Prototyping Parallel Simulations on Manycore Architectures Using Scala: A Case Study

Jonathan Passerat-Palmbach, Romain Reuillon, Claude Mazel, David Hill

To cite this version:

Jonathan Passerat-Palmbach, Romain Reuillon, Claude Mazel, David Hill. Prototyping Parallel Simulations on Manycore Architectures Using Scala: A Case Study. IEEE High Performance Computing and Simulation (HPCS) 2013, Jul 2013, Helsinki, Finland. pp.405 - 412, 2013, <10.1109/HPCSim.2013.6641447>. <hal-01098626>

HAL Id: hal-01098626

https://hal.inria.fr/hal-01098626

Submitted on 28 Dec 2014

Prototyping Parallel Simulations on Manycore Architectures Using Scala: A Case Study

Jonathan Passerat-Palmbach [1] [*] [†] [‡] [§] ,
Romain Reuillon [¶] ,
Claude Mazel [*] [†] [‡] [§],
David R.C. Hill [*] [†] [‡] [§]

Originally published in: IEEE International High Performance Computing and Simulation Conference (HPCS) 2013 — July 2013 — pp 405-412
http://dx.doi.org/10.1109/HPCSim.2013.6641447

[1] This author's PhD is partially funded by the Auvergne Regional Council and the FEDER

[*] ISIMA, Institut Supérieur d'Informatique, de Modélisation et de ses Appplications, BP 10125, F-63173 AUBIERE
[†] Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 CLERMONT-FERRAND
[‡] Clermont Université, Université Blaise Pascal, BP 10448, F-63000 CLERMONT-FERRAND
[§] CNRS, UMR 6158, LIMOS, F-63173 AUBIERE
[¶] Institut des Systèmes Complexes, 57-59 rue Lhomond, F-75005 PARIS

Abstract: At the manycore era, every simulation practitioner can take advantage of the computing horsepower delivered by the available high performance computing devices. 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 to compare the parallelizations. Thus, simulation practitioners would mostly benefit of a simple way to evaluate the potential benefits of choosing one platform or another to parallelize their simulations. In this work, we study the ability of the Scala programming language to fulfill this need. We compare the features of two frameworks in this study: Scala Parallel Collections and ScalaCL. Both of them provide facilities to set up a data-parallelism approach on Scala collections. The capabilities of the two frameworks are benchmarked with three simulation models as well as a large set of parallel architectures. According to our results, these two Scala frameworks should be considered by the simulation community to quickly prototype parallel simulations, and choose the target platform on which investing in an optimized development will be rewarding.

Keywords: Parallelization of Simulation, Threads, OpenCL, Scala, Automatic Parallelization, GPU, Manycore

1 Introduction

Simulations tend to become more and more accurate, but also more and more 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 take advantage 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 their underlying models and algorithms, simulations will display greater speedups when parallelized on some architectures than others. For example, model relying on cellular automata algorithms are likely to scale smoothly on GPU devices or any other vector architecture Caux et al. (2011); Topa and Młocek (2012).

The problem is sometimes, scientists champion a given architecture without trying to evaluate the potential benefits their applications could gain from other architectures. This can result in disappointing performances from the new parallel application and a speed-up far from the original expectations. Such hasty decisions can also lead to wrong conclusions where an underexploited architecture would display worse performance than the chosen one Lee et al. (2010). One of the main examples of this circle of influence are GPUs, which have been at the heart of many publications for the last 5 years. Parallel applications using this kind of platform are often proved relevant by comparing them to their sequential counterparts. Now, the question is: is it fair to compare the performances of an optimized GPU application to those of a single CPU core? Wouldn't we obtain far better performances trying to make the most of all the cores of a modern CPU?

On the other hand, many platforms are available today, and in view of the time and workload involved in the development of a parallel application for a given architecture, it is nearly impossible to invest much human resources in building several prototypes to determine which architecture will best support an application. As a matter of fact, parallel developers need programming facilities to help them build a reasonable set of parallel prototypes targeting a different parallel platform each.

