Repeatability with Random Numbers Using Algorithmic Skeletons

Alexis Pereda, David R.C. Hill, Claude Mazel, Bruno Bachelet

Repeatability with random numbers using algorithmic skeletons

Alexis Pereda
David R.C. Hill
Claude Mazel
Bruno Bachelet
Université Clermont Auvergne, CNRS, LIMOS
Clermont-Ferrand, France
e-mail: {alexis.pereda, david.hill, claude.mazel, bruno.bachelet}@uca.fr

Keywords
algorithmic skeletons, parallel computing, repeatability, pseudorandom numbers, template metaprogramming

Abstract

This article presents a solution to ensure repeatability at software level when using pseudorandom numbers in parallel computations. This is achieved automatically to ease the developer, without inducing performance loss compared to a manual approach thanks to template metaprogramming.

Based on the data flow mechanism proposed in a previous work to design and execute algorithmic skeletons, we automate the correct usage of Pseudorandom Number Generator (PRNG) streams. This mechanism makes it possible to assign a PRNG stream to any part of an algorithm, and reaching repeatability can be done by providing the same random number sequence to any parallelizable task, whether it is in a parallel run, regardless of the degree of parallelism, or in a sequential one. Parallelizable tasks can easily be identified within algorithmic skeletons as they provide information about each component of their structure, accessible at compile-time.

We illustrate our solution on a metaheuristic to solve an Operational Research (OR) problem that allows several levels of parallelism.

1 Introduction

Science, and more specifically the current scientific method, requires, amongst other criteria (refutability and non-contradiction), experiments to be reproducible [9]. By reproducing a scientific experiment, one reduces (and aims to eliminate) variations on the results due to random effects, errors or even frauds. Hence, the validity of a scientific result depends on the reproducibility of the work that lead to it. Having multiple different experiments, with different contexts, leading to the same scientific conclusion is essential for "good science".

For good computer science, we also have a strong need of repeatability of each different computer experiment. Like Drummond, we found confusion in research papers between reproducibility and repeatability [12]: some computer scientists use bitwise reproducibility to mean repeatability. We want to stick to the general meaning of these terms as found in epistemology (philosophy of knowledge).

However, it has been observed that numerous publications in various fields failed to be reproduced [8], to the point it is currently known as the reproducibility crisis. Computer science is not spared by this phenomenon, and another issue is raised by the use of parallelism: the loss of repeatability, which is key to debugging software and obtaining reliable output. Repeatability is, in a sense, stronger than reproducibility: it means that, for identical input and computing environment, there should be identical results, whereas reproducibility only requires that the conclusions based on the results remain similar. Once this point is acquired, we can rely on the program results. It is a first step that is supposed granted by many since we think of computers as deterministic machines.

The problem is that parallelism causes issues in that respect at multiple levels. There is for example dynamic execution [1] that can lead to more efficient hardware parallel execution within the processor pipeline by reordering instructions. This optimization can change numerical results when using the non-associative floating point operations, thus possibly breaking repeatability. This optimization being done at hardware level, this paper will not address it.

In contrast, at software level, computer science also often requires random numbers. While their use in a sequential program is normally safe, when paired with parallelism, one must be very careful to the choice of the generator and its parallelization technique [16]. In addition, making the program repeatable with results comparable to a sequential execution is also a key point to validate a scientific program [21]. This latter problem is hard in a parallel context because to guarantee repeatability regardless of the degree of parallelism, one must ensure that each parallelizable task accesses the same random number sequence.

Programming parallel software is widely regarded as a

difficult task, and it becomes more complex because of random numbers. This has led to the production of tools to relieve developers [16]. For parallel aspects, algorithmic skeletons [2] notably provide easy to use patterns, completed with user code to be executed in parallel.

In this paper, we address the repeatability issue with random numbers in a parallel setup while limiting runtime performance loss. We have already proposed our own solution for algorithmic skeletons [24], and we show here how to automatically build a repeatable program without additional work from the developer. Our solution lies on the use of our algorithmic skeleton library data flow mechanism, called links, in order to provide each parallelizable task with its own Pseudorandom Number Generator (PRNG) stream. We test our method by solving the Operational Research (OR) Traveling Salesman Problem (TSP) using the GRASPxELS [13] metaheuristic that allows multiple levels of parallelization.

The next section presents related work on reproducibility, repeatability and algorithmic skeletons. Then, this paper summarizes our algorithmic skeleton library to introduce links, our mechanism to express data flow of algorithms. Based on this system, our solution to automatically make repeatable programs is then explained. Lastly, we show the performance run-time overhead of our solution: first the near zero overhead of the skeleton abstraction due to template metaprogramming, and second, the low overhead of enabling repeatability.

