Geneva 1.6: Improving the Performance of Highly Concurrent Workloads in Parametric Optimization

Dr. Rüdiger Berlich[*]
Gemfony scientific UG (haftungsbeschränkt)
E-mail: `r.berlich@gemfony.eu`

Dr. Sven Gabriel
Gemfony scientific UG (haftungsbeschränkt)
E-mail: `s.gabriel@gemfony.eu`

Dr. Ariel García
Gemfony scientific UG (haftungsbeschränkt)
E-mail: `a.garcia@gemfony.eu`

This paper discusses strategies for decreasing execution times of highly concurrent workloads in parametric optimization, on the example of version 1.6.1 of Geneva, a collection of optimization algorithms that focusses on problem domains with particularly long running evaluation functions. Particular emphasis lies on the *Courtier* broker architecture used for parallelization in environments ranging from many-core systems and GPGPU to Grids and Clouds. The paper also introduces other means that have proven to be useful for the reduction of overall compute time, such as acyclic creation of random numbers or avoidance of regions of parameter space that are known to lead to invalid results.

International Symposium on Grids and Clouds 2015
15-20 March 2015
Academia Sinica, Taipei, Taiwan

[*]Speaker.

1. Introduction and Context

The acronym *Geneva* stands for **G**rid-**e**nabled **e**volutionary **a**lgorithms. The library was initially developed for the parametric optimization of particle physics analysis [4, 6, 8], but is today also used in the automotive industry, for the optimization of the acoustics of exhaust systems [5]. The term *parametric optimization* refers to procedures that, through iterative evaluation of candidate solutions, search for those settings that lead to optimal solutions. If only a single evaluation criterion is used, searching for optimal solutions can be likened to the minimization or maximization of a mathematical function. However, in the general case, many standard mathematical procedures cannot be applied, as the evaluation of a parameterset might be performed in program code, not through a mathematical function. Hence the only information available to an optimization algorithm could be the numeric evaluation of a parameter set at a given location in the parameter space. Metaheuristic optimization algorithms, such as Evolutionary Algorithms, are nevertheless quite capable of finding *sufficiently* good solutions to a problem, however *without* usually being able to yield the theoretically best result. Additional computing power may help, however, as more iterations will allow the optimization algorithm to perform a more thorough search. Likewise, where an optimization algorithm does not mandate a fixed *population* size, larger populations will yield better results. Minimally useful population sizes will strongly depend on the size of the parameter space, as will the number of iterations until a sufficient optimum is found. The evaluation of individuals is then by far the most frequent action that an optimization algorithm needs to perform. Here the term "population" refers to all *individuals* constituting a given iteration, and "individual" refers to a set of parameters, combined with one or more evaluation criteria. Where, as is the case for Geneva, algorithms target particularly complex and computationally expensive evaluation functions, optimization runs may last for hours or even days. Reducing overall execution times then becomes critically important. In order to achieve this goal, Geneva allows parallel execution on many-core systems, Cluster, Grid and Cloud, as well as through an optional GPGPU-backend. This is of particular importance for long running evaluation functions, as parallelization may happen most easily on the level of the evaluation of candidate solutions, and may be implemented independently of the actual optimization algorithm. Note, though, that the discussion in this paper goes beyond the design chosen to allow efficient execution on a wide range of parallel devices, as other factors may directly impact overall execution times.

As performance figures were presented in earlier papers [7], this text focusses on the architecture of Geneva and its sub-libraries. Note that, in the following, the term "*Geneva*" may either refer to the "*Geneva library collection*" or a sub-library thereof with the same name. Geneva started as a single library of optimization algorithms. Over the course of Geneva's now over 20 years of development (under different names), a lot of functionality was outsourced to seperate libraries that might be helpful for other use cases. For historic reasons, though, the Geneva library collection and the Geneva optimization library still share the same name.

We will commence with a discussion of the overall architecture of Geneva, as it is the intention of the authors to encourage users to explore other deployment scenarios of the (sub-)libraries.