This proposition implies that developers master many parallel programming technologies if they want to be able to develop a set of prototypes. Thus, programming facilities must also help them to factor their codes as much as possible. The ideal paradigm in this case is Write Once, Run Anywhere, that suggests different parallel platforms understand the same binaries. To do so, a standard called OpenCL (Open Computing Language) was proposed by the Khronos group Khronos OpenCL Working Group (2011). OpenCL aims at unifying developments on various kinds of architectures like CPUs, GPUs and even FPGAs. It provides programming constructs based upon C99 to express 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 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. We need high-level programming APIs to hide this complexity to scientists, while being able to automatically generate OpenCL kernels and initializations.

Another widespread cross-platform tool is Java and especially its virtual machine execution platform. The latter makes of Java-enabled solutions a perfect match for our needs, since any Java-based development is Write Once, Run Anywhere. Although the Java Virtual Machine (JVM) tends to become more and more efficient, the Java language itself evolves slower 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 and parallelism. Among JVM based 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. Meanwhile, functional programming copes very well with parallelism due to its immutability principles. Having immutable objects and collections highly simplifies parallel source code since no synchronization mechanism is required when accessing the original data. As many other functional languages such as Haskell, F# or Erlang, Scala has leveraged these aspects to provide transparent parallelization facilities. In the standard library, these parallelization facilities use system threads and are limited to CPU parallelism. In the same time, the ScalaCL library generates OpenCL code at compile-time and produces OpenCL-enabled binaries from pure Scala code. Consequently, it is easier to generate automatically parallelized functional programming constructs.

To sum up, OpenCL and Scala both appear like relevant tools to quickly build a various set of parallel prototypes for a given application. This study therefore benchmarks solutions where one or the two of the these technologies were used to build prototypes of parallel simulations on manycore CPU and GPU architectures. We concentrate on simulations that can benefit from data-parallelism in this study. As a matter of fact, the tools that we have chosen are designed to deal with this kind of parallelism, as opposed to task-parallelism. We intend to show that prototypes harnessing Scala's automatic parallelization frameworks display the same trend in terms of speed-up than handcrafted parallel implementations. To do so, we will:

- Introduce Scala frameworks providing automatic parallelization facilities;
- Present the models at the heart of the study and their properties;
- Discuss the results of our benchmark when faced with a representative selection of parallel platforms;
- Show the relevancy of using Scala to quickly build parallel prototypes prior to any technological choice.

2 Automatic Parallelization with Scala

2.1 Why is Scala a good candidate for parallelization of simulations?

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 Dolbeau et al. (2007), FCUDA Papakonstantinou et al. (2009) and Par4all Amini et al. (2011). Other studies Karimi et al. (2010), as well as our own experience, show that CUDA displays far better performance on NVIDIA boards than OpenCL, since it is precisely optimized by NVIDIA for their devices. However, automatically generated

CUDA cannot benefit of the same tuning quality. That is why we rather consider OpenCL code generation instead of CUDA. The former having been designed as a cross-platform technology, it is consequently better suited for generic and automatic code production.

First of all, let us make a brief recall on what is Scala. Scala is a programming language mixing two paradigms: object oriented and functional programming. Its main feature is that it runs on top of the Java Virtual Machine (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 proposal 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 that intrinsically facilitates parallelization. As a matter of fact, Scala default collections are immutable: every 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 mechanisms are involved anymore. Thus, concurrent accesses to the collection elements are not a problem anymore as long as they are read-only accesses that don't introduce any overhead. It is a classical functional programming pattern to issue a new collection as the result of applying a function to an initial 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 Okasaki (1999).

2.2 Scala Parallel Collections

Scala 2.9 release introduced a new set of Parallel collections mirroring the classical ones. They have been described in Prokopec et al. (2011). 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 `par` method 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 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 Lea (2000). This framework was released with the latest Java 7 SDK. It is based upon the divide and conquer paradigm. Fork/Join introduces the notion of tasks, which is 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. Tasks are assigned to workers' queues, but when a worker is idle, it can steal tasks from another worker's queue, which helps achieve the whole computation faster. Scala Parallel Collections implement an exponential task splitting technique detailed in Cong et al. (2008) to determine the ideal task granularity.