2 Related work

The non-reproducibility, and furthermore non-repeatability, in computer science comes from multiple sources [23]. First, hardware specifications matter as already stated in the introduction: hardware optimizations can induce variations in produced numerical results. More generally, different hardware can lead to different results from the same input program. Then, differences in execution context are also involved: whether it is a different operating system or even just a different version of the same operating system. Eventually, libraries are another possible source, from standard libraries provided by the system to user libraries [17]. To solve these issues, [18] proposes repositories and versioning to ensure the correct state of each software component for a given project.

Provided by libraries or by the system, pseudorandom number generators are a potential cause of non-reproducibility in a parallel context [17]. To avoid misuses, studies provide some guidelines as well as tools for a proper manipulation of PRNGs that ensures the statistical validity of the output and its reproducibility [6, 16, 19, 22]. However, repeatability when PRNGs are involved has not clearly been addressed, albeit it being of major concern to be able to debug any parallel software and to provide reliable outputs. To our knowledge, common frameworks (e.g. OpenMP, Intel Threading Building Blocks, C++ parallel standard library...) that assist developers to produce parallel code have no built-in mechanism to deal with repeatability (i.e. to guarantee a given task is always given the same random number sequence).

Algorithmic skeletons [2] have been studied for a long time to help developers writing parallel software. They consist in providing parallel patterns a program can use and complete with user code to produce automatically a parallel implementation of an algorithm. Plenty of solutions have been proposed: some are implemented as new languages [26] or as compilers [15]. These implementations offer a Domain Specific Language (DSL) (as opposed to a General Purpose Programming Language (GPPL)) designed for parallelism. While allowing more flexibility and usually resulting in better performances compared to libraries, these solutions constrain the developer to learn a new language and to dismiss the language he is used to, and which could better fit the domain requirements. A library can also offer a better extensibility: adding a new pattern to it can be as simple as writing a new class in an object-oriented language, for example.

Thus, there are libraries for various classic GPPLs, including C, C++, Java, Python [7, 11, 14, 25]. A library normally acts during the program execution and causes a time overhead. However, the C++ language, well-known for its template mechanism which enables metaprogramming, makes it possible to write active libraries [5]. An active library acts during the compilation of the program, and it is possible to take advantage of it to reduce the runtime overhead such abstraction usually incurs, as demonstrated by [10]. Moreover, paired with the C++ capability of overloading functions (and in particular operators), it is possible to conceive an Embedded Domain Specific Language (EDSL) [20], mitigating the low flexibility of libraries compared to languages by providing the user a sub-language within the host language.

Many issues have been addressed by the evolution of algorithmic skeleton implementations, for example nesting parallel patterns or type safety. However, we failed to observe any proposal that tries to help the developer in producing a parallel executable which, despite using random numbers, is guaranteed to be repeatable.

For that reason, we propose a solution that automatically ensures, when using random numbers, repeatability at software level of the produced program. To do so, we extend an existing algorithmic skeletons library by using one of its inner mechanisms that handles the data flow.

3 Algorithmic skeletons

The main objective of algorithmic skeletons [2] is to provide tools to help writing parallel code. Using algorithmic skeletons is describing an algorithm structure (named skeleton) using elements that implement specific parallel patterns, and filling it with user-written code that is fully sequential (named muscles). In [24], we proposed an al-

gorithmic skeleton library in C++, built with templates and metaprogramming. Our work depending on specifics that this library provides, this section will summarize our previous paper.

This section will be based on describing a common algorithm in Operational Research (OR), GRASPxELS [13]. This algorithm is actually the composition of two usual metaheuristics: Greedy Random Adaptive Search Procedure (GRASP) and Evolutionary Local Search (ELS), ELS being in this case the local search GRASP will use.

GRASP is presented in algorithm 1. Its goal is to find a solution S^* for a given optimization problem P by keeping the best solution of a set built by repeatedly constructing randomly, then improving, a solution.

Algorithm 1 GRASP

```
function GRASP(P)
    for i = 1..N do
        S_i ← constructiveHeuristic(P)
        S_i ← localSearch(P, S_i)
    end for
    S* ← select({S_1, S_2, ..., S_N})
    return S*
end function
```

ELS, presented in algorithm 2, is a local search and tries to improve an existing solution. It improves a given solution S by generating a set of randomly mutated solutions S_j, then improving them using another local search. The best solution S^* of all S_j is kept as the new reference solution for next iterations if better than the current one. The best solution S of all S^* is returned.