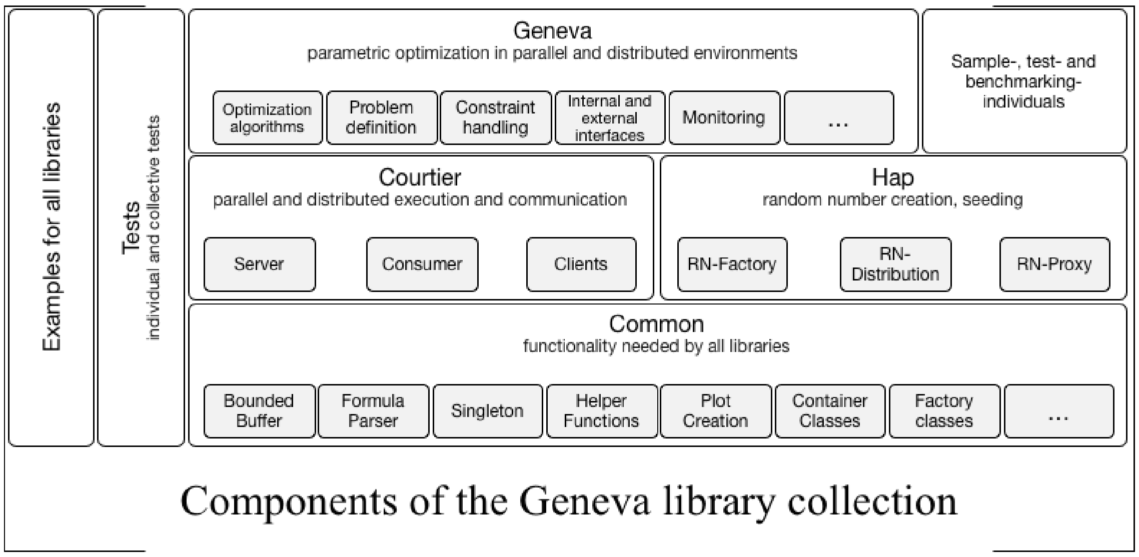
Examples for all libraries
Tests
individual and collective tests
Geneva
parametric optimization in parallel and distributed environments
Optimization algorithms
Problem definition
Constraint handling
Internal and external interfaces
Monitoring
...
Sample-, test- and benchmarking-individuals
Courtier
parallel and distributed execution and communication
Server
Consumer
Clients
Hap
random number creation, seeding
RN-Factory
RN-Distribution
RN-Proxy
Common
functionality needed by all libraries
Bounded Buffer
Formula Parser
Singleton
Helper Functions
Plot Creation
Container Classes
Factory classes
...
Components of the Geneva library collection

Figure 1: A summary of the modules contained in the Geneva library collection

2. Architecture and Ecosystem

In Geneva 1.6.1, all libraries are implemented according to the C++98-standard[1]. The only external dependency is on the Boost library collection [2][2]. Geneva runs on Unix(-like) systems, such as Linux, BSD and MacOS/Darwin, with the g++ and clang++ compilers. An experimental port to Microsoft Visual C++ exists. Apart from the standard x86-64 architecture, Geneva may also be compiled on some ARM architectures under Linux. The library is designed as a toolkit, trying to enable an as wide range of deployment scenarios as possible, giving users a lot of flexibility, but at the price of additional work needed for integration on their side. The code is freely available under an Open Source license [1] and may thus be extended or altered by users, subject to the terms and conditions of the GNU Affero General Public License v3.

Geneva's design makes no assumptions regarding the type of optimization problem other than that evaluation function(s) might be computationally expensive. Hence, in the library code, fault tolerance and code stability are rated higher than code efficiency (in the sense of execution times of individual code segments, not efficiency of the optimization algorithms). If optimization runs may optionally last for days, a crash inside of the Geneva code (as opposed to user-code in the individuals) must be avoided wherever possible.

Individuals may be defined either as pure program code that is linked with the Geneva library and is based on a class hierarchy provided for this purpose, or may optionally comprise external programs (such as simulations) for the evaluation step. Apart from the eponymous *evolutionary algorithms*, Geneva today also supports particle swarm optimization, simulated annealing, gradient descents and parameter scans [3]. Further algorithms are planned. They have in common the need for the evaluation of virtually independent candidate solutions in successive iterations. The library was

[1]The current trunk version [1] in the repository uses the C++11-standard.

