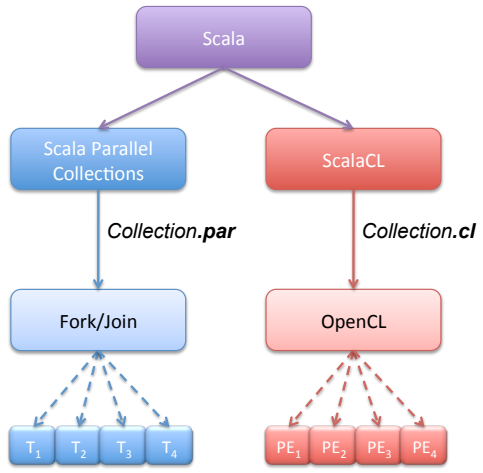


Figure 1: Schema showing the difference between the two approaches: Scala Parallel Collections spreads the workload among CPU threads (T), while ScalaCL addresses OpenCL Processing Elements (PEs)

III. Case study: Ising Model simulation

In this section, we will show how Scala parallel implementations of a simulation model are close to the Scala sequential implementation. Here, we compare sequential Scala code with its parallel declinations using Scala Parallel Collections and ScalaCL. The code snippets introduced hereafter particularly draw attention on the automatic aspect of the two studied approaches, which respective code sources remain very close to the genuine. Our benchmark consists in running several iterations of multiple configurations of the well-known Ising model.

A. Ising models

The Ising model is a mathematical model representing ferromagnetism in statistical physics [12]. Basically, Ising models deal with a lattice of points, each point holding a spin value, which is a quantum property of physical particles. At each step of the simulation process and for each spin of the lattice, we have to determine whether it has to be flipped or not. Ising models have been studied in many dimensions. For the purpose of this study, we consider a 2D toric lattice.

The purpose of this simulation algorithm is to compute the energy of the simulated configurations, in order to obtain a sample representative enough to build a bar chart of the empirical distribution of the energy levels. Such a distribution can, for instance, be compared to the actual distribution found in [13], in order to validate the implementation. Another goal is to compute an average energy and its associated standard deviation. Then we can test whether the energy's expected value, which is known accurately, belongs to the confidence interval obtained by simulation. Please note that such values have for instance been used in works like [14], [15] to characterize the statistical quality of pseudorandom sequences.

Several algorithms have been designed to rule this evolution. In our case, we have implemented a Scala version of the Metropolis algorithm [16].

The idea of this algorithm is to try to obtain a new $X_{t+1} = y$ lattice configuration from the current $X_t = x$ configuration by flipping the spin of a single point of the lattice. The new configuration is validated given the following probability:

$P(X_{t+1} = y \mid X_t = x) = min\{1, e^{-\beta \Delta E}\}$, with $\Delta E = E(y) - E(x)$.

$E(x)$ and $E(y)$ are respectively the energy levels of the configurations x and y, given by Formula 1:

$$E(x) = -J \sum_{<i,j>} s_i(x) s_j(x), \qquad (1)$$

where <i,j> is a notation to consider, in the sum, all the couples (i, j) of points of the lattice that are in the same Von Neumann neighbourhood, and $s_i(x)$ is the spin of the point i.

That is to say when $E(y)$ is lower or equal to $E(x)$, then the y configuration is accepted, whereas when $E(y)$ is greater than $E(x)$, the new configuration owns a $e^{-\beta \Delta E}$ probability to be accepted. Having said this, the Metropolis algorithm writes as follows:

Algorithm 1 Metropolis algorithm

Initialize the x configuration to a given state (usually randomly)
Let s be a random point of the lattice, chosen using a uniform distribution
Figure out the Von Neumann neighborhood of the point
Sum the spins of the neighbors and multiply it by the spin of the current point
Compute ΔE according to the Formula (1)
if $\Delta E \leq 0$ **then**
    The y configuration is obtained from x by flipping the spin of s
    $x \leftarrow y$
else
    Let u be a random number from [0 ; 1]
    if $u < e - \beta \Delta E$ **then**
        The y configuration is obtained from x by flipping the spin of s
        $x \leftarrow y$
    end if
end if

