MNRAS **000**, 1–4 (2020) Preprint 6 November 2020

On the parallelization of the VEGAS algorithm.

A. F. Gómez-Muñoz,[1]★
[1]*Instituto de Física, Facultad de Ciencias Exáctas y Naturales, Universidad de Antioquia (UdeA), Medellín, Colombia*

6 November 2020

ABSTRACT
Multidimensional integration is one of the most useful applications of Montecarlo methodologies in science, and therefore requires the attention towards the development of efficient algorithms. In this paper we first discuss how an integration algorithm could beneficially impact its performance and accuracy from its parallelizable nature as well as from its adaptive behaviour even when dimensions are increased. With this in mind we expose VEGAS algorithm thanks to its optimal performance in high dimensions and propose a methodology for its parallelization based in the results of a gaussian integration. From that an order O(N) algorithm with an efficient data communication and its principal bottleneck parallelized is presented. We then conclude that for a common integration problem a speedup of 70-3000 can be obtained, that the asynchronous work have a high impact in the results, and that an explicit uniform distribution of work between processors is not always the most efficient option to parallelize an algorithm.

Key words: VEGAS algorithm – parallelized Montecarlo – adaptative integration methods – Montecarlo integration – importance sampling.

1 INTRODUCTION

Montecarlo methods are a broad class of computational algorithms that are based in random sampling to obtain numerical results, which in principle permit to solve any problem with a probabilistic interpretation. Therefore, this methodologies are quite general and often used in science to solve optimization, numerical integration and sampling problems. Some examples of the application of Montecarlo methods are found in: the calculation of Quantum Mechanical observables, evolution of statistical physical systems, Galactic Evolution, business risk analysis, Artificial Intelligence applied to games, estimate the impact of pollution, and others Roese (2014); Weinzierl (2000).

In some applications as numerical integration the results are obtained from an estimate of the mean value of some distribution, and its error is directly obtained from the estimate deviation of such mean value. And although within this framework simple Montecarlo algorithms usually perform better than standard integration ones in high dimensions, they still have lots of problems. For example, crude montecarlo will pass most of its time sampling regions where there are not relevant points for the calculation of the integral. Additionally, the fact that it samples equal number of points in the most variable regions of the function as in others, also motivates a more variable error in the result. Therefore the question of what methodology should we use to efficiently sample important points to integrate such that the deviation is partly minimized naturally arises.

This question is precisely the one to which we would like to contribute in this article. For that, we first show how the VEGAS algorithm Lepage (1978) takes advantages of the variance reduction that can be obtained from stratified sampling, as at the same time it adapts to an ideal sampling distribution using the basic principle of importance sampling. Once the general remarks of the algorithm are clear we explicitly describe it.

Then, we propose a methodology to parallelize the algorithm. In short: all the routines of the algorithm are constructed to be of order O(N), so that from a given size we would have $t_{run} \propto 1/N_{processes}$. Afterwards, a serial version of the program is analyzed to measure the proportion of time that takes each bottleneck; from that the more important parallelizable parts became clear and limits to the possible speedups are obtained. Finally, we analyze the data communication and propose from it a suitable structure for the code as well as the possibility of asynchronous tasks to reduce parallel overhead and distribute the work more efficiently.

With this ideas in mind a parallelized version of the algorithm is constructed. Then, its results integrating a gaussian function are compared with the serial ones and analyzed. From that, it is then concluded that for a common integration problem a speedup of 70-3000 times was obtained, that the asynchronous work had a high impact in the speedup, and that an explicit uniform distribution of work between processors was not the most efficient solution in this case; despite that partly Blaise (2020) recommended it and implement it in its examples.

★ E-mail: andres.gomez27@udea.edu.co (UdeA)

2 VEGAS ALGORITHM

This algorithm initially developed by G. Peter Lepage Lepage (1978) has been widely used for decades to evaluate integrals of 2 or more dimensions numerically, having therefore natural adoption by libraries as the GNU Scientific Library (GSL) and packages in high level languages as python. What makes this algorithm so useful in high dimensions is the capability that it has to adjust to an optimal distribution for the integration. As the routine proceeds the sample probability p, defined as a step function by N successive intervals Δx_i to which each x belong uniquely, begins to tend to the distribution indicated in 1, and hence the estimates for the integral $E(I)$ and its variance $S^2(I)$ given by equations 2 begin to converge more rapidly, as can be readily observed by a direct replacement; in fact complete convergence to the limit probability leads to 0 error (note that I denotes here the value to which the estimator refers). Furthermore, the majority of the process involved in the algorithm can be done simultaneously and can be considered simple, making VEGAS highly paralelizable; as we will see later a theoretical speedup of the order of thousands can be obtained.