[2]In the ongoing C++11-port of Geneva, these dependencies have been reduced.

tested with up to 5000 parameters per individual, and allows floating point, boolean and integer parameters[3] for the problem description. A population may optionally consist of thousands of individuals[4]. Figure 1 shows the overall architecture of Geneva, with its sub-libraries and their corresponding roles:

- The *Common*-library holds functionality of a general nature that is needed by all parts of the library collection. Of particular importance is an implementation of a thread-safe bounded buffer (used e.g. in the creation of random number packages described in section 4 and in the submission of work items discussed in section 3), a parser for C-style mathematical functions used in constraint handling (compare section 5.1), and plot creation for the ROOT analysis framework.
- Parallelization happens via the *Courtier*-library. The framework currently supports networked execution through Boost.ASIO, multi-threaded execution on many-core systems via Boost.Thread[5], as well as an optional OpenCL backend for evaluation on GPGPUs. *Courtier* does not depend on the "optimization" use-case and could be used in additional contexts. It is implemented as a template library that makes some assumptions about the API of the work items submitted through it, but apart from that has no knowledge about their nature.
- The *Hap*-library implements a random number factory that allows acyclic creation of random numbers. These may be used by random number proxies in local clients to produce random numbers of arbitrary distributions.
- The *Geneva*-library holds all functionality directly related to parametric optimization. Optimization algorithms themselves only form a fraction of the code, though[6]. Geneva in addition covers constraint handling and monitoring as important components. In a nutshell, the definition of an optimization problem requires a specification of the parameter types together with one or more associated figures of merit. Geneva also holds a rich collection of parameter types, ranging from boolean over integer to floating point types. All of Geneva is based on smart pointers[7]. Individuals are added to optimization algorithms through an interface with a std::vector API, likewise parameters are added to individuals through the same API. Apart from parameters, some algorithms, such as Evolutionary Algorithms and Simulated Annealing, require "adaptors". Geneva holds a rich collection of these.
- Last but not least there is test code, comprising both unit- and manual tests, many examples and sample individuals for benchmarking, tests and illustration of common approaches for the expression of an optimization problem. All properties of an optimization problem are expressed through the individual – problem definition is indepdendent from the optimization agorithm that modifies the parameters of individuals and calls their evaluation function(s).

[3]However, some algorithms, such as gradient methods, will only modify fitting parameter types.

[4]The number of individuals is fixed for some algorithms (e.g. gradient methods), but is free for others.

[5]This dependency will be replaced by std::thread as part of the C++11-port

[6]. . . which implies that implementing new algorithms is relatively easy – these may then use the entire infrastructure provided by the Geneva library collection.