To harness parallel architectures, we need to slightly rewrite this algorithm. At the moment, a step consists in trying to flip the spin of a single point of the lattice. This action is considered as a transformation between two configurations of the lattice. However, each point can be treated in parallel, provided that its neighbors are not updated in the same time. Therefore, the points cannot be chosen at random anymore, this would necessarily lead to a biased configuration. A way to avoid this problem is to separate the lattice in two halves that will be processed sequentially. This technique is commonly referred to as the "checkerboard algorithm" [17]. In fact, the Von Neumann neighborhood of a point located on what will be considered a white square, will only be formed by points located on black squares, and vice versa. The whole lattice is then processed in two times to obtain a result equivalent to the sequential process. The process is summed up in Figure 2:

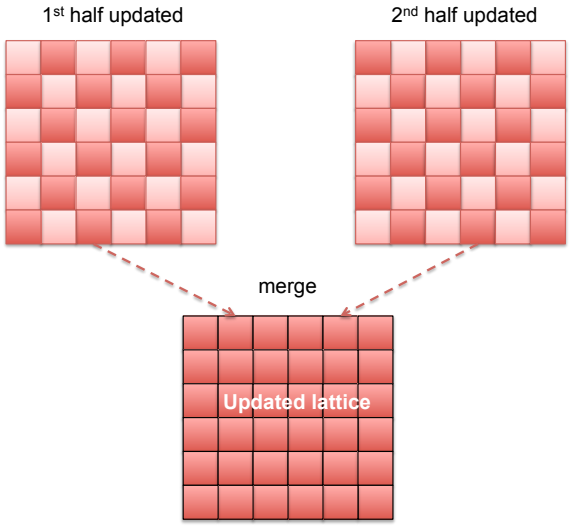


Figure 2: Lattice updated in two times following a checkerboard approach

B. Scala implementation

Our Scala implementation represents the spin lattice by an IndexedSeq. The latter is a trait: an enhanced interface in the sense of Java that also allows methods definition, and in this case models all sorts of indexed sequences. It bears operations on this kind of collections such as applying a given function to every element of the collection (*map*) or applying a binary operator on a collection, going through elements from left or right (*foldLeft*, *foldRight*). These operations are sufficient to express most of the algorithms. Moreover, they can be combined since they build and return a new collection containing the new values of the elements.

For instance, computing the magnetization of the whole lattice consists in applying a foldLeft on the IndexSeq to sum the values corresponding to the spin of the elements. Indeed, our lattice is a set of tuples contained in the IndexedSeq. Each tuple stores its coordinates in the 2D-lattice in order to easily build its Von Neumann neighborhood, and a boolean indicating the spin of the element (false is a negative spin, whereas true stands for a positive spin).

Each iteration consists in updating the whole lattice in two times following a checkerboard algorithm, as we explained previously. The implementation lies in applying an operation to update each spin through two successive *map* invocations on the two-halves of the lattice. Not only this approach is crystal clear for any developer to read, but also it is quite easy to parallelize. Indeed, *map* can directly interact with the two Scala automatic parallelization frameworks presented earlier: Scala Parallel Collections and ScalaCL. Let us describe how parallelization API interacts smoothly with already written Scala code with a concrete example.

Listing 1 is a snippet of our Ising model implementation in charge of updating the whole lattice in a sequential fashion:

Listing 1: Sequential version of method processLattice from class IsingModel

```
1 def processLattice(_lattice: Lattice)(implicit
      rng: Random) =
2
3   new Lattice {
4     val size = _lattice.size
5     val lattice =
6       IndexedSeq.concat (
7         _lattice.filter{case((x, y), _) => isEven
              (x,y)}.map(spin => processSpin(
              _lattice, spin)),
```

```
8         _lattice.filter{case((x, y), _) => isOdd(
              x,y)}.map(spin => processSpin(
              _lattice, spin))
9       )
10    }
```

