PROCEEDINGS OF SCIENCE

Parallelization of events generation for data analysis techniques

Alfio Lazzaro*
CERN openlab, Geneve
E-mail: alfio.lazzaro@cern.ch

With the startup of the LHC experiments at CERN, the involved community is now focusing on the analysis of the collected data. The complexity of the data analyses will be a key factor for finding eventual new phenomena. For such a reason many data analysis tools have been developed in the last several years, which implement several data analysis techniques. Goal of these techniques is the possibility of discriminating events of interest and measuring parameters on a given input sample of events, which are themselves defined by several variables. Also particularly important is the possibility of repeating the determination of the parameters by applying the procedure on several simulated samples, which are generated using Monte Carlo techniques and the knowledge of the probability density functions of the input variables. This procedure achieves a better estimation of the results. Depending on the number of variables, complexity of their probability density functions, number of events, and number of sample to generate, the whole procedure can be high CPU-time consuming. In this paper we show how the Monte Carlo generation of the events for each simulated sample can be parallelized using OpenMP to scale over multi-cores in a single computational node.

13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research
February 22-27, 2010
Jaipur, India

*Speaker.

PoS(ACAT2010)070

1. Introduction

In the last years many complex techniques have been used in the High Energy Physics (HEP) community, such as maximum likelihood, neural networks, and boosted decision trees [1, 2]. These techniques are used to analyze the huge amount of data collected by the experiments. Data are a collection of independent *events*, an event being the measurement of a set of *variables* (energies, masses, spatial and angular variables...) recorded in a brief span of time by the physics detectors. The events can be classified in different *species*, which are generally denoted with *signals*, for the events of interest for physics phenomena, and *backgrounds*, for the rest. The discrimination between the different species is obtained using particular variables (*discriminating variables*), or more in general combination of these variables, which have different characteristics for signal and background events. These techniques have several advantages with respect to the simple *cut and count* analysis method [1] as better discrimination between signals and backgrounds, the possibility to take in account errors with a better precision and correlations between the discriminating variables used in the analysis. However, they require long CPU-time execution. An important role in the determination of the results is played by the possibility of repeating the evaluation of the parameters on several distinct simulated samples of data, doing the so called *pseudo-experiments*. The samples are generated using Monte Carlo techniques and the knowledge of the probability density functions (PDFs) of the input variables [4]. This procedure achieves a better estimation of the results. Also in this case it can require long CPU-time execution, depending on:

- number of events and variables to generate;
- number of data samples to generate (usually on the order of thousands);
- complexity of the models.

Pseudo-experiments are used in statistical techniques based on frequentist inference, such as the determinations of confidence intervals and significance tests of the parameters. In this case the number of pseudo-experiments can be greatly high (5σ statistical significance test requires more than one million of pseudo-experiments for an accurate estimation). Assuming this, we can conclude that it is particularly important to speed-up the generation of the simulated samples. An easy solution is to parallelize the events generations, taking benefit from the new multi-core CPUs. In the last years, vendors like Intel and AMD have not incremented the performance of single computational unit as in the past, but they are working on multi-core CPU. Currently we have up to 12 cores implemented on one single chip. This fact represents a possible revolution in the development of new programs. Indeed we can parallelize the code using a shared memory paradigm obtaining great benefits from new multi-core architectures. So we have to reformulate some algorithms generally used for HEP data analyses. These techniques of High Performance Computing are well established in other fields, like computational chemistry and astrophysics. In HEP community there is not such a large use, but in the future it can be an elegant solution in all the cases where the data analyses will get more and more complicated.

An implementation for the parallelization of the maximum likelihood procedure is described in the Ref. [3]. In the work described in this paper we focus on the parallelization of events generation

[1] Set of independent cuts on the input variables.

for pseudo-experiments. The parallelization is based on OpenMP, therefore using a shared memory paradigm. We will describe the implementation and we will show the results of scalability for an application used as benchmark.