[7]boost::shared_ptr<> up until Geneva 1.6.1, std::shared_ptr<> thereafter

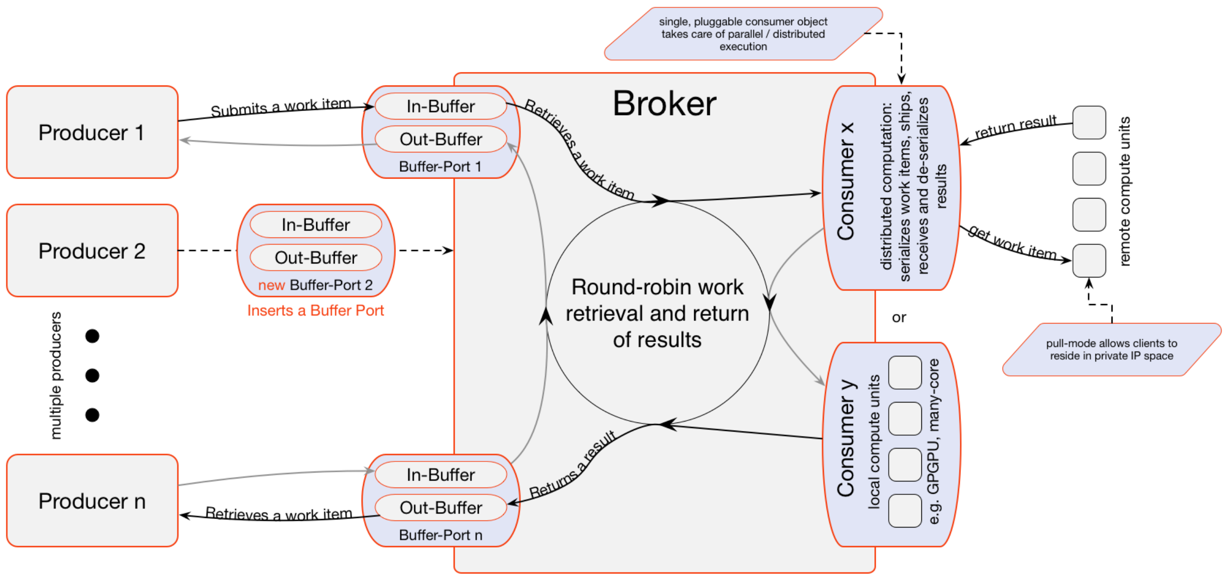

Figure 2: Principles of the Courtier library

3. Parallelization

The most straight-forward approach to decrease execution times of parametric optimization runs is to evaluate the individuals of a given iteration in parallel. Parallelization is in this case close to the “embarrassingly parallel” type, as the evaluation of two individuals will usually not depend on each other. Note, though, that successive iterations will usually depend on the results of previous iterations. Where execution times of the evaluation step are sufficiently large, parallelization may scale almost linearly even in distributed environments, such as clusters and Grid/Cloud, up to the number of individuals in an iteration. Our tests have shown that, depending on the number of parameters to be transferred, an evaluation time of a 4-5 seconds per individual will yield an almost linear speedup [7].

The maximum theoretical speedup is often described through Amdahl’s law [9], in which the ratio of execution times of parallelizable code compared to overall execution times in serial execution plays an important role. In a nutshell, according to Amdahl[8], if half of the execution time is parallelizable up to the point where results are available instantaneously, the maximum speedup[9] through parallelization will never be able to exceed a factor of 2. This fundamental rule has direct consequences for the paraellization of Geneva. Our tests have shown that a significant portion of the server-overhead stems from the (de-)serialization of candidate solutions.

Hence it becomes important for Geneva to not only support distributed execution but, whereever possible, local parallelization (e.g. through multi-threading or the use of a GPGPU backend), so that network communication and (de-)serialization is avoided. By the same token, for parallel work-loads that are sufficiently “heavy” for networked execution, Geneva should be capable to

[8]Communication overhead is neglected here

[9]Note that there may also be quantization effects. In a nutshell, when executing 11 work items on 10 compute nodes, the execution time can be twice as long as for 10 work items.

support an as wide array of technologies as possible. This led to the broker architecture called "*Courtier*" shown in figure 2. Geneva supports multiple, simultaneous optimization algorithms that may concurrently submit individuals for evaluation to the Courtier library. Work items are submitted to the "In-Buffer" of a an object called "buffer-port". Each producer[10] has its own buffer port. The actual parallelization is implemented in pluggable modules called "*consumer*".

In the simplest case of serial execition, a consumer will simply ask the broker for a single work item. The broker will then query the available buffer ports in a round-robin fashion for work items, until it finds one, and will hand it to the consumer. The consumer will then initiate processing of the work item through a pre-defined API-call (which constitutes the only knowledge the consumer has about the work item). Once processing is done, the result is returned to the broker, which uses a unique id associated with the buffer port to sort the work item back into its "Out-Buffer", from where the original "producer" may take it for further processing. All interaction happens through smart-pointers, so that no memory leaks occur during this process. Consumers may also choose to perform processing in parallel instead, e.g. through local processing units (multiple CPU cores or a GPU – provided the individual supports OpenCL). Depending on the consumer, it may also respond to remote clients that ask the consumer for work.

Geneva currently has a serial consumer (mostly for debugging purposes), a multi-threaded consumer, a consumer for networked execution through Boost.ASIO, as well as an optional OpenCL-consumer. In the case of the network consumer, remote clients may be transmitted through means of Grid- or Cloud-environments or even a batch submission system, and will work in a pull mode. So remote clients will contact the server for work and will initiate return of results. Through this setup networked Geneva-clients may reside in a private IP space, e.g. on some remote cluster. The only system which needs to be constantly reachable is the server.

The only concession for this setup on the side of the optimization algorithms is fault-tolerance, as it may not always be expected that all transmitted individuals actually return. So optimization algorithms must be able to repair themselves, if not all work items have returned, and there must be a timeout, after which work-items are considered to be "lost"[11]. The timeout is calculated on the basis of average return times, so varying execution times due to compute units of different speed and load are taken into account. Note, though, that this setup will not react well to vastly different, unforseeable execution times on the client side. In this case the user has the option to completely disable the timeout or to apply a factor to the timeout (which may happen through a configuration file) to take into account the worst case.