2.3 ScalaCL

ScalaCL is a project, part of the free and open-source *Nativelibs4java* initiative, led by Olivier Chafik. The *Nativelibs4java* 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 constructs might run slower than their classical Java equivalents. *Scalaxy* deals with this problem by pre-processing Scala code to replace some constructs 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. Up to version 0.2, a compiler plugin dedicated to OpenCL generation was called at compile time to normalize the code of the closure applied to the collection. Version 0.3, the current one, has led to a whole rewriting of the library. ScalaCL now leverages the Scala macros system, introduced in the 2.10 release of the Scala language. Thanks to macros, ScalaCL modifies the program's syntax tree during the compilation to provide transparent parallelization to manycore architectures from high level functional constructs. In both cases, 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.

3 Case study: three different simulation models

In this section, we will show how parallel Scala implementations of three different simulation models are close to the sequential Scala implementation. We compare sequential Scala code with its parallel declinations using Scala, and put the light on the automatic aspect of the two studied approaches, which respective source codes remain very close to the genuine. Our benchmark consists in running several iterations of the automatically parallelized models, and to compare them with handcrafted parallel implementations.

The three models used in the benchmark were carefully chosen so that they remain simple to describe, while being representative of several main modeling methods. They are: the Ising Model Ising (1925), a forest Gap Model Passerat-Palmbach et al. (2012) and the Schelling's segregation model Schelling (1971), and will be described more thoroughly in this section. Three categories are considered to classify the models:

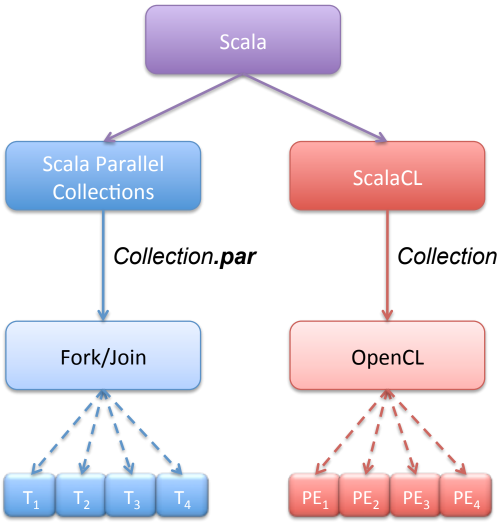

Figure 1: Schema showing the similarities between the two approaches: Scala Parallel Collections spreads the workload among CPU threads (T), while ScalaCL spreads it among OpenCL Processing Elements (PEs)

	Stepwise/Agent-based	Stochastic	Computational/Data
Ising	Stepwise	Yes	Computational
Gap Model	Stepwise	No	Data
Schelling	Agent-based	Yes	Data

Table 1: Summary of the studied models' characteristics

- Is the model stepwise or agent-based?
- Does the model outputs depend on stochasticity?
- Is the model performing more computations or data accesses?

Let us sum the characteristics of our three models according to the three aforementioned categories in Table 1.

3.1 The case of Discrete Event Simulations (DES)

Discrete-event based models do not cope well with data-parallelism approaches. In fact, they are more likely to fit a task parallel framework. For instance, Odersky et al. introduce a discrete-event model of a circuit in Odersky et al. (2008). The purpose of this model is to design and implement a simulator for digital circuits. Such a model is the perfect example of the difficulty to take advantage of parallel collections implementations when the problem is not suited. Indeed, this parallel discrete-event model implies communications to broadcast the events. Furthermore, events lead the order in which task are executed, whereas data-parallel techniques rely on independent computations. Task parallelism approaches are not considered in this work, but Scala also provides ways to easily parallelize such problems through the Scala

Actors framework Haller and Odersky (2009). The interested reader can find further details on the task-parallel implementation of the previously mentioned digital circuit model using Actors in Odersky et al. (2008).