2. Parallelization of Monte Carlo events generation

The Monte Carlo method is a numerical technique for calculating probabilities and related quantities by using sequences of random numbers [1]. The key factor is the generation of random numbers, or more properly *pseudo-random* numbers, using specific algorithms which are implemented in pseudo-random number generators (PRNGs) [4]. These generators produce long sequences of apparently random results (*streams*), which are in fact completely determined by a initial value, known as *seed*. Another important characteristic of the PRNGs is their periodicity: the maximum length of the sequence before they begin to repeat, so broken the randomness of the sequences. Generally PRNGs have long period (usual $> 10^{10}$), which allows the possibility to use them in most Monte Carlo applications without particular worries.

It is possible to generate numbers which follow simple PDFs, such as Gaussian and uniform distributions. There are a couple of methods to generate numbers from any generic PDF [4]. These methods involve transforming an uniform random number in some way. The most used in HEP is the *accept-reject* method. Given a generic PDF f, it involves several steps:

1. extraction of a random number x with a uniform distribution (or d random numbers in case of a PDF of d dimension) in the variable range of validity;
2. extraction of another random number y with uniform distribution in the range between 0 and the maximum value of f;
3. testing whether $f(x)$ is greater than the y value. If it is, the x value is accepted. Otherwise, the x value is rejected and the algorithm tries again.

Clearly the number of extractions from the uniform distribution needed to generate a valid random number from f is not predictable a priori. Highly-non-uniform multidimensional distributions can require several random number extraction before accepting a value. Instead, the minimum number of extractions is given by the dimension of the PDFs plus the extraction of y.

The usual adopted procedure in HEP for the generation of different samples for the pseudo-experiments is based on extracting different streams from the same PRNG, each stream with a different seed for each pseudo-experiment. The values of the seeds are saved for allowing the regeneration of the same events of a specific pseudo-experiment, independently from the other pseudo-experiments. Note that we are extracting different streams which are not guaranteed to be independent. The hope is that they will be non-overlapping and uncorrelated streams of the original PRNG. This is generally valid when there is a proper determination of the seed values in case of PRNG with large periodicity [5]. This hope, however, has no theoretical foundation. Consequence of that is the impossibility to be sure whether a PRNG is affected by correlations [6]. This procedure of events generation allows the possibility for parallelizing on the pseudo-experiments as entire entities, dividing them in different processes. The clear limitation is when we want to have a finer parallelization for the generations inside each pseudo-experiment. For such a case we require

appropriate parallel pseudo-random generators [7]. These generators provide methods for extracting *sub-streams* from a main stream (with an unique seed). Therefore the different sub-streams can be used for the parallel generation inside each pseudo-experiment. For the implementation described in this paper we use a PRNG provided by Tina's Random Number Generator library (TRNG) [8]. TRNG is a state-of-the-art C++ pseudo-random number generator library for sequential and parallel Monte Carlo simulations. It provides optimized PRNGs and methods for usage in parallel applications. We choose the `yarn5` PRNG which has a period of about 4.57×10^{46}. This PRNG implements a method for *block splitting*. Let m_i be the maximum number of calls to a PRNG by each processor i and let p be the number of processes ($i = 1, ..., p$), we can split the main stream of random number so that each process i will get a sub-stream with m_i random numbers. This method works only if we know m_i in advance or can at least safely estimate his value. To apply block splitting it is necessary to jump from the nth random number to the $(n + \sum_{j=1}^{i-1} m_j)$th number for the process i without calculating all the numbers in between. A method `jump` is provided in the `yarn5` PRNG for doing that.

2.1 Implementation of the algorithm

In this section we describe the implementation of the algorithm for events generation in case of pseudo-experiments. The implementation is based on the `yarn5` PRNG, using the `jump` method, coded in C++ language and OpenMP.

At this point a crucial consideration must be made: the generated sample must not depend on the number of processes. This applies either for the values and the order of the generated events. It is mandatory for debugging, especially in parallel environments where the number of parallel processes varies from run to run, but also guarantees that the quality of a PRNG with respect to an application does not depend on the degree of parallelization.