Algorithm 2 ELS

```
function ELS(P, S)
    for i = 1..N do
        for j = 1..M do
            S_j ← mutate(S)
            S_j ← localSearch(P, S_j)
        end for
        S* ← select1({S_1, S_2, ..., S_M})
        S ← select2(S, S*)
    end for
    return S
end function
```

In our implementation, algorithmic skeletons are composed of three main components: a structure, the muscles and the links. The muscles are the sequential implementations of tasks given by the end-developer, implemented as simple functions or any function-like element (e.g. functor).

3.1 Structure

The structure of a skeleton holds the execution patterns that will be used to implement the corresponding algorithm.

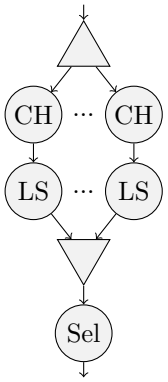

Figure 1: GRASP skeleton

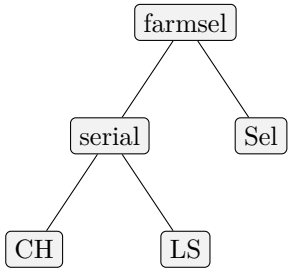

Figure 2: GRASP skeleton tree

For example, in the GRASP algorithm (cf. algorithm 1), the iterations do not depend on each other so they all can be run in parallel. This means a farm pattern [3] is fit to implement this algorithm. To be correct, it is actually a farm followed by a selection mechanism that will keep one of the results from the farm. This can be represented as in figure 1: a farm will execute independently the iterations of the loop of GRASP (in sequence, the constructive heuristic (`CH`) followed by the local search (`LS`)), and the output of the iterations (their solution for P) will be merged through the selection `Sel` of the best solution.

It is possible to similarly process ELS and conclude it is a composition of an outer iterative loop (with dependent iterations) where each iteration, the inner loop, is itself a farm.

The representation in figure 1 is explicit with what will be parallelized and useful for that matter. However, another representation, closer to how algorithmic skeletons work and how they are written, will be used here as in figure 2. In this tree, leaves are muscles and branch nodes are atomic structural elements that we call bones. The two used bones are `serial` which implements a sequential execution pattern, and `farmsel` which implements what is described above: a farm followed by a selection.

Using the tree representation, GRASPxELS is defined by replacing the `LS` from figure 2 by the skeleton tree of ELS: an iterative loop (bone: `itersel`) that executes a `farmsel` of a mutation `M` sequenced with a local search `LS` (cf. figure 3).

From this, writing the corresponding C++ code using our library is quite straightforward. Skeletons are modeled in the form of parameterized types (known as templates). Even though GRASPxELS can be defined directly as the tree of figure 3, meaning only one type definition in C++ code, listing 1 presents how to define GRASPxELS by defining separately the GRASP and ELS structures (the two first types, respectively starting at lines 1 and 10) and composing the GRASP definition with the ELS definition (third type, starting at line 23). C++ template parameters are used here to abstract the actual types of the muscles

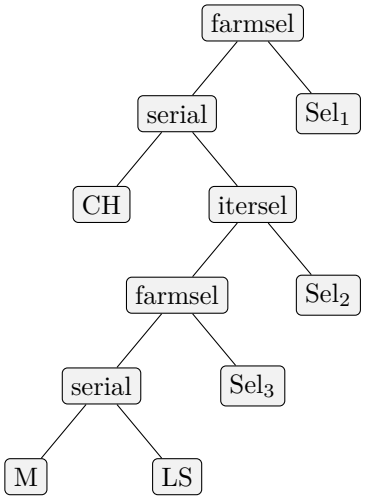

Figure 3: GRASPxELS skeleton tree
Note that Sel_1 is the GRASP Sel, and Sel_2 and Sel_3 are respectively the ELS Sel_1 and Sel_2.

(a skeleton structure is independent of this information) and to allow skeleton composition.

The first template represents the GRASP structure: a farm that executes in series two muscles (`CH` and `LS`) then selects (muscle `Sel`$_1$) the best result from its independent iterations. The `S` template comes from our library and stores structural information. The second template defines the ELS structure, that is an iterative loop that executes a farm of two muscles (`M` and `LS`) in sequence. Each level of loop doing selection of the best output from its iterations. The last template simply states that the GRASPxELS structure is the GRASP structure whose local search (the second argument) actually is an ELS structure.