3.2 Models description

3.2.1 Ising models

The Ising model is a mathematical model representing ferromagnetism in statistical physics Ising (1925). 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 stochastic simulation process and for every spin of the lattice, the algorithms determines whether it has to be flipped or not. The decision to flip or not the spin of a point in the lattice is taken in view of two criteria. The first is the value of the spins belonging to the Von Neumann neighborhood of the current point, and the second is a random process called the Metropolis criterion.

Ising models have been studied in many dimensions, but for the purpose of this study, we consider a 2D toric lattice, solved using the Metropolis algorithm Metropolis et al. (1953).

3.2.2 Forest Gap Model

The second model on which we apply automatic parallelization techniques is a Forest Gap Model described in Passerat-Palmbach et al. (2012). It depicts the dynamics of trees spawning and falling in a French Guiana rainforest area. The purpose of this model is to serve as a basis for a future agent-based model rendering the settling of ant nests in the area, depending on the trees and gaps locations. At each simulation step, a random number of trees will fall, carrying a random numbers of their neighbors in their fall. In the same time, new trees are spawning in the gaps area to repopulate them.

Here, we focus on the bottleneck of the model: a method called several times at each simulation step that represents about 70% of the whole execution time. Although the whole model is stochastic, the part we consider in this work is purely computational. The idea is to figure out the boundaries of the gaps in the forest. The map is a matrix of boolean wherein cells carrying a *true* value represents parts of gaps, while any other cell carries a *false* value. According to the value of its neighbors, a cell can determine whether it is part of a gap boundary or not.

3.2.3 Schelling's Segregation Model

The dynamic models of segregation of Schelling Schelling (1971) is a specialized individual based model that simulates the dynamic of two populations of distinct colors. Each individual wants to live in an area where at least a certain ratio of individuals of the same color as his own are living. Individuals that are unhappy with their neighborhood move at random to another part of the map. This model converges toward a space segregated by colors with huge clusters of individual of the same color.

At the heart of the segregation model is all the decisions taken by the individuals to move from their place to another. As long as this is the most computation intensive part of the segregation model, we will concentrate our parallelization efforts on this part of the algorithm. Furthermore, it presents a naturally parallel aspect since individuals decide whether they feel the need to move independently from each other.

3.3 Scala implementations

In this section we will focus on the implementation details of the Ising model. The other implementations would not bring more precisions concerning the use of the two Scala parallelization frameworks studied in this work.

Our Scala implementation of the Ising model represents the spin lattice by an *IndexedSeq*. *IndexedSeq* is a trait, i.e. an enhanced interface in the sense of Java that also allows methods definition; it abstracts all sorts of indexed sequences. Indexed sequences are collections that have a defined order of elements and provide efficient random access to the elements. It bears operations on this collection 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).

To harness parallel architectures, we need to slightly rewrite this algorithm. The initial version consists in trying to flip the spin of a single point of the lattice at each step. 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 at 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" Preis et al. (2009). 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.

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 the reader, 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 the parallelization APIs integrate smoothly in 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:

```
def processLattice(_lattice: Lattice)(implicit rng: Random) =

  new Lattice {
    val size = _lattice.size
    val lattice =
      IndexedSeq.concat (
        _lattice.filter{case((x, y), _) => isEven(x, y)}.map(spin => processSpin(
            _lattice, spin)),
        _lattice.filter{case((x, y), _) => isOdd(x, y)}.map(spin => processSpin(
            _lattice, spin))
      )
```