Listing 1: Sequential version of method processLattice from class IsingModel

Scala enables us to write both concise and expressive code, while keeping the most important parts of the algorithm exposed. Here, the two calls to *map* aiming at updating each half of the lattice can be noticed, they will process in turn all the elements within the subset they have received in input. This snippet suggests an obvious parallelization of this process thanks to the *par* method invocation. This call automatically provides the parallel equivalent of the collection, where all the elements will be treated in parallel by the mapped closure that processes the energy of the spins. The resulting code differs only by the extra call to the *par* method upstream of the map action, so as Listing 2 shows:

Listing 2: Parallel version of method processLattice from class IsingModel, using Scala Parallel Collections

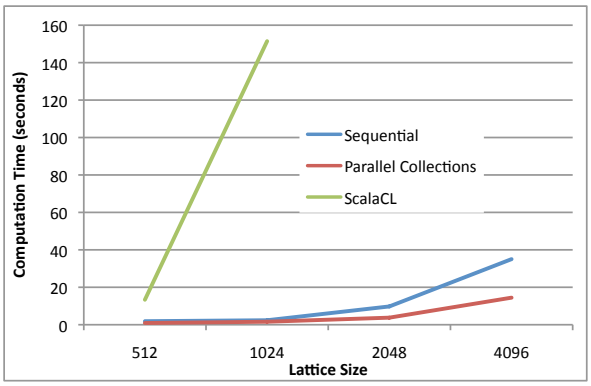
```
1  def processLattice(_lattice: Lattice)(implicit
       rng: Random) =
2
3    new Lattice {
4      val size = _lattice.size
5      val lattice =
6        IndexedSeq.concat (
7          _lattice.filter{case((x, y), _) => isEven
               (x,y)}.par.map(spin => processSpin(
               _lattice, spin)),
8          _lattice.filter{case((x, y), _) => isOdd(
               x,y)}.par.map(spin => processSpin(
               _lattice, spin))
9        )
10     }
```

Listing 2: Parallel version of method processLattice from class IsingModel, using Scala Parallel Collections

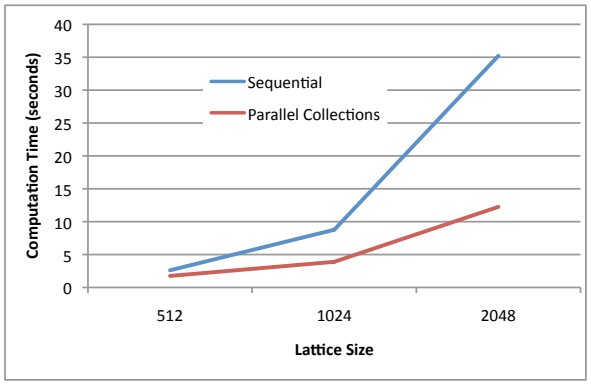
An equivalent parallelization using ScalaCL is obtained just by replacing the *par* method by the *cl* one from the ScalaCL framework, thus enabling the code to run on an OpenCL platform.

C. Results

At this stage, we will study how our different Ising Model implementations perform when asked to simulate the same lattices. We compare a sequential Scala implementation with its Parallel Collections and ScalaCL declinations. Typically, each model had to process a growing number of iterations on every lattice size we retained for this benchmark. By doing so, we have isolated the impact of the size and the ability to repeat the same computation after an initialization phase. Our tests have been performed on a bi-processor machine owning two Intel Westemere CPUs running at 2.53GHz and an NVIDIA Tesla C2050 GPU, the latter supporting the OpenCL computations only. Measures resulting from these runs are displayed in Figure 3a and 3b:



(a) Single Iteration



(b) 10 Iterations

Figure 3: Benchmark results for various lattice sizes and iterations numbers

As we might have guessed, the Parallel Collections using several threads outperforms the sequential version of the simulation no matter the considered configuration. At first, it might be surprising that the ScalaCL implementation performs so bad compared to its two challengers, but this figure is bound to the high rate of memory transfers required to transfer the lattice on the GPU and get back the new configuration on the host.

