REDISTRIBUTION NETWORKS FOR RESAMPLING

Michael Lunglmayr [*]

Institute of Signal Processing,
Johannes Kepler University Linz, Austria,
michael.lunglmayr@jku.at

Víctor Elvira [†]

School of Mathematics,
University of Edinburgh, United Kingdom,
victor.elvira@ed.ac.uk

ABSTRACT

Resampling is an important building block in core signal processing methods such as particle filters or genetic algorithms. This work describes how to accelerate the redistribution part of resampling, in a parallelized form utilizing a processing network composed of low-complexity nodes with $O(log_2^2(n))$ layers. We furthermore show how to use such networks for block-based processing of input vectors that are larger than the input of the network, allowing to trade-off the best of both worlds: block-based processing with its low area requirements and network-based processing with its high speed. We present simulation results not only showing the performance gains compared to the trivial linear method but also showing that by using the proposed architecture one can achieve processing times on edge devices that recently required high-performance server clusters.

***Index Terms*—** Resampling, Redistribution, network-based processing

1. INTRODUCTION

Resampling is an algorithmic part present in many statistical signal processing methods. Three notable use cases where resampling is an essential building block are particle filters [1], adaptive importance sampling [2], and evolutionary algorithms for optimization [3]. In particle filtering, a set of weighted particles approximates the filtering distribution of a latent state given available time-series observations. Particle filters sequentially process those observations by moving the particles and updating their associated weights accordingly. However, when few particles dominate the approximation (i.e., receiving significantly higher weights compared to the majority of the particles), a resampling step is needed in order to avoid the undesired so-called *particle degeneracy* [4, 5]. In adaptive importance sampling (AIS), a targeted distribution is approximated by a set of so-called proposal distributions that are improved over the iterations. Population Monte Carlo (PMC) algorithms are a family within AIS methods, where the set of proposals is adapted by performing resampling steps at each iteration [6, 7, 8, 9, 10]. A third important use case of resampling is in evolutionary algorithms such as algorithms for genetic programming (GP) [3, 11] where resampling can be employed to select individuals of a population to produce the next generation (the individuals of a population in GP have some similarities to the particles of a particle filter). For all these use cases, resampling is performed within a sequential algorithm. Thus, for real-time applications, speeding up the resampling operation is of great interest. While part of this process can be naturally parallelized, such as the evaluation of the weight of a particle (a more detailed description of resampling is given below), the parallelization is much harder for the redistribution step, the multiplying or discarding of particles. Especially when considering dedicated hardware acceleration, using a network structure composed of simple units (similar to sorting networks [12]) to parallelize the redistribution would be desirable.

The two main contributions of this paper are within the scope of implementing resampling for particle-based methods. In particular: a) We show, to the best of our knowledge for the first time, how to perform redistribution with a computation network composed of low-complexity elements in digital hardware. b) We furthermore show how to adapt the structure in a block-based manner to achieve a desired regime in the area/computation time trade-off. We demonstrate how this architecture enables performing the redistribution with low processing times that recently required high-performance server clusters.

2. BACKGROUND

2.1. Resampling

Resampling is a statistical task performing the replication of elements in a set, where the number of replicas of each element is, in expectation, proportional to a weight associated with each element. More precisely, let us consider the vector

[*]This work has been supported by the COMET-K2 "Center for Symbiotic Mechatronics" of the Linz Center of Mechatronics (LCM) funded by the Austrian federal government and the federal state of Upper Austria.

[†]This work has been supported by the *Agence Nationale de la Recherche* under grant ANR-17-CE40-0031-01, the Leverhulme Research Fellowship (RF-2021-593), and by ARL/ARO under grant W911NF-22-1-0235.