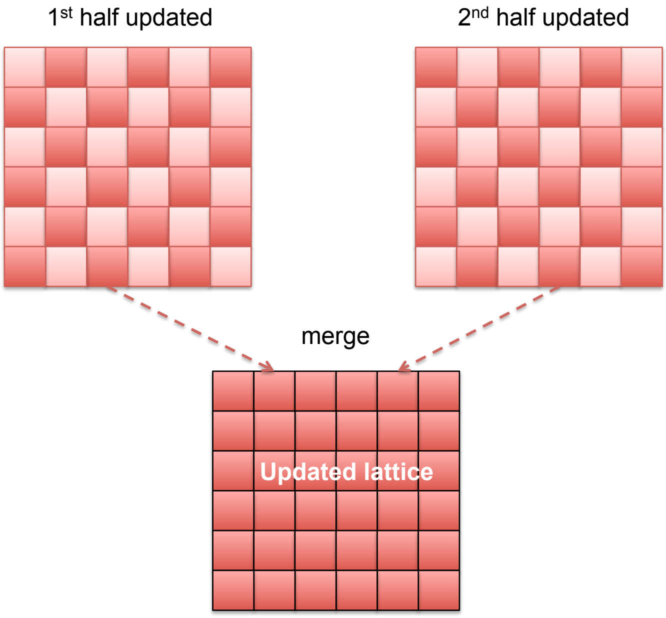

Figure 2: Lattice updated in two times following a checkerboard approach

```
  }
```

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:

```
def processLattice(_lattice: Lattice)(implicit rng: Random) =

  new Lattice {
    val size = _lattice.size
    val lattice =
      IndexedSeq.concat (
        _lattice.filter{case((x, y), _) => isEven(x, y)}.par.map(spin =>
            processSpin(_lattice, spin)),
        _lattice.filter{case((x, y), _) => isOdd(x, y)}.par.map(spin =>
            processSpin(_lattice, spin))
      )
  }
```

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

Model	**Part of the Sequential Execution Time Subject to Parallelization**	**Intrinsic Parameters**
Gap Model	70%	Map of 584x492 cells

Ising	38%	Map of 2048x2048 cells;
		Threshold = 0.5

Schelling	50%	Map of 500x500 cells;
		Part of free cells at initialization = 2%;
		Equally-sized B/W population;
		Neighbourhood = 2 cells

Table 2: Characteristics of the three studied models

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 GPUs.

As automatic parallelization applies on determined parts of the initial code, the percentage of the sequential execution time affected by the parallel declinations can be computed through a profiler. Thanks to this tool, we were able to determine the theoretical benefits of attempting to parallelize part of a given model. These figures are presented in Table 2, along with the intrinsic parameters at the heart of each model implementation. For more details about the purpose of these parameters, the interested reader can refer to the respective papers introducing each model thoroughly.

4 Results

In this section, we will study how the different models implementations behaved when run on a set of different architectures. All the platforms that were used in the benchmark are listed hereafter:

- 2-CPU 8-core Intel Xeon E5630 (Westmere architecture) running at 2.53GHz with ECC-memory enabled
- 8-CPU 8-core Intel Xeon E7-8837 running at 2.67GHz
- 4-core 2-thread Intel i7-2600K running at 3.40GHz
- NVIDIA GeForce GTX 580 (Fermi architecture)
- NVIDIA Tesla C2075 (Fermi architecture)
- NVIDIA Tesla K20 (Kepler architecture)

We compare a sequential Scala implementation executed on a single core of the Intel Xeon E5630 with its Scala Parallel Collections and ScalaCL declinations executed on all the platforms. Each model had to process the same input configurations for this benchmark. Measures resulting from these runs are displayed in Table 3.

Unfortunately, at the time of writing ScalaCL is not able to translate an application as complex as Schelling's segregation model to OpenCL yet. In the meantime, while ScalaCL managed to produce OpenCL versions of the Gap Model and the Ising Model, these two declinations were killed by the system before being able to complete their execution.

