Reproducible Roulette Wheel Sampling for Message Passing Environments

Balazs Nemeth[1], Tom Haber[1,2], Jori Liesenborgs[1], and Wim Lamotte[1]

[1] Expertise Centre for Digital Media, Wetenschapspark 2, 3590 Diepenbeek, Belgium
{balazs.nemeth,tom.haber,jori.liesenborgs,wim.lamotte}@uhasselt.be
[2] Exascience Lab, Imec, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract. Roulette Wheel Sampling, sometimes referred to as Fitness Proportionate Selection, is a method to sample from a set of objects each with an associated weight. This paper introduces a distributed version of the method designed for message passing environments. Theoretical bounds are derived to show that the presented method has better scalability than naive approaches. This is verified empirically on a test cluster, where improved speedup is measured. In all tested configurations, the presented method performs better than naive approaches. Through a renumbering step, communication volume is minimized. This step also ensures reproducibility regardless of the underlying architecture.

Keywords: Genetic Algorithms, Roulette Wheel Selection, Sequential Monte Carlo, HPC, Message Passing

1 Introduction

Given a set of n objects with associated weights w_i, the goal of Roulette Wheel Sampling (RWS) is to sample objects where the probability of each object j is given by a normalized weight, $\tilde{w_j} = w_j / \sum_i^n w_i$. In genetic algorithms, objects are individuals and their weight is determined by its fitness [4]. After individuals have been selected for survival, they are either mutated or recombined to form the next generation. RWS is used in the resampling step of Sequential Monte Carlo methods [7, 1], where objects are weighted particles. Hereafter, this paper refers to objects in general.

The resampling step is commonly implemented in one of two ways. The first approach, referred to as the cumulative sum approach, is to generate $u \sim \mathcal{U}(0, 1)$, and to select the last j for which $u \leq \sum_{i=0}^{j} \tilde{w_i}$. Computing the cumulative sum takes $\mathcal{O}(n)$ time and finding an object takes $\mathcal{O}(\log n)$. The second approach is the alias method [10]. Constructing an alias table takes $\mathcal{O}(n)$ time and taking a sample takes $\mathcal{O}(1)$ time. This results in a lower execution time, but, as Section 2 details, the cumulative approach is a better fit for parallelization.

This paper relies on parallel random generation techniques [8]. Since RWS is typically executed multiple times, each object is provided with a unique random generator from which a random number sequence can be generated in parallel.

However, if such techniques are not available, any pseudo random number generator (RNG) that can either jump in its sequence or a pre-generated sequence can be used instead.

Reproducibility is a desirable property of any scientific computing code. For this reason, only methods that output the same samples are considered. This means that the results are reproducible not only for a given parallel configuration if executed repeatedly, but also if the number of processors, p, is changed.

The remainder of this paper is structured as follows. Section 2 describes how to parallelize RWS in a reproducible fashion. Experimental results are shown in Section 3. Section 4 lists related work. Section 5 concludes the paper and proposes future work.

2 Reproducible RWS

Given a sequence of weights, $(w_1, \ldots, w_n)$, the output of RWS is a sequence $S_1 = (s_1, \ldots, s_n)$ where s_i is the index of the object that has been selected. Let $S_2 = (s'_1, \ldots, s'_n)$ denote the output sequence of the cumulative sum approach applied to another sequence of weights, constructed by replacing the subsequence $w_j, \ldots, w_{j+k}$ by its sum. The sequence S_1 can be transformed into S_2 as follows. First, if $s_i < j$, then $s'_i = s_i$. Second, if $s_i \in [j, j+k]$, then $s'_i = j$. Finally, if $s_i > j + k$, then $s'_i = s_i - k$. In other words, the cumulative sum approach is only affected partially if weights are aggregated as shown by Figure 1. Parts of the output sequence that correspond to non-aggregated weights are recoverable.

Let S_3 and S_4 be output sequences of applying the alias method to the same two sequences of weights. Sadly, there is no clear relationship between the elements of S_3 and S_4. The algorithm first calculates the average weight, w_a. Next, the entries of two tables are built by repeatedly combining two weights w_i and w_j for which $w_i < w_a \leq w_j$, to form entries of the two tables. Weight w_j is replaced by $w_j - w_a + w_i$ and w_i is removed. The process is repeated until all weights have been removed. With small changes to weights, the entries in this table can change drastically making the alias method unstable. Therefore, this paper focuses on parallelization of the cumulative sum approach, but the alias method is mentioned here since it has the best sequential performance and forms the baseline for comparison in the performance results shown in Section 3.

2.1 Naive Approaches to Parallelization

This paper considers only static load balancing, where each of p processors is assigned an equal share of n objects. Collecting all weights at a single processor to perform RWS leads to a centralized approach where the master processor quickly become the bottleneck, and more communication is required as n grows. Therefore, this approach is not considered further.