This puts the light on a major interest of the Scala automatic parallelization frameworks for simulations: the ability to quickly evaluate a parallelization approach. Indeed, an important workload would have been required to provide a handcrafted implementation of our Ising Model in OpenCL, and the result would have been very disappointing as shows our benchmark. Consequently, Scala might reveal an outstanding tool to evaluate different parallelization approaches. Although these results are not positive for the OpenCL version, they do not imply that

a finely tuned OpenCL implementation will not perform well. Here, we can conclude that the multithreaded CPU approach is worth being deeper investigated to obtain a maximum speedup.

IV. Discussion

A. Pseudorandom Numbers Generation

Lots of simulations, including the simulation of the Ising model that we took as a basis for our experimental results, are stochastic simulations. They must interact with a Pseudorandom Numbers Generator (PRNG) to pick up the random numbers needed by their computations. Scala pseudorandom numbers generation facilities are provided in the *scala.util.Random* class that is nothing more than a wrapper of the *java.util.Random* class. Now, we know from [18] that the underlying PRNG is a Linear Congruential Generator (LCG) algorithm, which might appear quite strange, given that this algorithm has been stated as flawed by lots of studies [14], [19], [20]. This problem taken apart, the question is, how are these pseudorandom numbers handled by the two parallelization solutions studied in this work?

First, let us concentrate on Scala Parallel Collections. Assume an application using scala.util.random that we would try to parallelize with the framework. Save from its feeble statistical quality, Scala's default PRNG suffers the same problem than its Java base: it is synchronized to enable thread-safe accesses. It means that threads can access the PRNG concurrently without risking to corrupt it, unfortunately, it also implies to serialize the accesses, thus creating a bottleneck on the random source. This way to distribute random numbers across computational elements is known as Central Server in [21], and it is clearly stated as a technique that should be avoided when using stochasticity in high performance computing applications. In addition, this technique disables the reproducibility of parallel stochastic simulation results when different numbers of processing elements are considered. Knowing that Scala Parallel collections are built upon the Java Fork/Join framework [10], we could have hoped that they would also have taken advantage of the ThreadLocalRandom class that is shipped with. The latter implements a parallel version of the classic LCG coming with Java, and handles random streams distribution automatically. The fact that they rely on an LCG generator is a problem by itself, such generators cannot be used for scientific applications, they have been known for many years for weak statistical properties inherent to their structure [22]. Actually, it appears that the Parallel API of Scala is only focused on collections for the moment. As a matter of fact, it does not perform a particular parsing of the closure's body to figure out whether its content has to be adapted. By doing so, Scala Parallel collections forces developers to handle pseudorandom streams on their own, which in our case reduces its automatic parallelization capabilities.

In order to tackle this lack, we have developed a Scala wrapper to ThreadLocalRandom, which allows an application to get rid of the synchronization bottleneck, and consequently see its performance increase. Thanks to our wrapper, the developer is also spared from partitioning the random sequence at hand. When our Ising Model implementation only picks up a random number per spin of the lattice, we have benchmarked our wrapper with a simple but significant case drawing 4 numbers per element of a collection.

Listing 3: Example of use our ThreadLocalRandom Scala wrapper

```
1 implicit val rng = new fr.isima.random.
      ThreadLocalRandom
2
3 (1 to N).par.map {
4   i =>
5     rng.nextDouble * rng.nextDouble * rng.
          nextDouble * rng.nextDouble
6 }
```

Listing 3: Example of use our ThreadLocalRandom Scala wrapper

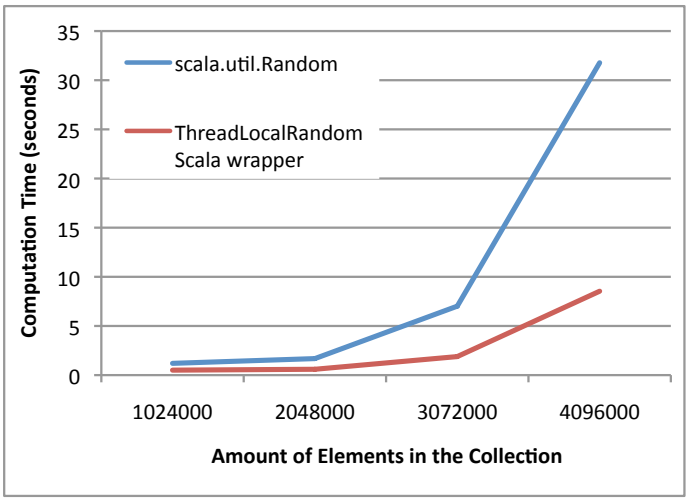


Figure 4: Benchmark of Scala PRNG classes: scala.util.Random versus ThreadLocalRandom wrapper