In summary, all parallel execution is triggered by consumers. Producers do not initiate parallel execution and do indeed not need to have any knowledge about the type of parallel execution that was chosen by the user. The courtier library is also independent of the original optimization use-case and may be used in other scenarios.

Current research on the Courtier library involves a "client-side" ping in order to influence the time-out, as well as asynchronous transfer of individuals between client and server. In the latter scenario, network transfers, including (de-)serialization, may happen while another individual is being evaluated. This scenario may mask the time needed for data transfers entirely, provided that

[10] A "producer" equals an optimization algorithm in this context

[11] Geneva nevertheless tries to integrate "late" returns even after the timeout in most algorithms.

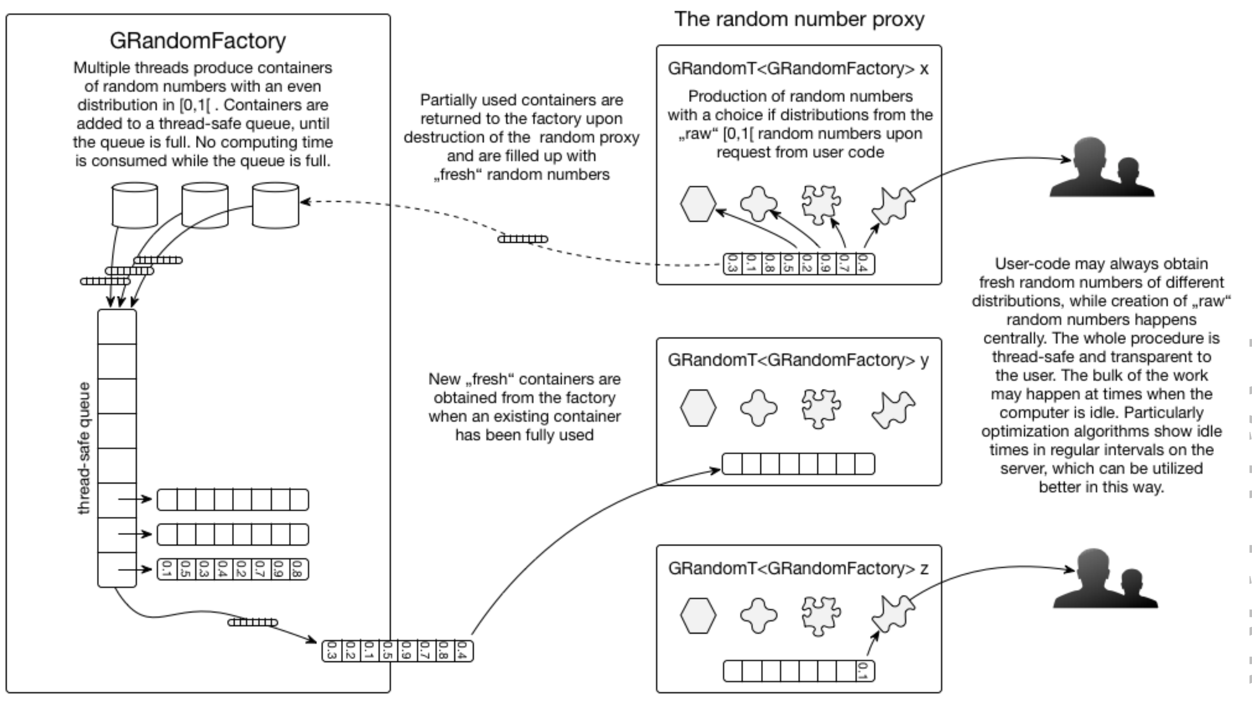
GRandomFactory
Multiple threads produce containers of random numbers with an even distribution in [0,1[. Containers are added to a thread-safe queue, until the queue is full. No computing time is consumed while the queue is full.
thread-safe queue
Partially used containers are returned to the factory upon destruction of the random proxy and are filled up with „fresh“ random numbers
New „fresh“ containers are obtained from the factory when an existing container has been fully used
The random number proxy
GRandomT<GRandomFactory> x
Production of random numbers with a choice if distributions from the „raw“ [0,1[random numbers upon request from user code
GRandomT<GRandomFactory> y
GRandomT<GRandomFactory> z
User-code may always obtain fresh random numbers of different distributions, while creation of „raw“ random numbers happens centrally. The whole procedure is thread-safe and transparent to the user. The bulk of the work may happen at times when the computer is idle. Particularly optimization algorithms show idle times in regular intervals on the server, which can be utilized better in this way.