Let $w_{k,j}$ denote the weights of objects assigned to processor p_k. One straightforward approach to parallelization is to fix the assignment of objects to processors. First, each processor p_k shares all its local weights $w_{k,j}$ through an

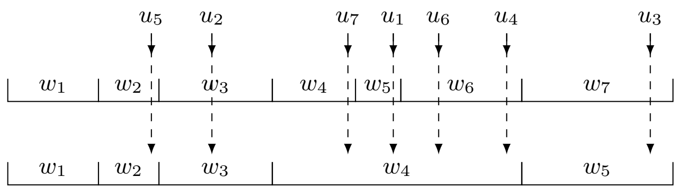

Fig. 1. Effect of replacing the subsequence w_4, w_5, w_6 by their sum. Given the same sequence of random numbers $(u_1, \ldots, u_7)$, where $u_i \sim \mathcal{U}(0, \sum_{i=1}^{7} w_i)$, the sequence at the top is $S_1 = (5, \mathbf{3}, \mathbf{7}, 6, \mathbf{2}, 6, 4)$ and the sequence at the bottom is $S_2 = (4, \mathbf{3}, \mathbf{5}, 4, \mathbf{2}, 4, 4)$. Bold indices are not effected or can be reconstructed.

all-to-all broadcast requiring $\mathcal{O}(n)$ time [5]. Next, since all weights are available, each processor builds the alias table in $\mathcal{O}(n)$ time and generates n/p samples in $\mathcal{O}(n/p)$ time. Each processor requests objects that it needs to initialize all its local output objects. Processors exchange objects by sending objects to their owner. The expected communication volume is $\mathcal{O}(n - n/p)$.

Alternatively, to save bandwidth, processors can also share the sum of their local weights, $W_k = \sum_{j=0}^{n/p} w_{k,j}$, in $\mathcal{O}(p)$ time. It might seem that the alias method could be used in this case as well. However, since the alias table would be built using the weights W_k, a different table would be built depending on p. If the parallel environment changes, the output of the sampling process will change as well, which precludes reproducible results. Instead, once all aggregate weights W_k are available, two cumulative sums are calculated in $\mathcal{O}(n/p + p)$ time and n samples are taken through a nested binary search in $\mathcal{O}(n \log(p) + (n/p) \log(n/p))$ time. Here, the first binary search is over the cumulative sum of W_k. If an object resides on p_k, a second binary search is performed over the cumulative sum of local weights, $w_{k,j}$. A single random number is used for both searches. Again, each object is sent to the processor to which it was assigned.

Three factors limit performance in both of these parallelizations. First, an all-to-all broadcast to share W_k causes communication volume to grow linearly in p. If $w_{k,j}$ are shared, communication volume also grows linearly in n. Second, each processor can communicate with every other processor when objects are exchanged. Third, the total expected communication volume to exchange objects, $\mathcal{O}(n - n/p)$, grows as either n or p increases.

2.2 Distributed Approach

The fundamental issue with the two approaches described above is that objects are assigned to processors and that this assignment is fixed. Instead, if objects are allowed to "move" in a way that minimizes communication required for exchanges, and reproducibility is maintained, efficiency can be improved.

Observe that each W_k will be distributed normally around $\sum_{i=0}^{p} W_i/p$ as n increases since all processors are treated equally. Hence the number of selected objects per processor is expected to be equal. The goal of the method presented in this paper is to exploit this fact to minimize communication. As noted earlier, the cumulative approach is parallelized. For this, each processor p_k needs to know only $\sum_{i=0}^{k-1} W_i$ and $\sum_{i=0}^{p} W_i$ since this determines the offset of its weights $w_{k,j}$ in the global context. Computing this prefix sum takes $\mathcal{O}(p)$ time [2]. In addition, $\sum_{i=0}^{p} W_i$ is needed to normalize the weights, which can be computed with an all-reduce which takes $\mathcal{O}(\log(p))$ time [5]. Next, a cumulative sum of weights $w_{k,j}$ is built locally. A single binary search suffices since a selection of objects owned by any of the processes $p_1, \ldots, p_{k-1}$ is detected directly. Finally, objects are renumbered in such a way that their identifier is independent of p.

Algorithm 1 summarizes these steps. Processor p_k draws u_i from the random generator of object i to determine where the selection is located. The total number of samples, q, for which the selected object is located at the processors $p_0, \ldots, p_{k-1}$ can be tracked since the prefix sum is available at processor p_k. Next, each processor maintains a count table of length n/p to track the number of times each local object is selected. Selections falling on processors $p_{k+1}, \ldots, p_p$, are ignored. After all n samples have been generated, the count table is traversed in $\mathcal{O}(n/p)$ time and objects are created with identifiers starting from q. The identifiers determine which processor owns the object. This renumbering step can be seen as moving objects around without communication.