Let's consider the simple case of a generation of a single variable with an uniform distribution. In this case we do not need the accept-reject method, but we can directly extract from the corresponding distribution. Assuming that we want generate a sample with N events and we have p processes, we can easily calculate the number of events n_i to generate by each process: $n_i = (N \text{ DIV } p)$ plus 1 if $i \leq (N \text{ \% } p)$ [2]. In this case we have $m_i = n_i$ (i. e., one to one correspondence between calls to the PRNG and events), so that the parallelization of the generation is straightforward: the process i does the generation of n_i events, which are stored in a local (to the process) data structure (such as a C++ `std::vector`); then, at the end of the parallel generation, each process copies his local generated data in the final global data structure. This algorithm satisfies the condition of independence from the number of processes and it can be easily implemented using OpenMP.

Now we consider the case of generation of variables using accept-reject method from a single complex PDF of d dimension. As we said above, in this case we cannot predict a priori the value of m_i. However, we can calculate the lower limit on this value: $m_i \geq (d+1) \times n_i$. If we consider $(d+1) \times n_i$ as the maximum number of random numbers available for the process i, we end up with g_i events generated by each process and stored in the local data structures. If we consider that all events are accepted at the first extraction of the corresponding random numbers, then $g_i = n_i$. Since this is normally not valid, we can conclude that $g_i < n_i$. Giving the characteristic of the

[2] DIV is the integer division and % is the module of the integer division as in the C++ definition.

`jump` method to split the main stream in continuous sub-streams for each process, we have that the (g_i+1)th event will be generated by process $i+1$ (and so on for the remaining events). This means that considering $m_i = (d+1) \times n_i$, we will generate the first $\sum_{i=1}^{p} g_i$ events of the global final sample (copying together all local data structures in the global data structure, following the order rank i of the processes). Then we can iterate the procedure of parallel generation considering now the remaining $N - \sum_{i=1}^{p} g_i$ events for the generation. This requires corresponding jumps in the PRNG for each process, which introduces some overheads in the parallelization (the complexity of jump grows logarithmically in its argument). Other overheads are introduced by the handling of the data containers. To reduce these effects, the parallel generation is stopped when the remaining number of events to generate is less that 1% of total request events, with the remaining events generated by only one process up to the completion of the global sample. Also this procedure guarantees that this sample will be the same independently by the number of parallel processes.

The last case we consider is the generation of events with variables from different PDFs, which is the usual case in the pseudo-experiments. The solution adopted in this case is the generation from each single PDF, one at a time, which basically leads to the above discussed cases. This is the only procedure we found for guaranteeing that the generated samples will not depend by the number of parallel processes. The side effect is an further overhead when merging all the values of the variables in the final dataset.

3. Benchmark example

As tests of the implemented parallel algorithm, we want to generate a data sample from the following PDFs (in parenthesis we report the d dimension of the PDFs):

- uniform $(d = 1)$;
- truncated normal distribution $(d = 1)$;
- 2^{nd} order polynomial $(d = 1)$;
- neutrino oscillation distribution [9]:

$$P(\nu_\mu \to \nu_e) = \sin^2(2\theta)\sin^2\left(\frac{1.27\Delta m^2 L}{E}\right), \tag{3.1}$$

  where P is the probability for a ν_μ to transform into a ν_e , L is the distance in km between the creation of the neutrino from meson decay and its interaction in the detector, E is the neutrino energy in GeV, and Δm^2 and θ are two parameters which characterize the oscillation. In this formula the two variables to generate are L and E $(d = 2)$.

So in total our sample is composed by 5 variables. Note that for the last two PDFs the generation is done using the accept-reject method. Plots of the PDFs used in our tests and an example of corresponding generated data distributions are shown in figures 1 and 2 .