$$p(x_i) = \frac{1}{N\Delta x_i} \rightarrow \frac{f(x)}{I} \tag{1}$$

$$\begin{aligned} E[I] &= \sum_{i=0}^{N} \frac{f(x_i)}{p(x_i)} \\ S^2[I] &= \frac{1}{(N-1)N} \sum_{i=0}^{N} f(x_i)^2 - E[I]^2 \end{aligned} \tag{2}$$

In addition to this, the algorithm intends to choose the Δx_i so that each interval contributes equally to the final integral result. This is done by subdividing each Δx_i in m_i subintervals given by equation 3 and then defining each new interval Δx_i by grouping equal number of successive subintervals from left to right for every new interval. In this way is clear that K in 3 accounts for the maximum divisions to be made in a given interval and so for the precision in calculating each Δx_i. All this also implies that when N is sufficiently high it will decrease the total deviation, as stratified sampling, making each interval to have the same standard deviation.

$$m_i = K \frac{\sum_{x_i \in \Delta x_i} \frac{f(x_i)}{p(x_i)}}{E[I]} \rightarrow K \frac{\int_{\Delta x_i} f(x)dx}{\int f(x)dx} \tag{3}$$

With this considerations in mind the following general procedure for the algorithm is considered:

1. Sample M points from the piecewise distribution p.
2. Estimate (I, S^2) and evaluate possible error convergence to end program and output estimations.
3. Calculate the mi's to subdivide intervals, group them again and define the probability p.
4. Evaluate convergence $m_i = m_j$ $[i, j = 0, 1, \ldots, N]$ to end program and give estimations.
5. Repeat from 1 or until the maximal number of tries defined.

It is important to note that in from two dimensions the probability distribution is supposed to be separable, and therefore the same process for each $p_j(x)$ in $p(x) = p_1(x)p_2(x)...p_{N_{dim}}(x)$ is made.

3 PARALLELIZATION METHODOLOGY

Amdahl's Law set the maximum speedup that could be obtained in a code given that a portion P of the program can be parallelized. This is very important because programs in which only a few percent of the work can be done simultaneously would not actually improve considerably and all the effort in the parallelization process can became meaningless. Even programs in which half of the work can be done simultaneously can only increase at most by 2 times in velocity. On the other hand, programs in which 80.0% - 99.0% of work can be parallelized can have more important speedups of the order of 5-100. Hence, as the majority of the processes that we propose in the algorithm can be parallelized in each step an analysis of them is rather promising. In addition to this, its simplicity make it highly suitable to be a scalable parallel code.

With this considerations in mind and the algorithm exposed, a serial version of it was done, writing for the mapping of p a rejection method. A fundamental consideration in the construction of the program was to write the algorithm such that it is at most of order O(N). In this way it was possible to make the program parallel scalable so that from given size $t_{run} \propto \frac{1}{N_{processors}}$ or equivalently in this case, the amount of work (from a given size) became proportional to the run time.

Now, to identify the limitations in the speedup and the parts that could be parallelized in the code, all the bottlenecks which coincide to be parallelized were analyzed for the simple integration of a gaussian. The results of measuring the proportion of time that each process takes are shown in tables 1 and 2 for two different machines. This measurements were made from 10000 independent runs of the serial program, each of 10000 sample points.

Process name	time proportion
Rejection	92.7 %
Estimates	0.1 %
Adapt p	4.5 %
Convergence	0.1 %
Total parallelizable	97.4%

Table 1: Results in personal machine with 8 CPU's. Adapt p basically refers to the third step of the algorithm in section 2.

Process name	time proportion
Rejection	99.1 %
Estimates	0.01 %
Adapt p	0.6 %
Convergence	0.01 %
Total parallelizable	99.7 %

Table 2: Results in the university cluster with 24 CPU's. Adapt p basically refers to the third step of the algorithm in section 2.

Under further analysis of the algorithm constructed, it was decided to program it in a class, this permitted an additional time reduction of 1-2% because non mutable objects are needed to be created in each process; using with this, in a programming sense, a shared memory model. Also, this let us with a more simple conceptual framework to program.

In addition, the fact that some tasks are not only paralelizable, but that they do not depend of any kind of expensive input to be read permit us to perform them asynchronously, as in the case of the rejection for example. The two cases of asynchronously and synchronously work were tested, the communication implied between processors in such cases shown to have important consequences.

4 SERIAL AND PARALLEL COMPARISON

Two parallelizations were implemented. One that consists in paralelizing the rejection process of the algorithm, and other that consists in running the same program in different nodes simultaneously.