Listing 1 GRASPxELS skeleton structure definition

```
1  template<
2    typename CH, typename LS, typename Sel
3  >
4  using GraspStruct =
5  S<FarmSel,
6    S<Serial, CH, LS>,
7    Sel
8  >;
9
10 template<
11   typename M, typename LS,
12   typename Sel1, typename Sel2
13 >
14 using ElsStruct =
15 S<IterSel,
16   S<FarmSel,
17     S<Serial, M, LS>,
18     Sel1
19   >,
20   Sel2
21 >;
22
23 template<
24   typename CH, typename Sel1,
25   typename M, typename LS,
26   typename Sel2, typename Sel3
27 >
28 using GraspElsStruct =
29 GraspStruct<CH, ElsStruct<M, LS, Sel2, Sel3>, Sel1>;
```

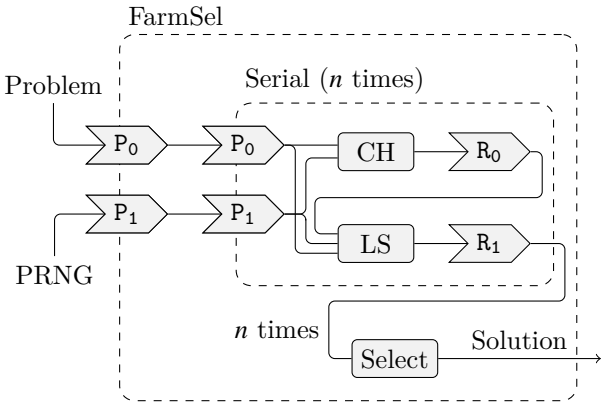

Figure 4: GRASP data flow

3.2 Links

Our algorithmic skeleton library provides a mechanism to describe how data will be transferred from one muscle to another. This feature will prove to be also useful to make repeatability possible. The listing 2 presents how to describe the links for GRASP as illustrated by figure 4. Templates are used again, keeping the same construct as for the structure template (where `S` is replaced by `L` in the code, and skeleton descriptions are completed with their function-like signatures, e.g. `A(B, C)` is a function with two parameters of types `B` and `C` and returning a value of type `A`). For example, line 7, the placeholder `P<0>` indicates that the first muscle of the `Serial` bone will have as argument the first one from its caller. This corresponds transitively to the first argument given to the overall built function-like object, that is the instance of a given problem. Line 6, the placeholder `R<1>` indicates that the value that will be returned by this bone will be the one returned by the second callee.

Listing 2 GRASP skeleton links definition

```
1  template<
2    typename Problem, typename Solution, typename PRNG
3  >
4  using GraspLinks =
5  L<FarmSel, Solution(Problem const&, PRNG&),
6    L<Serial, R<1>(P<0>, P<1>),
7      Solution(P<0>, P<1>),
8      Solution(P<0>, R<0>, P<1>)
9    >,
10   Solution(Solution const&, Solution const&)
11 >;
```

The `FarmSel` bone handles itself which arguments are given to its second muscle, the selector, hence the usage of direct types instead of placeholders at line 10. The absence of placeholders at line 5 is because of two reasons. The first one is the same as line 10, for the returned value: this is constrained by the bone. The second one concerns the parameter list. This is the outer most link, so it will get its parameters directly from its caller, which is not within the skeleton: this layer corresponds to the function-like produced object.

4 Repeatability

As described in the previous section, GRASPxELS makes use of random numbers, first to build random initial solutions, then to mutate solutions. This means the results obtained from executing GRASPxELS will depend on how the random numbers are obtained. To ensure that the result will be repeatably the same, even when running in parallel the program, it becomes essential to correctly manage random numbers.

For each task requiring random numbers, there are three possibilities to access a Pseudorandom Number Generator (PRNG):

1. a global PRNG stream shared by all tasks;
2. one PRNG stream per thread;
3. one PRNG stream per parallelizable task.

The first solution implies that two executions of the program could (and will highly likely) produce different results. This is because of the concurrent access to the PRNG stream by the threads. The figure 5 illustrates the issue by showing how two different executions will almost certainly lead to different random number sequences with a shared PRNG stream.

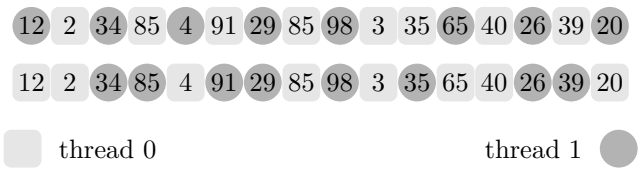

Figure 5: Two possible random number sequences for two threads sharing the same PRNG stream

The second solution can be repeatable only for a fixed degree of parallelism (i.e. for a determined number of threads), and as long as all parallelizable tasks are executed in the same order in a thread at any run. This depends on the orchestration of the tasks: with thread pools [4] for instance, the assignment of a task to a thread is dynamic and the order of execution of the tasks on a thread is not guaranteed to be the same between runs. In our solution, we ensure tasks are always executed on the same thread and with the same order [24], thanks to algorithmic skeletons analysis at compile-time, hence ensuring repeatability at fixed degree of parallelism.

The last solution, where each task that could be run in parallel gets its own PRNG stream, ensures repeatability in the case of varying degree of parallelism. However, it could lead to the issue of non-scientific soundness of the results. To guarantee the scientific soundness, it is required to build independent streams [6, 16, 22].