We ran the tests on an Intel Westmere-EP server which is available as test machine at CERN/Openlab. It is a dual-socket machine, where each CPU is an Intel Westmere-EP X5670, with 6 cores (2 hardware threads per core) running at 2.93 GHz (so a total of 12 cores and 24

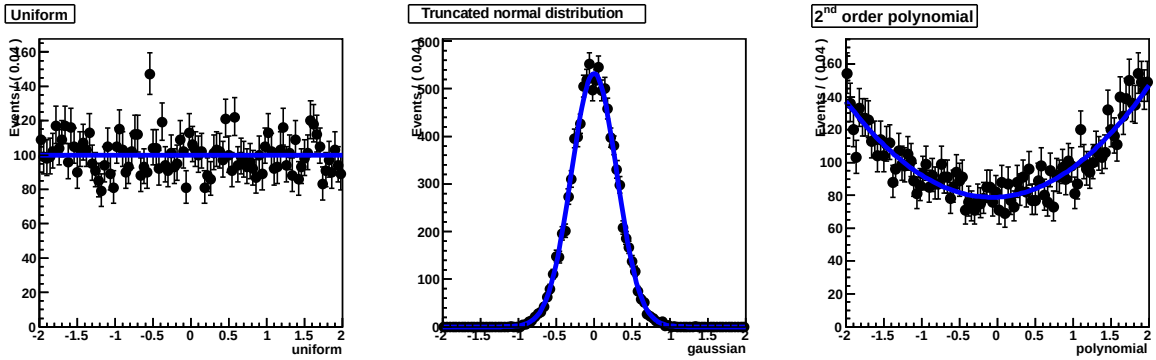

Figure 1: Plots of the unidimensional PDFs used in the benchmark test (blue line). Points with errors are an example of generated data distributions (10,000 events). Note that the curves are just rescaled and superimposed to the points.

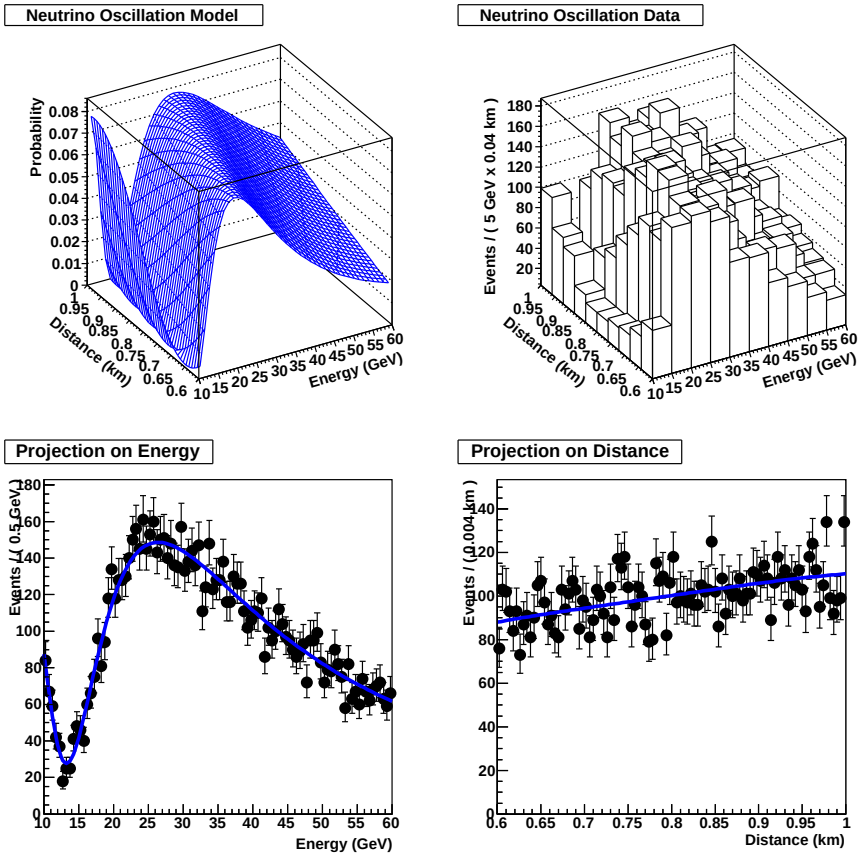

Figure 2: Plot of the neutrino oscillation model used in the benchmark test (top left plot). An example of generated data distribution (10,000 events) is shown in the 2D histogram (top right plot). In the two bottom plots we show the projections on the two corresponding generated variables. Note that the curves are just rescaled and superimposed to the points.