$\mathbf{x} \in \mathbb{R}^n$ with an associated vector of weights $\mathbf{w} \in [0,\ 1]^n$, with $\sum_{d=1}^{n} w(d) = 1$ and $w(d) \geq 0$, $d = 1, ..., n$. The resampling step consists of creating a vector $\mathbf{x}' \in \mathbb{R}^n$, generally in a stochastic manner, where each component $\mathbf{x}'$ also exists in $\mathbf{x}$. In the so-called unbiased resampling schemes, the number of replicas that each value $x(d)$ will appear in $\mathbf{x}'$ is in expectation $w(d) \cdot n$. The most common resampling is the so-called multinomial resampling, in which each element of $\mathbf{x}'$ is independent and identically distributed as the probability mass function defined by $\mathbf{x}$ and $\mathbf{w}$, which implies that, for all $d = 1, ..., n$, the element $x'(d) = x(j)$, with probability $w(j)$, $j = 1, ..., n$ (see more details in [1, 13]). Resampling can be performed by first a random sampling process based on the probabilities $w(j)$, $j = 1, ..., n$ (together forming a pmf), resulting in integers numbers $r(j) \in 0, \ldots, n$, $j = 1, ..., n$, describing how often a particle will be replicated after resampling. The vector $\mathbf{r}$ formed by the numbers $r(j)$ will be subsequently used in a so-called redistribution step forming a new vector of particles $\mathbf{x}'$ where particle j appears $r(j)$ times. In this, work we will focus on this second step.

2.2. State-of-the-art hardware implementations

To the best of the authors' knowledge, no computation network-based dedicated hardware of the redistribution has been proposed in literature before. Parallel algorithms have been proposed in the literature for multiprocessor environments. In [14], an algorithm for speeding up the redistribution on multiprocessor systems with time complexity $O(\log_2^2(n))$ has been proposed that has been refined to an algorithm with time complexity $O(\log_2(n))$ in [15]. However, due to relying on operations specific to multiprocessor systems of the state-of-the-art works, such as message passing over a common communication network, the approach described in this work, although having a time complexity of $O(\log_2^2(n))$ can be quite competitive as we describe in Section 4. This is due to the simple operations performed in our computation network, without the requirements of, for dedicated hardware implementations often costly, operations of a multiprocessor environment. The approach presented in this work allows parallelizing the redistribution operation on edge devices without requiring a server-based computation cluster.

3. REDISTRIBUTION NETWORKS

In the following, we propose a redistribution network performing the redistribution task in $O(\log_2^2(n))$ layers. For this, we assume that the tuples (x_i, r_i) are sorted in ascending order according to r_i (a task that can be performed with time complexity $O(\log_2^2(n))$ when using sorting networks [12]). All the splitting and sorting operations described below will be done on the elements of $\mathbf{r}$, the elements of $\mathbf{x}$ will be assumed to be routed accordingly. Using this once-sorted sequence, one can build a single-layer splitting network as

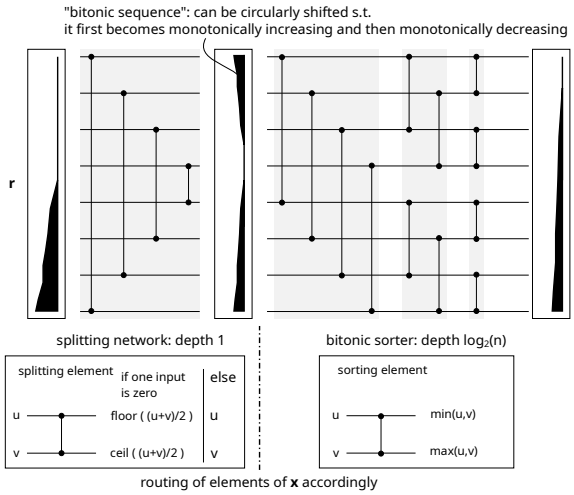

Fig. 1. Redistribution stage composed of splitting and bitonic sorting.