Figure 3: The “raw” material for random numbers of different distributions is produced centrally, while user-code sees a local random number generator.

evaluation times are long compared to the time needed for data transfers. The procedure also only makes sense for sufficiently large numbers of work items, as otherwise some compute units will consume more work items, and others might run empty. So if this facility *does* enter the Geneva production code, it will be as a purely optional feature.

4. The Hap library

According to Amdahl’s law, it makes sense to reduce server-side execution times as much as possible, as only this may ensure a sufficient speedup. Some algorithms require vast amounts of random numbers, and there are many objects interested in these. E.g. in an evolutionary algorithm with a population size of 1000 and 5000 parameters per individual, 5 million objects per iteration will require random numbers. Giving each of them their own random number generator is a nightmare, particularly as in networked execution, individuals (including their parameter objects) are sent to remote systems and may thus vanish entirely from the address space of the server, resulting in the constant creation and destruction of random number gernerators. Seeding is just one big issue of this scenario.
In order to avoid this, “raw” random numbers (evenly distributed floating point numbers in the range $[0,1[$) are produced in a central random number factory (compare figure 3). Objects interested in random numbers then employ a random number proxy that to them *looks* like a local generator. Random number proxies are also accessible through thread-local storage, so objects can avoid to store the proxies locally.

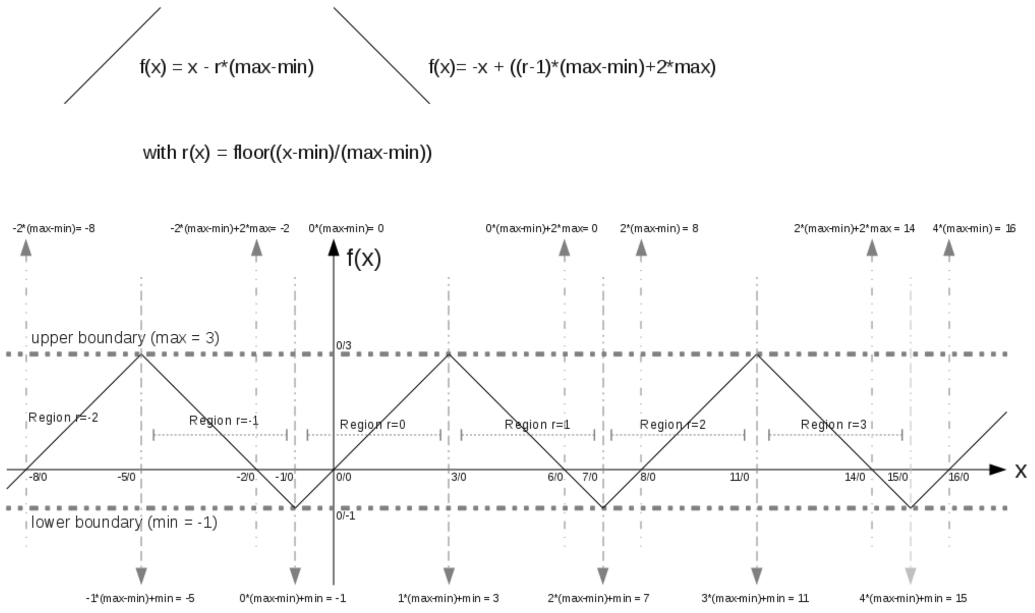

Figure 4: Constraints of individual floating point values are modelled as a mapping from an internal to a user-visible value. This allows to apply modifications of the core value to an unconstrained range, while presenting a constrained value to the user.

When a proxy is first instantiated, it contacts the random number factory and obtains a buffer with random numbers from it. Upon request from user-code, it may then produce random numbers of various predefined distributions from the raw material (e.g. gaussian-distributed numbers for evolutionary algorithms). Once the buffer has run empty, a new one is obtained from the factory. Upon destruction of the proxy, partially used containers are returned to the factory and are filled up with "fresh" numbers. They will then enter the cycle again.
Production of "raw" random numbers on the factory-side happens in multiple threads simultaneously. Fresh buffers are added to a thread-safe queue, until it is full. Threads will then block until there is again space in the queue. This has the big advantage that most random numbers will be produced while the host system is otherwise idle, which may particularly happen in the case of networked execution. This way the time needed for the production of "raw" random numbers does not affect the maximum speedup achievable by Geneva, as the numbers are produced while clients work on the work items. And issues with thousands of random number generators can be totally avoided.