One common way to overcome this issue is, first, to build independent streams, and then, to manage to distribute them to the tasks. Functions that use random numbers have to be designed with a PRNG as parameter that becomes their sole source of randomness. With our skeleton library, this solution can be described with links as we already did in listing 2.

Note that this solution is not a correct version yet for repeatability because, as is, the same PRNG is transfered from the top skeleton to all the muscles, corresponding to a global PRNG stream shared by all tasks. Our library, thanks to the analysis of the algorithmic skeletons, can associate to each task to be executed in parallel a unique identifier (a number $id \in [\![0, N[\![$ where N is the total number of tasks). This is done using the known structure of the algorithm: when a sequential bone (e.g. `Serial`) is encountered, all its tasks are given the same identifier; contrarily a potentially parallel bone will associate to each parallelizable subtask a different identifier.

To illustrate our assignment for GRASPxELS, the figure 6 represents its skeleton tree so that each level of parallelization is explicitly shown with the number of iterations for each `FarmSel`.

The root node ($\texttt{farmsel}_1$) is given the identifier 0, which is transmitted as is to the associated selection task $\texttt{Sel}_1$, because it will be executed in sequence after the rest of this bone. To determine which identifiers will be given to the first parallel task (that is one of the a $\texttt{serial}_1$), we must go through the whole tree from this node to detect any other parallelizing node inside, which in this case will be $\texttt{farmsel}_2$. This is required to acquire its number of iterations (here, b), and therefore the number of parallel tasks it can spawn.

If multilevel parallelization was found, i.e. a parallelizing node P_1 having an offspring P_2 that is also a parallelizing node, the number of iterations (say x) of the top-level parallelizing node P_1 is multiplied by the number of iterations (say y) of the parallelizing node P_2, i.e. $x \times y$.

This result is used to space the identifiers given to the a $\texttt{serial}_1$ tasks created by the root node: knowing that each iteration of $\texttt{farmsel}_1$ can possibly run b parallel tasks, these identifiers thus are $\{0, b, 2b, ..., (a-1)b\}$.

All tasks in the second layer (the first parallelization level) coming from the same instance of $\texttt{serial}_1$ will share the same identifier because none will be executed simultaneously (note that all these tasks are effectively executed in parallel a times, parallel instances of each one of them have a distinct identifier). Then, for the third layer (the second parallelization level), the $\texttt{serial}_2$ having no parallelizable bone in its children, the spacing between affected identifiers will be 1 and will start from its parent identifier. The $\texttt{farmsel}_2$ identified by 0 will spawn the $\texttt{serial}_2$ tasks from 0 to $b-1$, the next one from b to $2b-1$ and so on. This method ensures that at any time, no identifier is shared by multiple concurrent tasks.

Depending on this number, it is possible to provide a unique set of local parameters (including a PRNG stream) that are hence guaranteed to be accessed only by one concurrent task at a time. All elements of the set (in particular all PRNG streams) are constructed beforehand. By doing this, it is possible to ensure all PRNG streams to be independent from each other.

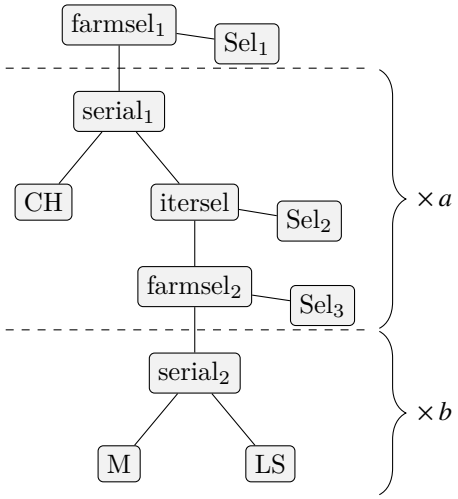

Figure 6: GRASPxELS tree highlighting parallelizable levels

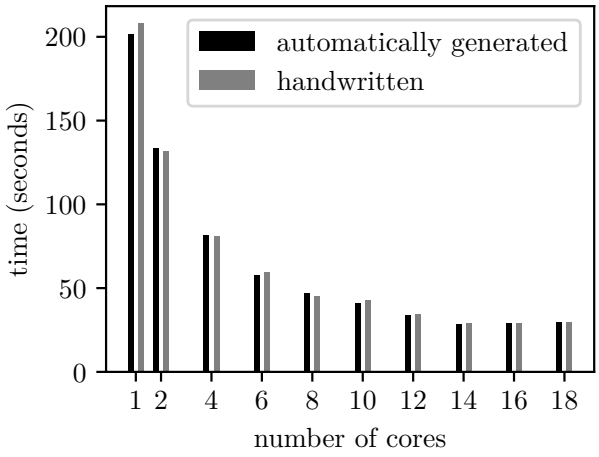

Figure 7: Execution time with varying number of cores for handwritten and automatically generated programs (on a single processor)

With that procedure in mind, what is left to do is providing a way to give the correct PRNG stream to each task requiring one. In addition to existing argument placeholders (`P<>` and `R<>`), the library now supports a third, independent, category to handle passing other arguments. This includes a PRNG stream by using the new placeholder `PRNG`.