hardware threads). The Turbo Mode of the CPU is switched off. The processes were pinned to the cores running them. The system allows hardware threading and it was used during the tests. Thus, if there were no more physical cores available, the jobs would be pinned to hardware threads, still maximizing the amount of physical cores used. The system is running Scientific Linux CERN 5.4 (SLC5), based on Red Hat Enterprise Linux 5 (Server). We compile the code with GCC 4.1.2 in 64 bit, the standard compiler available with SLC5.

We do the average of the runtime (wall-clock time) by running 10 times each test. The wall-clock time spent by the generation of a single sample (i. e. one pseudo-experiment) with 10,000 events is 30.4 seconds when running in sequential. This time increases linearly with the number of events and with the number of pseudo-experiments. Note that in this work we are focusing on the parallelization of a single pseudo-experiment, but we should consider that usually the number of pseudo-experiments is in the order of thousands for a data analysis application, which means a significant amount of total time for executing the application.

We consider two cases in our tests:

- **Weak scaling tests**. These tests focus on the scalability, which is defined as throughput. We increase the number of events to generate proportionally to the number of processes involved in the parallelization. In an ideal case, as more processes with more work are added, one would expect the throughput to grow proportionally to the added resources. We take as reference the generation of a sample with 10,000 events. Results of the efficiency of the parallelization are shown in figure 3. The efficiency is defined as the scaling of the software relative to the sequential runtime, confronted with ideal scaling determined by the core count. In cases where multiple hardware threads are being used, perfect scaling is defined by the maximum core count of the system (12). From the plot we observe a good scaling up to 6 processes (99.0% with 6 processes). The small decrease in the efficiency is consistent with the overhead introduced by the parallelization. Over 6 processes we observe a drop in the efficiency (90.9% with 12 processes), which is not attributable to the overhead of the parallelization. We think that the reason for this drop is the data access when the application is running on the 2 CPUs (note that 6 is the number of cores per CPU). The efficiency curve surpasses 100%, since for thread counts higher than 12, expected scalability is fixed to 12x. Thus a final value of 108.3% indicates that the system loaded with 24 threads of the benchmark yields 8.3% more throughput than a perfectly scaled serial version on 12 physical cores. One should note that this extra 8.3% of performance is traded in for a penalty in memory usage, as the number of software processes is double the one in the case of 12 cores.

- **Strong scaling tests**: In this case we are interested to see the speed-up of the application for a fixed number of generated events. For this reason we do not use the hardware threading. Results are shown in figure 4. We observe a excellent scaling. The small penalty is due two factors: the overhead introduced by the parallelization, and the non-parallelizable part of the application (mainly the data handling), which represents less than 0.5% of the total sequential execution time.

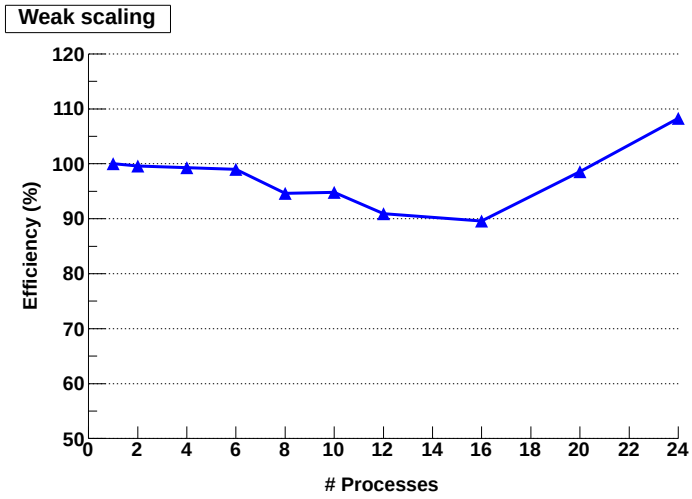

Figure 3: Efficiency in case of weak scaling test (see text for details).