Algorithm 1: Distributed RWS on processor p_k

```
Data: Objects $(o_1, \ldots, o_{n/p})$, associated weights $(w_{k,1}, \ldots, w_{k,n/p})$
Result: New objects $(o'_1, \ldots, o'_{n/p})$
$W_k = \sum_{j=0}^{n/p} w_{k,j}$, $W_{\text{total}} = \text{allReduce}(W_k, +)$, $W_{\text{below}} = \text{prefixSum}(W_k)$
$\text{countTable} = [0, \ldots, 0]$, $q = 0$
for $i = 1 \ldots n$ do
    $u_i \sim \mathcal{U}(0, W_{\text{total}})$
    if $u < W_{\text{below}}$ then
        $q = q + 1$
    else if $W_{\text{below}} < u < W_{\text{below}} + W_k$ then
        $s = \text{cumSumSearch}(u - W_{\text{below}}, (w_{k,1}, \ldots, w_{k,n/p}))$
        $\text{countTable}[s] = \text{countTable}[s] + 1$
end
for $i = 1 \ldots n/p$ do
    for $j = 1 \ldots \text{countTable}[i]$ do
        create new object from $o_i$ with identifier $q$
        $q = q + 1$
    end
end
rebalanceObjects()                ▷ Typically, few objects moved
```

Sums of local weights W_k will be distributed around $\sum_{i=0}^{p} W_i/p$. Hence, approximately the same number of objects will be selected from each processor and only deviations need to be corrected. This minimizes communication volume. Whenever two processors communicate, one processor will receive objects and the other processor will transmit objects, but never both. This is easy to see by dividing the processors into two groups: $p_1, \ldots, p_k$ and $p_{k+1}, \ldots, p_p$. If the first group has less than $k \times n/p$ objects, objects will be transmitted from the second group to the first. The opposite case is also possible. A useful consequence of the numbering scheme is that, in many cases, rebalancing can be achieved by transferring objects between neighboring processors p_k and p_{k+1}. Compared to the naive approaches from Section 2.1 where objects can travel in both directions and tend to travel between any pairs of processors, the presented renumbering scheme reduces network contention. Finally, since identifiers are determined from a global context, they do not depend on the number of processors. This makes the presented method reproducible across different parallel architectures.

3 Results

To evaluate performance in practice, a Message Passing Interface (MPI) implementation of Algorithm 1 is compared with the naive approaches described in Section 2.1. Results for the parallel alias method have been omitted since they almost coincide with the results for the naive cumulative approach. Random weights are used during each step. Execution time is averaged over 10 runs, each with a different RNG seed. Figure 2 shows speedup as the number of nodes, p, is increased. The number of objects, n, increases from 2^{14} to 2^{17} vertically. The object size increases from 1 byte to 2048 bytes horizontally.

The test cluster consists of 16 node interconnected with infiniband. Each node has two Intel X5660 processors, running at 2.80 GHz, for a total of 12 cores. Speedup, $S = T_s/T_p$, with respect to the fastest sequential algorithm is studied. Here, T_s is the sequential execution time of the alias method, and T_p is the execution time of the parallel versions with p processes, one for each system in the cluster. Each process consists of 12 threads which map to 12 cores.

First, while it is not clearly visible, both naive methods perform better on a single node than on multiple nodes. The added overhead caused by communication causes performance to degrade.

Second, in the distributed version, only aggregate information is exchanged, while information *per object* is exchanged in the naive versions. With more objects, the communication overhead during the steps leading up to the rebalancing phase for the distributed version will remain minimal. Comparing figures from top to bottom for a fixed object size shows that scalability improves with more objects. For example, with 2^{14} objects of 1 byte each, all approaches show poor scalability. Note that even in this case, the distributed version still outperforms the naive versions. Moving from 2^{14} objects to 2^{17} objects increases the speedup from 2.6x to 10x with 16 nodes.

Third, communication volume in the rebalancing phase is kept to a minimum in the distributed version. Hence, compared to the sequential execution time of the alias method, speedup increases as overhead in the rebalancing phase is kept to a minimum. Comparing results from left to right confirms this behavior. For example, with 2^{15} objects of 1 byte each, speedup is limited to 4x, but with objects of 1024 bytes, this limit increases to 10x.