4.1 Paralelization of Rejection Part

In this part the analysis made is rather promising. For the personal machine a maximum speedup of 6.6 can be achieved, while for the cluster a maximum of 24 can be achieved; if processors could be increased to the order of thousands the algorithm speedups would be 33 and 1000 for each case.

Now, parallelizing with the multiprocessing library in python and using equal explicit distribution of work for each processor it was found that the algorithm did not run faster, in fact it was approximately 2 % slower, as hypothesis we supposed that this had to do with the communication between processors. Consequently, we rewrite the algorithm for the rejection to be done asynchronously, sampling portions of approximately 25% of $N_{sample\_points}/N_{processors}$. With this a speedup of $\approx$ 2 was obtained for the personal machine while a speedup of $\approx$ 8 for the cluster; these are approximately 30 % of the maximum speedups possible, respectively.

4.2 Same Program Several Thread Processes

Basically several nodes run the same or a slightly modified version of the program simultaneously. Now, in order to measure how much time the program would take as the sample size and number threads are increase proportionally we measured it for different sample sizes based in independent runs. Using then bootstraping for the time estimates of each size the result shown in figure 1 were obtained. From the graph it is seen that approximately from a sample size of 10000 the time of run is in fact proportional to the sample size.

Summing up, this two methodologies used together can therefore be used to increase the speedup algorithm from $2N_{sample\_size}/100000$ to $8N_{sample\_size}/100000$ in each machine respectively.

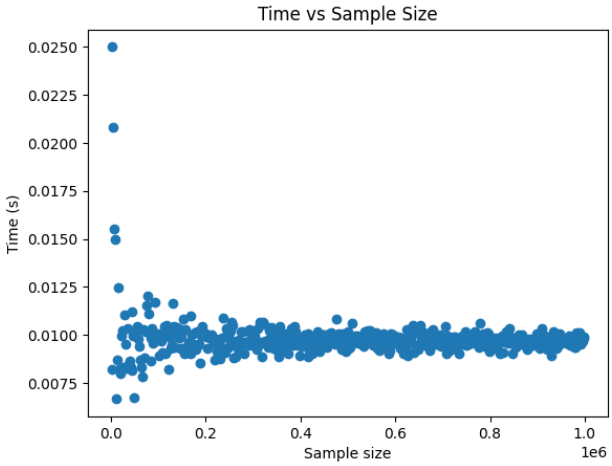

Figure 1: The graph illustrates the times measured for the integration of a simple gaussian as the sample size and the number of processors increase proportionally, each point is the results of 100 runs in a bootstraping sampling.

5 CONCLUSIONS

We were able to parallelize the initial serial code up to approximately 30 % percent of the theoretical limit speedup; only considering the rejection process.

The paralelizations in section 4 together show remarkable results, for example a 1 million sample point integration in 1 dimension with 10 threads and 24 processors have a speedup of 70; a similar reasoning for a $50x10^6$ sample points integration leads to a speedup of $\approx$ 3000.

A first good approximation that set the bases for a good parallelized VEGAS algorithm was made, in fact if we were not constrained by the number of processors, speedups of 4 times faster would be obtained.

It was found that a uniform explicit distribution of work for each CPU does not necessarily gives the best results, and although this could be for a variety of reasons, in the presence work was found that these explicit order increased parallel overhead in the rejection process at such degree that it was actually faster the serial version. Nevertheless, when chunks of $\approx \frac{N_{samples}}{4N_{processors}}$ were sample asynchronously the parallel overhead greatly decrease and the parallelization was actually successful as exposed.

6 PERSPECTIVES

Write a part of the program to find the optimal chunk size of data for the processors to sample with the rejection method asynchronously. A more explicit calculation with the processor instead of a pool could also be tried to be implemented to make a better analysis; principally in communication matters.

Compare the Nth dimensional integration results obtained in the base paper Lepage (1978) with the ones obtained by our program. Additionally, study the behaviour with peak type and

other complex functions.

Implement the strong convergence methodology exposed in the last work to talk in probabilistic terms (confidence intervals) of the results obtained.

Implement a general class with different Nth Montecarlo integration methods (uniform, stratified, VEGAS, and others) to compare the results obtained between them.

ACKNOWLEDGEMENTS

I thank my teacher Jose David Luis, because his classes and recommended readings permit me to propose an organize methodology (that can also be use in other cases) to carry out the paralelization of an algorithm that in principle was complicated for me to program.

REFERENCES

Blaise B., 2020, Lawrence Livermore National Laboratory Page
Lepage G. P., 1978, Journal of Computational Physics, 27, 192
Roese D. P. e. a., 2014, WIREs Comput Stat., 6, 386–392
Weinzierl S., 2000, Introduction to Monte Carlo methods