5. Algorithm Efficiency

This section covers some of the ways in which the Geneva library tries to improve the efficiency of the optimization process, in the sense of reducing the number of iterations and the size of the populations, until a satisfactory optimum is found.

5.1 Constraint Handling

The size of the parameter space grows exponentially with the number of parameters. However, con-

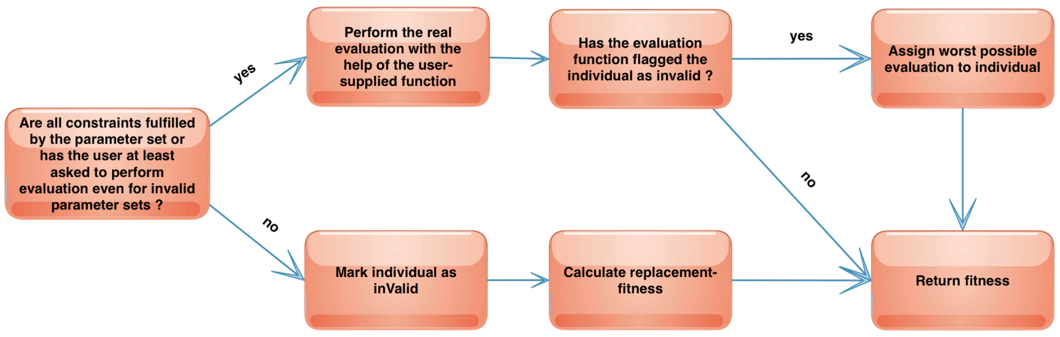
Are all constraints fulfilled by the parameter set or has the user at least asked to perform evaluation even for invalid parameter sets ?
yes
no
Perform the real evaluation with the help of the user-supplied function
Has the evaluation function flagged the individual as invalid ?
yes
no
Assign worst possible evaluation to individual
Mark individual as inValid
Calculate replacement-fitness
Return fitness

Figure 5: Evaluation workflow in the presence of potentially invalid solutions

straints may work into the opposite direction and reduce the size of the parameter space. Provided that optimization algorithms spend most or all of their time in "valid" areas of parameter space, the overall optimization procedure is sped up, as only few candidate solutions need to be rejected. Two types of constraints are considered in Geneva:

- Constraints of the allowed value range of single parameters. E.g., a floating point parameter might only be allowed to assume values in the range $[0,1]$. Geneva takes care of this situation by transforming an unbounded "inner" parameter value to a bounded "outer" parameter value. This way optimization algorithms do not need to be aware of the constraints of individual parmeters, which greatly simplifies the task of adding new algorithms. See figure 4 for an illustration of the procedure.

- Constraints may also involve two or more parameters. E.g., with two parameters x and y, each with an allowed value range $[0,C]$ (where C is a constant), a constraint of the form $x+y<C$ will render half of the parameter space invalid.

The latter constraint-type (henceforth called "inter-parameter constraint") is taken care of in the following way:

- First an "invalidity" I is calculated for a given parameter set. Values of I in the range $[0,1]$ indicate that the parameter set is in a valid area of the parameter space. Values $I>1$ indicate an invalid parameter set. The deviation from 1 gives an indication of "how invalid" the parameter set is. In the above example, $(x+y)/C$ could be used to specify the invalidity.

- Next, valid evaluations are transformed with a sigmoid function. For invalid parameter sets, depending on whether Geneva is used to maximize or minimize, the upper or lower boundary of the sigmoid function is multiplied with the invalidity.

- As a consequence, a joint quality surface is created for valid and invalid parameter sets. The quality surface drops towards valid areas of the parameter space (in the case of minimization) and will always have a worse value than valid solutions. Optimization algorithms are in this way "drawn" towards valid areas of the parameter space, where they will spend most time.

- As a direct consequence, evaluaton functions will not be called for invalid parameter sets, which might otherwise lead to bad results (crashes or incorrect evaluations).