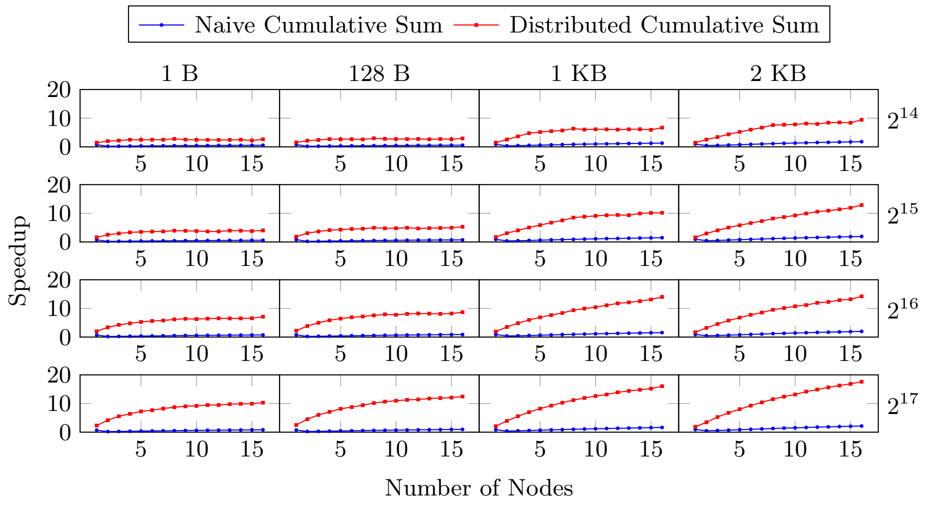

Fig. 2. Performance comparison of the parallel naive approaches described in Section 2.1 with the method presented in Section 2.2. Horizontally, object size increases from 1 byte to 2048 bytes. Vertically, the number of objects increases from 2^{14} to 2^{17}.

4 Related Work

Parallel genetic algorithms have been extensively studied in the past [3]. A single population can be managed by a master in a master-slave architecture. Again, since the master processor executes RWS, it can become the performance bottleneck. Alternatively, multiple populations can be evolved in parallel on multiple systems with occasional migrations between populations. While this improves utilization of the underlying parallel system, the output will depend on the number of processors. In contrast, the parallelization presented in Section 2.2 is only one step of genetic algorithms. It does not impact mathematical properties of the algorithm in which it is used.

Lipowski et al. [6] use rejection sampling to sample from a set of weights w_i. Although the authors do not discuss parallelization, the downside of their method is that its computational complexity is determined by the expected number of attempts before acceptance. This is given by $\max\{w_i\}/\sum_{i=0}^{n} w_i$ which depends on the distribution of weights. Using their method in a message passing environment, either all weights are shared, or repeated communication to share weights is required for each attempt. In contrast, the run time of the parallelization from Section 2.2 is independent of the distribution of the weights.

5 Conclusion and Future Work

While the results show that speedup starts to converge, the presented method outperforms the naive approaches. The biggest improvements are expected for use cases with large objects. In all of the tested configurations, the distributed version performs the best and is therefore the preferred approach.

This work uses static load balancing where each processor is assigned an equal number of objects n/p. In practice, RWS is executed iteratively after objects have been updated. Typically, the time required to update objects is imbalanced between consecutive calls to the RWS subroutine. For this reason, future work will focus on dynamic load balancing techniques like work stealing [9]. Instead of restoring balance after each iteration, objects will be stolen from neighboring processors, p_{k-1} and p_{k+1}, if those processors are lagging behind.

The loop over all n objects to generate random numbers on each processor causes speedup to converge as p increases. This part of the presented method can be interpreted as being executed sequentially. It is possible to partition the loop over all processors and have each processor maintain p count tables. However, the reduction in execution time is outweighed by the additional communication volume required to share all weights and count tables. Preliminary testing has shown that, as long as p is small, such partitioning is beneficial. Hence, future work will explore exchanging weights in sets of a few processors to *partially* parallelize the loop over all objects.

6 Acknowledgments

Part of the work presented in this paper was funded by Johnson & Johnson.

References

1. Nando de Freitas & Neil Gordon Arnaud Doucet, editor. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.
2. Guy E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990.
3. Erick Cant-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.
4. David E. Goldberg. *Genetic algorithms*. Pearson Education India, 2006.
5. Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.
6. Adam Lipowski and Dorota Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 391(6):2193–2196, 2012.
7. Pierre Del Moral, Ajay Jasra, Kody J. H. Law, and Yan Zhou. Multilevel sequential monte carlo samplers for normalizing constants. *ACM Trans. Model. Comput. Simul.*, 27(3):20:1–20:22, August 2017.
8. John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 16:1–16:12, New York, NY, USA, 2011. ACM.
9. Shigang Li, Jingyuan Hu, Xin Cheng, and Chongchong Zhao. Asynchronous Work Stealing on Distributed Memory Systems. pages 198–202. IEEE, February 2013.
10. Michael D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on software engineering*, 17(9):972–975, 1991.