shown in Fig. 1 on the left, comprised of splitting elements pairwise connecting the lines of the network from the outside to the inside. The splitting elements work as follows. If one of the inputs of a splitting element is 0, the r value of the other input will be split using the shown strategy. If none of the inputs is 0, the elements will just be routed through. As the input vector of the splitting network was sorted in ascending order, the resulting sequence is bitonic, meaning that a bitonic sorting network [12] composed of $\log_2(n)$ layers of sorting elements can be used to obtain a sorted sequence again. Such a sorting element just routes its minimum input to the top output and the maximum element to the bottom output. Considering the worst case vector of $\mathbf{r}$, that is the vector $[n, 0, 0, \ldots, 0]$ one can see that $\log_2(n)$ redistribution stages of Fig. 1 are required, giving an overall number of layers (a layer is drawn in Fig. 1 as a gray rectangle) of $\log_2(n)(\log_2(n) + 1)$ required for redistribution of the sorted sequence. Combining this with a sorting network in front, that can be built from $\log_2(n)(\log_2(n) + 1)/2$ layers consisting of the sorting elements shown in Fig. 1, as described in [12], this means that redistribution can be done with

$$3/2 \log_2(n)(\log_2(n) + 1) \tag{1}$$

layers. Assuming for simplicity that each layer is processed in one clock cycle (because of the simplicity of the network elements, depending on the desired clock frequency, one can often process multiple layers in one clock cycle) one can obtain the redistributed vector in the same number of time steps. Considering that each layer has $n/2$ elements (either splitting elements or sorting elements), the overall number of required elements is $3/4n\ \log_2(n)(\log_2(n) + 1)$. Using a network structure in digital hardware as described above not only has the advantage of a significant reduction in terms of processing steps (the number of steps for the trivial redistribution grows linear in n) but also, that the redistributed elements are rep-

resented in parallel after redistribution (in contrast to being stored in a memory requiring sequential access). This allows for easy parallel processing (e.g. for calculating the weights of the particles in a particle filter) after redistribution. The limiting factor, however, when implementing this approach, is the required area for large n. A synthesis done for FPGAs from Intel (Stratix V) showed that for a sorting element (10 bits for the r and x inputs, respectively) 31 adaptive logic modules (ALMs), and for a splitting element 33 ALMs are used. As a reference, at the time of writing, the biggest FPGAs from Intel have up to 3.5 Mio ALMs, allowing for a maximum size of redistribution networks of 512 input elements. Similarly, when implementing a redistribution network on an ASIC, area requirement is the defining cost factor. As an alternative, as we demonstrate in the next section, one can divide a vector into blocks, and, by using a combination of a sorting and redistribution network twice the block length, one can still perform the redistribution of a vector blockwise (obtaining a result as if one would have implemented a full redistribution network). Although the time-complexity of this is not squared logarithmic anymore, by choosing a large enough block length, one can still obtain a significant speed-up compared to sequential redistribution.

3.1. Block-based redistribution

We now propose an alternative block-based redistribution structure for building the vector of resampled particles. When dividing the vectors $\mathbf{x}$ and $\mathbf{r}$ into blocks of size m (with $m < n$ and assuming n/m is integer) the elements of a block (subvector) of $\mathbf{r}$, $\mathbf{r}_z = [r_{zm+1}, r_{zm+2}, \ldots, r_{(z+1)m}]$ with $z \in \{0, \ldots, n/m - 1\}$, typically do not sum up to m anymore. If one would just use a redistribution network of size m, this would mean that either more elements of the block of $\mathbf{x}$, $\mathbf{x}_z = [x_{zm+1}, x_{zm+2}, \ldots, x_{(z+1)m}]$, are to be placed in the output block $\mathbf{x}'_z$ (if $\mathbf{r}_z^T\mathbf{1} > m$) or that the elements of $\mathbf{x}_z$ cannot fill an output block (if $\mathbf{r}_z^T\mathbf{1} < m$). To overcome these problems, we propose the structure of Fig. 2.

For a block size of m, it uses a sorting network and a redistribution network of input sizes $2m$, respectively. The second input of size m is used to merge outputs that would not be able to fill a full output block. This was based on the following observations. When analyzing the output $\mathbf{r}$ part of the redistribution network one can differentiate the following three cases that are meaningful for operation, as shown in Fig. 3. As the redistribution network always outputs the $\mathbf{r}$ part in a non-decreasing order (the $\mathbf{x}$ part is sorted accordingly) one can identify each of the cases by checking if the values $r_{1,\text{out}}$ and/or $r_{m+1,\text{out}}$ are zero.