The procedure outlined abobe is used successfully in a use case involving the optimization of exhaust systems with Geneva [5]. It is illustrated again in figure 5. Note that the step "perform real evaluation" will involve the transformation with the sigmoid function. Geneva does support other types of transformations, so figure 5 shows the general situation.

5.2 Meta-Optimization

Optimization algorithms will usually comprise a large number of configuration parameters. Choosing the right configuration parameters can be very difficult, and performing optimization with suboptimal settings may lead to increased numbers of evaluations until a satisfactory optimum is found. It is possible, however, to make the number of evaluations itself the figure of merit of an optimization run. The task is then to optimize the configuration parametrs of a given optimization algorithm in such a way that it converges as quickly as possible. Doing this can be difficult or impossible for very long running evaluation functions, as many optimization runs have to be performed, before an optimal set of configration parameters is found. However, when it is possible to define a mathematical function with similar complexity and geometry to the "real" evaluation function, one might be able to use configuration parameters found for this replacement function instead. This is not entirely satsifying, but likely still better than a free guess at the correct parameters.

5.3 Increasing the size of populations

Finally, in the beginning of an optimization run, it is usually very easy to find better solutions than those the algorithm started with. For algorithms that do not need a fixed population size, it is then often possible to start with a small population, which is then increased as it becomes more difficult to find better solutions. This procedure reduces the number of overall evaluations and thus speeds up the optimization run. Mapping a varying number of individuals to a constant number of compute ressources may be difficult particularly in networked mode, though. Populations of varying sizes are implemented e.g in Geneva's Evolutionar Algorithm implementation.

6. Summary

Geneva allows to perform parametric optimization using a selection of different optimization algorithms on devices ranging from many-core systems to clusters, Grid and Cloud. A plug-in architecture was created that allows to easily add new means of parallelization. Optimization is de-coupled almost entirely from parallelization, which simplifies the code and allows to make the Geneva libraries more efficient and stable. Further measures help to reduce the overall number of evaluations, making Geneva suitable for the deployment in particularly complex optimization problems. The authors welcome suggestions for further improvements and new deployment scenarios beyond the current use-cases from the automotive industry and particle physics.

7. Thanks

The authors would like to thank the Helmholtz Society of German Research Centres as well as Steinbuch Centre for Computing at Karlsruhe Institute of Technology for their support of our work. Likewise, we would like to thank Tenneco for the opportunity to work on exciting optimization tasks in the automotive industry.

References

[1] Code of the Geneva library collection: `http://launchpad.net/geneva`

[2] The boost library collection: `http://www.boost.org`

[3] Dr. Rüdiger Berlich, Dr. Ariel García, Dr Sven Gabriel: *Parametric Optimization with the Geneva Library Collection*; Manual of the Geneva library collection; Available from `http://www.gemfony.eu/fileadmin/documentation/geneva-manual.pdf`; The file is constantly being updated. Last viewed April 19, 2015.

[4] Sam Harnew, Jonas Rademacker: *Model independent determination of the CKM phase γ using input from $D^0 - \bar{D}^0$ mixing*; Journal of High Energy Physics 2015:169; Springer Berlin Heidelberg; March 2015; Available under an Open Access policy from `http://link.springer.com/article/10.1007\%2FJHEP03(2015)169`

[5] Dominik Rödel, Dr. Rüdiger Berlich: *Lösungsansätze in der parametrischen Optimierung der Akustik von Abgasanlagen*; Conference proceeedings of "2. Internationaler Motorenkongress"; Baden Baden / Germany; February 2015

[6] Mathias Michel et.al.: *ComPWA: A common amplitude analysis framework for PANDA*; 2014 J. Phys.: Conf. Ser. 513 022025

[7] Dr. Rüdiger Berlich, Dr. Marcel Kunze, Dr. Ariel García, Dr Sven Gabriel: *Distributed Parametric Optimization with the Geneva Library*. In *Data Driven e-Science*; Conference proceedings of ISGC 2010; Springer; Simon C. Lin and Eric Yen (editors); ISBN 978-1-4419-8013-7; p.303 ff.

[8] Dr. Rüdiger Berlich: Application of Evolutionary Strategies to Automated Parametric Optimization Studies in Physics Research; PhD thesis, Institut für Experimentalphysik I der Ruhr-Universität Bochum; 2004

[9] A short description of Amdahl's law: `http://en.wikipedia.org/wiki/Amdahl's_law`