The correct implementation will be generated when using the links description shown in listing 3. The differences with listing 2 are quite subtle: the tasks making use of a PRNG stream are now using the placeholder `PRNG` instead of `P<1>`. This is because now, the actual PRNG stream will be provided directly by the library to guarantee repeatability. In consequence, the `Serial` no longer has to know about passing a PRNG stream (so the `P<1>` placeholder has been removed), neither does `FarmSel`.

Listing 3 GRASP skeleton links definition for automatic PRNG stream management

```
template<typename Problem, typename Solution>
using GraspLinks =
L<FarmSel, Solution(Problem const&),
  L<Serial, R<1>(P<0>),
    Solution(P<0>, PRNG),
    Solution(P<0>, R<0>, PRNG)
  >,
  Solution(Solution const&, Solution const&)
>;
```

This solution is totally invisible for the muscles implementations as they will continue to obtain as usual some PRNG stream as argument. The only difference is that now, they will always receive one that makes the executions repeatable, whether it is mono- or multi-threaded, regardless of the degree of parallelism.

The actual type of the PRNG is left to be defined by the developer, with a sensible default value. Serious PRNGs are repeatable and statistically sound parallelization methods can be found in [16].

5 Performance results

One objective of our implementation is that it is competitive with handwritten solutions, because losing performances in exchange for programming ease is better avoided. The run-time overhead of the original algorithmic skeleton library has already been showed negligible [24]. However, to validate that ensuring repeatability automatically does not lead to increased overhead, we compared handwritten repeatable GRASPxELS with the automatically generated counterpart, both applied to the same Traveling Salesman Problem (TSP) instance (graph with 194 nodes, 24 GRASP iterations, 20 ELS outer iterations and 20 ELS inner iterations). All tests have been performed using an Intel Xeon E7-8890 v3 CPU at 2.5 GHz, with 72 physical cores (4×18). Both programs were compiled by g++ 8.2.0 using the same flags, amongst them the `O2` optimization pack. All results are produced by means of 20 runs each using a different initialization status for each PRNG. For each tested method (handwritten and automatically generated) the same 20 random initialization statuses are used.

Figure 7 shows results obtained when running a sequential and a parallel version (from 2 to 18 allotted cores) of both the handwritten and automatically generated programs. The first version has required meticulous development from the end-developer to ensure repeatability, whereas the algorithmic skeletons automatically produce a repeatable and parallel code from the muscles provided by the end-developer. Despite this abstraction, we observe no significant overhead (less than 3%, see figure 8). It was expected because the main work on algorithmic skeletons is done during compile-time using template metaprogramming. Templates also help generating code that is really similar to the fine tuned handwritten version.

It is important to notice that a metaheuristic like GRASPxELS makes use of random numbers to move in

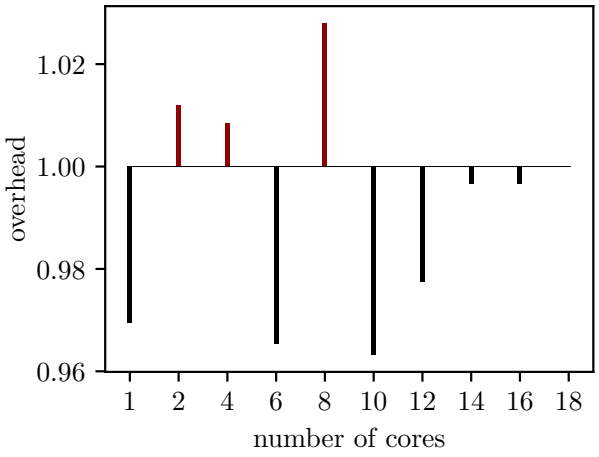

Figure 8: Overhead of automatically generated program compared to its handwritten counterpart (on a single processor)

the solution space so the quantity of visited solutions by the inner local search can vary with changes on random number sequences. Hence differences in execution time appear when the random initialization status is changed or when the repeatability is disabled. In our instance, we measured a standard deviation of 3.8 s with a mean time of 195 s for a sequential run.