Figure 4 shows the bottleneck introduced by synchronized calls to scala.util.Random when calls to our ThreadLocalRandom wrapper do not introduce the same stumbling block. As long as Scala Parallel Collections is part of the Scala standard library, it can easily take advantage of every Scala features, such as the simplicity to extend standard classes, and to interoperate with legacy Java code.

ScalaCL is also collection-driven, but its functioning is slightly different given that it operates on the closure's body. Indeed, ScalaCL tries to convert the code it finds in a closure to OpenCL code. Thus, it will make an attempt to

convert the PRNG functions to OpenCL. Unfortunately, this is not the expected behavior when dealing with pseudorandom streams distribution. Streams have to be distributed carefully, and once again this has to be done by the client. Although such a feature lacks at the time of writing, ScalaCL transformation mechanism should help further versions to provide automatic stream distribution.

As a conclusion, neither Scala Parallel Collections nor ScalaCL handle pseudorandom streams properly for the moment. This behavior is problematic in the case of stochastic simulations. Indeed, previous studies have shown how badly distributed pseudorandom streams could impact simulation results [23], [24], [25].

B. Master Underlying Parallelization

We have championed Scala Parallel Collections mostly for its automatic parallelization capabilities behind the scene. Unfortunately, this is also, in our opinion, a drawback of this API. Basically, you have no way to control the threads pool automatically created by the underlying Fork/Join framework. When using Scala Parallel Collections, you are not able to determine how much threads are created, nor can you handle the way they are scheduled explicitly. As a result, elements from the same collection might be treated in a different order from an execution to another, resulting in different output collections depending on threads scheduling. Moreover, executing the same code on different architectures will result in a different amount of worker threads in Fork/Join's pool. This will obviously impact scheduling and, once again, will show dissimilar outputs on each device.

Such a behavior is quite compromising when talking about simulations. We simulation practitioners appreciate that simulation results are reproducible, either to help debugging or to check a colleague's results. This lack of control might also become problematic if the order in which the elements of a collection are processed impacts the final result of the involved operation.

V. Conclusion

In this work we have benchmarked the Scala proposals aiming at parallelizing applications automatically and their applicability to simulations parallelization. Two frameworks have been detailed and compared: Scala Parallel Collections that automatically creates tasks, and executes them in parallel thanks to a pool of worker threads; ScalaCL, part of the Nativelibs4java project, intends to produce OpenCL code from Scala closures, and run it on the most efficient device available, be it GPU or multicore CPU.

Our study has stated that although ScalaCL was still in its infancy, it deserves to be regarded as a future great language extension that will perform well in many cases. For its part, Scala Parallel Collections is a satisfying framework that mixes ease of use and efficiency. Yet, it does not enable clients to master the underlying parallelization finely, since threads handling is delegated to the Java Fork/Join framework. Given that Scala Parallel Collections can be easily extended, as our modest ThreadLocalRandom wrapper has shown, a few more developments from the original team or third-party plugins should make it a reliable solution to offer great speedups to many applications without requiring much investment. Both tools remain an interesting manner to quickly shape up and evaluate several parallelization strategies for a simulation.

Apart from their respective drawbacks, these two approaches have displayed a lack of rigor considering pseudorandom numbers generation that opens perspectives for further works. Indeed, the ThreadLocalRandom class is not satisfying enough in regards of the poor statistical quality tied to its underlying LCG PRNG algorithm. Providing an implementation of a better quality PRNG, such as MRG32k3a for instance, but conserving the same API as ThreadLocalRandom, would highly serve Scala automatic parallelization frameworks to be adopted by the simulation community.