Model	**Sequential (Xeon E5630)**	**ScalaPC (i7)**	**ScalaPC (Xeon E7-8837)**	**ScalaPC (Xeon E5630)**	**ScalaCL (GTX)**	**ScalaCL (C2075)**	**ScalaCL (Kepler)**
Gap Model	150.04	75.67	128	109.42	killed by system	killed by system	killed by system
Ising	45.35	11.63	33	26.83	killed by system	killed by system	killed by system
Schelling	2961.52	525.63	344	412.86	X	X	X

Table 3: Execution times in seconds of the three models on several parallel platforms (ScalaPC stands for Scala Parallel Collections)

As a consequence, we are not able to provide any result about a ScalaCL implementation of these models. This leads to the first result of this work: at the time of writing, ScalaCL cannot be used to build parallel prototypes of simulations on GPU. It is just a proof of concept that Scala closures can be transformed to OpenCL code, but it is not reliable enough to achieve such transformations on real simulation codes yet.

On the other hand, the Scala Parallel Collection API succeeded to build a parallel declination of the three benchmarked models. These declinations have successfully run on the CPU architectures retained in the benchmark, and have shown a potential performance gain when parallelizing the 3 studied models.

Let us validate this trend by now comparing handcrafted implementations on CPU and GPU. For the sake of brevity, we will focus on the Forest Gap Model, which Scala Parallel Collections implementation runs twice as fast as the sequential implementation on the Intel i7, according to Table 3. The CPU declination uses a Java Thread Pool, whereas the GPU is leveraged using OpenCL. Figure 3 shows the speed-ups obtained with the different parallelizations of the Forest Gap Model.

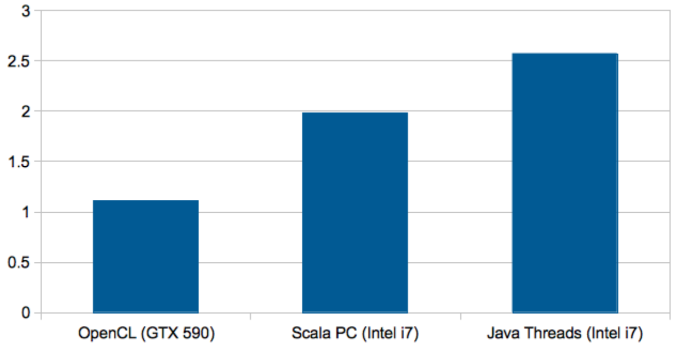

Figure 3: Speed-up obtained for the Gap Model depending on the underlying technique of parallelization

Results from Figure 3 show that an handcrafted parallelization on CPU follows the trend of the parallel prototype in terms of performance gain. In view of the lower speed-up obtained on GPU using an OpenCL implementation, it is likely that a automatically built GPU-enabled prototype would have displayed worse performance than its CPU counterpart.

Not only this last result show that the automatic prototypes approach gives significant results, when using Scala Parallel Collections, but this also validates the relevancy of the whole approach when considering the characteristics of the involved simulation model. In Table 1, the Forest Gap Model had been stated as a model performing more data accesses than computations. Thus, it is logical than a parallelization attempt on GPU suffers from the latency related to memory accesses on this kind of architecture. Indeed GPUs can only be fully exploited by applications with a high computational workload. Here, the OpenCL declination performs slightly faster than the sequential one (1.11X), but it is the perfect case of the unsuited architecture choice that leads to disappointing performance, as exposed in Introduction.

Automatically built parallel prototypes quickly put the light on this weakness, and avoid wasting time to develop a GPU implementation that would be less efficient than the parallel CPU version.

In the same way, we can see from Table 3 that the number of available cores is not the only thing to consider when selecting a target platform. Depending on the characteristics of the model, which are summed up in Table 1, parallel prototypes behave differently when faced to manycore architectures they cannot fully exploit due to frequent memory accesses. The perfect example of this trend is the results obtained when running the models on the Xeon E5630, ECC-enabled host. As fast as it can be, this CPU-based host is significantly slowed down by its ECC memory. Indeed, ECC is known to impact execution times of about 20%.