6 Conclusion

In this paper, we presented our solution to automatically ensure repeatability with algorithmic skeletons. We summarized the functioning of our algorithmic skeletons library and we explained how we use its data flow mechanism to automatically provide each parallelizable task with its own Pseudorandom Number Generator (PRNG) stream so that, independently of the degree of parallelism, it runs identically and produces the same output to reach our repeatability goal.

From the structure of the algorithm described by algorithmic skeletons, we were able to detect parallelization possibilities (e.g. multilevel parallelization) and to distribute PRNG streams to muscles so that repeatability is guaranteed. Template metaprogramming is used to do most of the work on algorithmic skeletons at compile-time. Moreover, templates makes it possible to generate code close to a handwritten and dedicated one, nearly cancelling the possible abstraction overhead.

Our strategy is to assign a PRNG stream to each task that could be independently run in parallel. It could use much more streams than what is actually necessary to ensure repeatability in a known specific context. Considering the capabilities of available machines and the scenarios of parallelization that are setup to run applications, we are studying a solution to automatically control the number of distributed streams as future work.

References

[1] Robert M. Keller. “Look-ahead processors”. In: *ACM Computing Surveys* 7.4 (Dec. 1975), pp. 177–195. ISSN: 03600300. DOI: 10.1145/356654.356657. (Visited on 06/23/2020) (cit. on p. 1).

[2] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. Cambridge, MA, USA: MIT Press, 1989. ISBN: 978-0-262-53086-6 (cit. on p. 2).

[3] Duncan K. G. Campbell. *Towards the classification of algorithmic skeletons*. en. Tech. rep. YCS 276. University of York department of computer science YCS, 1996 (cit. on p. 3).

[4] Douglas C. Schmidt. “Evaluating architectures for multithreaded object request brokers”. In: *Communications of the ACM* 41.10 (Oct. 1998), pp. 54–60. ISSN: 00010782. DOI: 10.1145/286238.286248. (Visited on 07/16/2020) (cit. on p. 5).

[5] Todd L. Veldhuizen and Dennis Gannon. “Active libraries: rethinking the roles of compilers and libraries”. en. In: *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*. SIAM Press, 1998, pp. 286–295 (cit. on p. 2).

[6] Makoto Matsumoto and Takuji Nishimura. “Dynamic Creation of Pseudorandom Number Generators”. In: *Monte-Carlo and Quasi-Monte Carlo Methods 1998*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 56–69. ISBN: 978-3-540-66176-4. DOI: 10.1007/978-3-642-59657-5_3 (cit. on pp. 2, 5).

[7] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. “Flexible skeletal programming with eSkel”. en. In: *Euro-Par 2005 Parallel Processing*. Vol. 3648. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 761–770. ISBN: 978-3-540-28700-1. DOI: 10.1007/11549468_83. (Visited on 06/14/2018) (cit. on p. 2).

[8] John P. A. Ioannidis. “Why most published research findings are false”. en. In: *PLoS Medicine* 2.8 (Aug. 2005), e124. ISSN: 1549-1676. DOI: 10.1371/journal.pmed.0020124. (Visited on 07/09/2020) (cit. on p. 1).

[9] Karl Popper. *The logic of scientific discovery*. en. Google-Books-ID: LWSBAgAAQBAJ. Routledge, Nov. 2005. ISBN: 978-1-134-47002-0 (cit. on p. 1).

[10] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. “Quaff: efficient C++ design for parallel skeletons”. en. In: *Parallel Computing*. Algorithmic Skeletons 32.7 (Sept. 2006), pp. 604–615. ISSN: 0167-8191. DOI: 10.1016/j.parco.2006.06.001. (Visited on 04/23/2018) (cit. on p. 2).

[11] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. “A library of constructive skeletons for sequential style of parallel programming”. en. In: *Proceedings of the 1st international conference on Scalable information systems - InfoScale '06*. Hong Kong: ACM Press, 2006, pp. 287–302. ISBN: 978-1-59593-428-4. DOI: 10.1145/1146847.1146860. (Visited on 07/09/2020) (cit. on p. 2).

[12] Chris Drummond. “Replicability is not reproducibility: nor is it good science”. en. In: *Proceedings of the Evaluation Methods for Machine Learning Workshop*. 26th International Conference for Machine Learning. Montreal, Quebec, Canada, 2009, pp. 696–701 (cit. on p. 1).

[13] Christian Prins. “A GRASP x Evolutionary Local Search hybrid for the Vehicle Routing Problem”. en. In: *Bio-inspired Algorithms for the Vehicle Routing Problem*. Studies in Computational Intelligence. Springer, Berlin, Heidelberg, 2009, pp. 35–53. ISBN: 978-3-540-85152-3. DOI: 10.1007/978-3-540-85152-3_2. (Visited on 05/17/2018) (cit. on pp. 2, 3).