References

[1] Khronos OpenCL Working Group, "The opencl specification 1.2," Khronos Group, Specification 1.2, November 2011.

[2] R. Dolbeau, S. Bihan, and F. Bodin, "Hmpp: A hybrid multicore parallel programming environment," in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[3] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W. Hwu, "Fcuda: Enabling efficient compilation of cuda kernels onto fpgas," in *IEEE 7th Symposium on Application Specific Processors, 2009. SASP'09.* IEEE, 2009, pp. 35–42.

[4] M. Amini, F. Coelho, F. Irigoin, and R. Keryell, "Static compilation analysis for host-accelerator communication optimization," in *The 24th International Workshop on Languages and Compilers for Parallel Computing, Fort Collins, Colorado*, September 2011.

[5] K. Karimi, N. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," http://arxiv.org/abs/1005.2581v3, August 2010, submitted. [Online]. Available: http://arxiv.org/abs/1005.2581v3

[6] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," Technical Report IC/2004/64, EPFL Lausanne, Switzerland, Tech. Rep., 2004.

[7] M. Odersky, "Scala 2.8 collections," 2010.

[8] C. Okasaki, *Purely functional data structures.* Cambridge Univ Pr, 1999.

[9] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky, "A generic parallel collection framework," in *Euro-Par 2011 Parallel Processing.* Springer, 2011, pp. 136–147.

[10] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande.* ACM, 2000, pp. 36–43.

[11] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in *Parallel Processing, 2008. ICPP'08. 37th International Conference on.* IEEE, 2008, pp. 536–545.

[12] E. Ising, "Beitrag zur theorie des ferromagnetismus," *Zeitschrift für Physik A Hadrons and Nuclei*, vol. 31, pp. 253–258, 1925, 10.1007/BF02980577. [Online]. Available: http://dx.doi.org/10.1007/BF02980577

[13] P. Beale, "Exact distribution of energies in the two-dimensional ising model," *Physical review letters*, vol. 76, no. 1, pp. 78–81, 1996.

[14] A. Ferrenberg, D. Landau, and Y. Wong, "Monte carlo simulations: Hidden errors from "good"random number generators," *Physical Review Letters*, vol. 69, no. 23, p. 3382, 1992.

[15] P. Coddington, "Random number generators for parallel computers," NHSE, Tech. Rep. 2, 1996.

[16] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller *et al.*, "Equation of state calculations by fast computing machines," *The journal of chemical physics*, vol. 21, no. 6, p. 1087, 1953.

[17] T. Preis, P. Virnau, W. Paul, and J. Schneider, "Gpu accelerated monte carlo simulation of the 2d and 3d ising model," *Journal of Computational Physics*, vol. 228, no. 12, pp. 4468–4477, 2009.

[18] Oracle, *Java Platform, Standard Edition 7 - API Specification*, 2011. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/

[19] P. Hellekalek, "Good random number generators are (not so) easy to find," *Mathematics and Computers in Simulation*, vol. 46, no. 5-6, pp. 485–505, 1998.

[20] ——, "Don't trust parallel monte carlo!" in *Proceedings of Parallel and Distributed Simulation PADS98*. IEEE, 1998, pp. 82–89.

[21] D. Hill, "Practical distribution of random streams for stochastic high performance computing," in *IEEE International Conference on High Performance Computing & Simulation (HPCS 2010)*, 2010, pp. 1–8, invited paper.

[22] P. L'Ecuyer, *Encyclopedia of Quantitative Finance*, 2010, ch. Pseudorandom Number Gemerators.

[23] K. Pawlikowski, H. Jeong, J. Lee *et al.*, "On credibility of simulation studies of telecommunication networks," *IEEE Communications Magazine*, vol. 40, no. 1, pp. 132–139, 2002.

[24] D. Hill, "Distribution of random streams in stochastic models in the age of multi-core and manycore processors," in *IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation*, 2011, p. 2, keynote.

[25] J. Passerat-Palmbach, C. Mazel, and D. Hill, "Pseudo-random number generation on gp-gpu," in *IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation*, 2011, pp. 146–153.