For *case I*, an output block of size m could not fully be filled ($r_{\text{out}}(m+1) = 0$). In this case, the right parts (m elements) of the output blocks of $\mathbf{r}_{\text{out}}$ and $\mathbf{x}_{\text{out}}$ are fed back to the redistribution network allowing to merge it with the next input block that is fed in on the left part of the redistribution network.

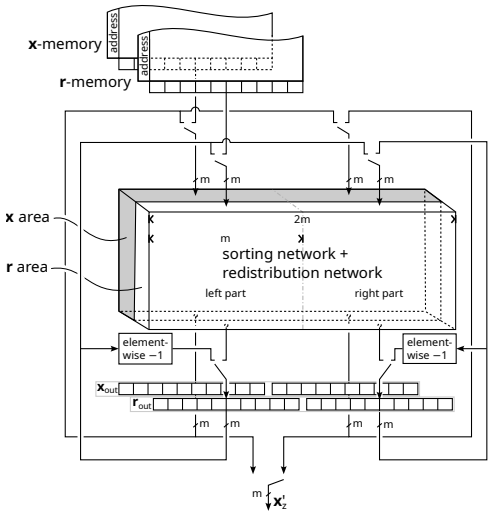

Fig. 2. Structure of block-based redistribution.

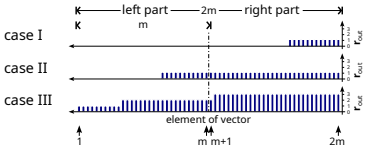

Fig. 3. Three cases to differentiate for operation of block-based redistribution

For *case II* ($r_{\text{out}}(1) = 0$), the left part cannot fill an output block but the right part can. In this case, the redistribution network distributes the elements in such a way that the non-zero $\mathbf{r}_{out}$-vales will be all ones. The corresponding $\mathbf{r}_{out}$ and $\mathbf{x}_{out}$ values of the right part will be routed to the output of the whole architecture (e.g. to a storing memory or to subsequent processing elements). The left part of the output will be fed back to the right part of the input of the sorting and redistribution network. The network will combine these feedback elements with a new block that is fed in from the left.

In *case III* ($r_{\text{out}}(1) \geq 1$), all $\mathbf{r}_{out}$ values of the distribution network are at least one (due to the non-decreasing order one can check this by testing if the first element of $\mathbf{r}_{out}$ is non-zero), meaning that each part can fill at least one output block. In this case, the left and right parts of $\mathbf{x}_{out}$ are alternatively routed to the output block $\mathbf{x}'_z$, each time element-wisely reducing the corresponding parts of the $\mathbf{r}_{out}$ element-wise by 1 and copying the corresponding $\mathbf{x}_{out}$-values to the output block. This is done for each part as long as its most left r value is still larger than 0 (tested via the values $r_{1,out}$ and $r_{m+1,out}$, respectively). If the right part of $\mathbf{r}_{out}$ is all zero (can be tested via $r_{2m,out}$) the left part is routed to the right part of the input and a new block is fed into the left. If, after this repeated elementwise reduction be one, both parts of $\mathbf{r}_{out}$ have non-zero elements at their end (indices m and $2m$ respectively), the vectors are fed to the input of the redistribution network, for merging of the two parts.