5 Conclusion

In this work we have benchmarked two Scala proposals aiming at automatically parallelizing applications and their ability to help simulation practitioners to figure out the best target platform to parallelize their simulation on. The two frameworks have been detailed and compared: Scala parallel collections that automatically creates tasks, and execute them in parallel thanks to a pool of worker threads; ScalaCL, part of the *nativelibs4java* project, which intends to produce OpenCL code from Scala closures, and to run it on the most efficient device available, be it GPU or multicore CPU.

Our study has stated that ScalaCL is still in its infancy and can only translate a limited set of Scala constructs to OpenCL at the time of writing. Although it is not able to produce a parallel prototype for a simulation yet, it deserves to be regarded as a future great language extension if it manages to improve its reliability. For its part, Scala Parallel Collections is a really satisfying framework that mixes ease of use and efficiency.

Considering data-parallel simulations, this works shows how simulation practitioners can easily determine the best parallel platform on which to concentrate their development efforts. Especially, this study puts forward the inefficiency of some architectures, in our case GPUs, when faced with problems they were not designed to process initially. In the literature, some architectures have often been favored because of their cutting-edge characteristics. However, they might not be the best solution to retain and other solutions might be underrated Lee et al. (2010). Thus, being able to quickly benchmark several architectures without further code rewriting is a great decision support tool. Such an approach is very important when speed-up matters to make sure that the underlying architecture is the most suited to fasten the problem.

In the future, we plan to propose metrics to set out the benefits of automatically built prototypes using Scala compared to handcrafted ones when considering the human resources involved and the development time. In this way, we will evaluate the possibilities to take part to the development of ScalaCL, to shorten the gap between this tool and the Scala Parallel Collections.

References

Amini, M., Coelho, F., Irigoin, F., and Keryell, R. (2011). Static compilation analysis for host-accelerator communication optimization. In *The 24th International Workshop on Languages and Compilers for Parallel Computing, Fort Collins, Colorado.*

Caux, J., Hill, D., Siregar, P., et al. (2011). Accelerating 3d cellular automata computation with gp-gpu in the context of integrative biology. *Cellular Automata-Innovative Modelling for Science and Engineering*, pages 411–426.

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

Dolbeau, R., Bihan, S., and Bodin, F. (2007). Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007).*

Haller, P. and Odersky, M. (2009). Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220.

Ising, E. (1925). Beitrag zur theorie des ferromagnetismus. *Zeitschrift für Physik A Hadrons and Nuclei*, 31:253–258. 10.1007/BF02980577.

Karimi, K., Dickson, N., and Hamze, F. (2010). A performance comparison of cuda and opencl. `http://arxiv.org/abs/1005.2581v3`. submitted.

Khronos OpenCL Working Group (2011). The opencl specification 1.2. Specification 1.2, Khronos Group.

Lea, D. (2000). A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., et al. (2010). Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., Teller, E., et al. (1953). Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087.

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala.* Artima Incorporated.

Okasaki, C. (1999). *Purely functional data structures.* Cambridge Univ Pr.

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

Passerat-Palmbach, J., Forest, A., Pal, J., Corbara, B., and Hill, D. (2012). Automatic parallelization of a gap model using java and opencl. In *Proceedings of the European SImulation and Modeling Conference (ESM).* to be published.

Preis, T., Virnau, P., Paul, W., and Schneider, J. (2009). Gpu accelerated monte carlo simulation of the 2d and 3d ising model. *Journal of Computational Physics*, 228(12):4468–4477.

Prokopec, A., Bagwell, P., Rompf, T., and Odersky, M. (2011). A generic parallel collection framework. In *Euro-Par 2011 Parallel Processing*, pages 136–147. Springer.

Schelling, T. C. (1971). Dynamic models of segregation. *The Journal of Mathematical Sociology*, 1(2):143–186.

Topa, P. and Młocek, P. (2012). Gpgpu implementation of cellular automata model of water flow. *Parallel Processing and Applied Mathematics*, pages 630–639.

UMR 6158 CNRS

RESEARCH CENTRE
LIMOS - UMR CNRS 6158

Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

Publisher
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France
http://limos.isima.fr/