[14] Mario Leyton and José M. Piquer. “Skandium: multi-core programming with algorithmic skeletons”. In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. Print ISBN: 978-1-4244-5672-7. Pisa: IEEE, Feb. 2010, pp. 289–296. ISBN: 978-1-4244-5673-4. DOI: 10.1109/PDP.2010.26. (Visited on 02/03/2019) (cit. on p. 2).

[15] Cedric Nugteren and Henk Corporaal. “Introducing "Bones": a parallelizing source-to-source compiler based on algorithmic skeletons”. en. In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units - GPGPU-5*. London, England: ACM Press, 2012, pp. 1–10. ISBN: 978-1-4503-1233-2. DOI: 10.1145/2159430.2159431. (Visited on 02/06/2019) (cit. on p. 2).

[16] David R. C. Hill, Claude Mazel, Jonathan Passerat-Palmbach, and Mamadou K. Traore. “Distribution of random streams for simulation practitioners: CPE HPCS 2010 special issue submission”. en. In: *Concurrency and Computation: Practice and Experience* 25.10 (July 2013), pp. 1427–1442. ISSN: 15320626. DOI: 10.1002/cpe.2942. (Visited on 07/10/2020) (cit. on pp. 1, 2, 5, 6).

[17] V. T. Dao, L. Maigne, V. Breton, H. Q. Nguyen, and David R. C. Hill. “Numerical reproducibility, portability And performance of modern pseudo random number generators: preliminary study for parallel stochastic simulations using hybrid Xeon Phi computing processors”. In: *European simulation and modelling conference*. Porto, Portugal, 2014, pp. 80–87. (Visited on 07/09/2020) (cit. on p. 2).

[18] Ivo Jimenez, Carlos Maltzahn, Adam Moody, and Kathryn Mohror. *Redo: reproducibility at scale*. en. Tech. rep. UCSC-SOE-14-12. University of California Santa Cruz, 2014, p. 2 (cit. on p. 2).

[19] Timothy D. Andersen and Michael Mascagni. “Memory Efficient Lagged-Fibonacci Random Number Generators for GPU Supercomputing”. In: *Monte Carlo Methods and Applications* (June 2015), pp. 163–174 (cit. on p. 2).

[20] Daniel Berenyi. “C++ EDSL for parallel code generation”. In: *2015 conference grid, cloud & high performance computing in science (ROLCG)*. Cluj-Napoca, Romania: IEEE, Oct. 2015, pp. 1–5. ISBN: 978-606-737-040-9. DOI: 10.1109/ROLCG.2015.7367231. (Visited on 07/09/2020) (cit. on p. 2).

[21] David R. C. Hill. “Parallel random numbers, simulation, and reproducible research”. In: *Computing in Science & Engineering* 17.4 (July 2015), pp. 66–71. ISSN: 1521-9615. DOI: 10.1109/MCSE.2015.79. (Visited on 07/13/2020) (cit. on p. 1).

[22] Pierre L’Écuyer. “Random Number Generation with Multiple Streams for Sequential and Parallel Computing”. In: *2015 Winter Simulation Conference (WSC)*. 2015 Winter Simulation Conference (WSC). Huntington Beach, CA: IEEE, Dec. 2015, pp. 31–44. ISBN: 978-1-4673-9743-8. DOI: 10.1109/WSC.2015.7408151 (cit. on pp. 2, 5).

[23] David R. C. Hill. “Numerical reproducibility of parallel and distributed stochastic simulation using high-performance computing”. en. In: *Computational Frameworks*. Elsevier, 2017, pp. 95–109. ISBN: 978-1-78548-256-4. DOI: 10.1016/B978-1-78548-256-4.50004-1. (Visited on 07/09/2020) (cit. on p. 2).

[24] Alexis Pereda, David R. C. Hill, Claude Mazel, and Bruno Bachelet. “Modeling Algorithmic Skeletons for Automatic Parallelization Using Template Metaprogramming”. In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. Dublin: IEEE, July 2019, pp. 265–272. DOI: 10.1109/HPCS48598.2019.9188128 (cit. on pp. 2, 5, 6).

[25] Jolan Philippe and Frédéric Loulergue. “PySke: algorithmic skeletons for Python”. In: *The 2019 International Conference on High Performance Computing & Simulation (HPCS)*. Dublin, Ireland, July 2019, pp. 40–47. (Visited on 06/25/2020) (cit. on p. 2).

[26] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. “Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons”. en. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. Limassol Cyprus: ACM, Apr. 2019, pp. 1534–1543. ISBN: 978-1-4503-5933-7. DOI: 10.1145/3297280.3297434. (Visited on 06/10/2020) (cit. on p. 2).