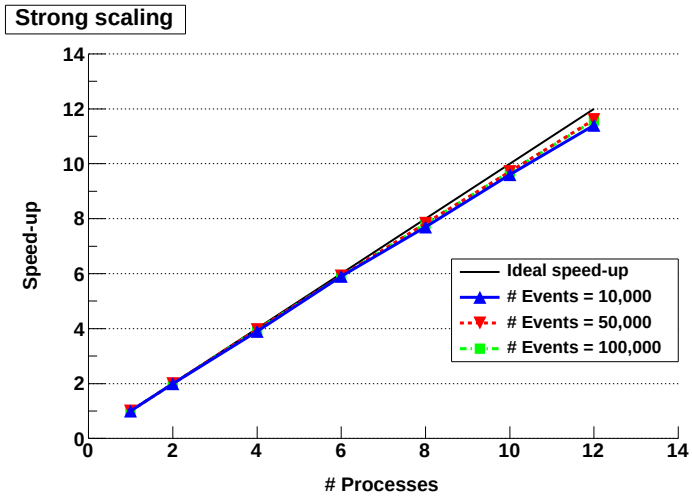

Figure 4: Speed-up in case of strong scaling test (see text for details). We perform 3 tests with 10,000, 50,000, and 100,000 generated events. Black line represents the ideal speed-up, which corresponds to the number of processes used in the parallelization.

4. Conclusion

The algorithm adopted for parallelization gives good results for the tests performed. It satisfies the requisite to obtain the same generated events (same values in the same order) independently by the number of parallel processes. The implementation will be made available inside the RooFit package (as part of ROOT framework) [10] in the new releases, providing a general interface for the parallel generation of sample for pseudo-experiments.

Acknowledgments

The work has been performed under the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures. We would like to thank W. Verkerke at NIKHEF for the collaboration, D. Verhoeven and C. Schrijvers at SARA and HPC-EUROPA2 people for organizing the visit at NIKHEF, and G. Cerizza, V. Innocente, L. Moneta, A. Nowak, S. Jarp at CERN for the fruitful suggestions for doing the parallel implementation.

References

[1] G. Cowan, *Statistical Data Analysis*, Clarendon Press (1998).

[2] J. Friedman, T. Hastie, R. Tibshirani, *The Elements of Statistical Learning*, Springer (2001).

[3] A. Lazzaro and L. Moneta, *MINUIT package parallelization and applications using the RooFit package*, *J. Phys.: Conf. Ser.* **219** 042044 (2010).

[4] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, Third Edition, Cambridge University Press (2007).

[5] M. Matsumoto and T. Nishimura, *Monte Carlo and Quasi-Monte Carlo Methods 1998*, Springer (2000), pp. 56–69.

[6] P. Hellekalek, *Don't trust parallel Monte Carlo!*, proceeding of Parallel and Distributed Simulation (1998), pp. 82–89.

[7] H. Bauke and S. Mertens, *Random numbers for large-scale distributed Monte Carlo simulations*, *Phys. Rev. E* **75**, 6, 066701 (2007).

[8] See the Tina's Random Number Generator Library webpage: `URL:/http://trng.berlios.de/`.

[9] G. J. Feldman and R. D. Cousins, *Unified approach to the classical statistical analysis of small signals*, *Phys. Rev. D* **57**, 7, 3873 (1998).

[10] See the web page of the ROOT project: `URL:http://root.cern.ch/`.