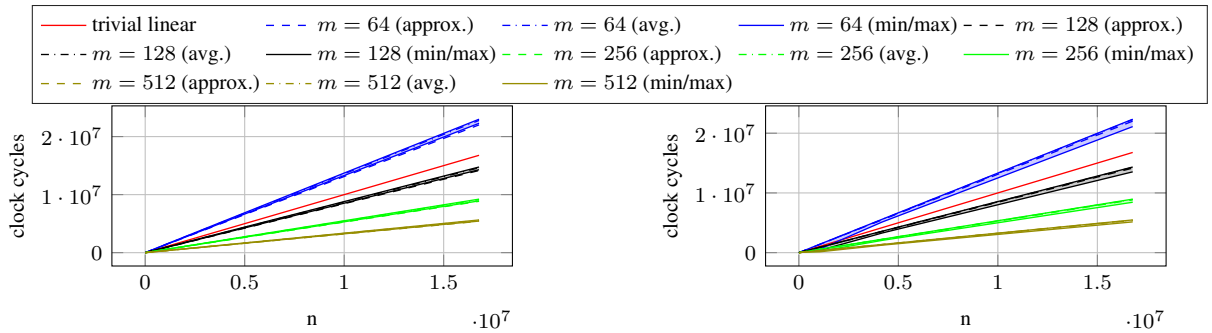

Fig. 4. Required clock cycles for redistribution, generalized Pareto distribution: $\sigma = n$, $u = 0$, $k = 1$ (left) and $k = 2$ (right).

The required merging operations by the redistribution network lead to more redistribution operations than the ideal n/m. However, as the simulation results in the next section show, the numbers of overhead redistribution operations for merging are typically small.

4. SIMULATION RESULTS

We perform numerical simulations to quantify the number of required clock cycles of the different variants of the proposed structure. We generated random numbers using a generalized Pareto distribution to form the weight vector $\mathbf{w}$. The reason for choosing this distribution is that recent works [16, 17] demonstrated that particle weights can be modeled with the generalized Pareto distribution (we also simulated other distributions showing no significant deviation from the results). We subsequently scaled and rounded the obtained random variables of $\mathbf{w}$ down to the next lower integer and then used residual resampling to obtain $\mathbf{r}$ with components summing up to n.

We simulated the sorting and redistribution networks for block lengths $m \in \{64, 128, 256, 512\}$ and counted the number of required block redistributions. Eq. (1) expresses the number of cycles required for a $2m$ sorting and re-distribution network (in this equation replacing n by $2m$). Multiplying the number of required block redistributions by the number of cycles required for performing a $2m$-long redistribution gives the overall number of clock cycles for a simulated test case. Fig. 4 shows these results for the different block lengths the location parameter $u = 0$ and a scale parameter $\sigma = n$, and shape parameter (also known as tail index [18]) $k = 1$ (left) and $k = 2$ (right), respectively (for a definition of the generalized Pareto distribution we e.g. refer to [17]). The curves marked with "approximation" are the values $n/m \ 3/2 \ \log_2(2m)(\log_2(2m) + 1)$. The filled areas of the simulated curves are spanned by the minimum and maximum numbers of cycles obtained when simulating 100 test cases for each block length. For comparison, we included a curve marked as "trivial linear" assuming that a single clock cycle is used for each element when performing redistribution in a sequential manner which can be seen as a lower bound for sequential redistribution (due to memory accesses one will typically require more clock cycles for sequential redistribution). The experiments show that, for block lengths of 128 and above, using block-based redistribution with the described networks results in a significant speedup compared to the sequential approach. When increasing the shape parameter to 2 one can see that the minimum numbers of the clock cycles drop down (a general pattern we observed, higher k lead to lower minimum numbers of clock cycles; the maximum always stayed slightly above the approximation curves). To put the numbers of clock cycles in perspective, we can compare the results to the work of [15], where a redistribution algorithm with time complexity $O(\log_2(n))$, specifically tailored to multicore processor environments was proposed. There, when using eight interconnected servers utilizing 256 processors each running at 2 GHz, for $n = 2^{24}$ runtimes of about 2^{-6} seconds are reported. When comparing this to our block-based clock cycle simulations, e.g. for block lengths of 256 we require about $9 \cdot 10^6$ clock cycles for a redistribution of vectors of the same length. When implementing this on dedicated hardware running at 600 Mhz we would achieve a similar computation time, without the need of a server cluster. This enables parallelizing the redistribution operation with comparable speed on edge devices.

5. CONCLUSION

In this work, we have proposed resampling networks for parallelizing the resampling process in digital hardware that are composed of low-complex processing nodes. We further showed how to use such resampling networks for input lengths much larger than the input size of the resampling network in a block-based manner trading area vs processing speed while still outputting correctly redistributed vectors. We showed simulation results demonstrating the gains in terms of required processing clock cycles. These results demonstrate that, when implementing the proposed architecture as dedicated digital hardware, one can perform the redistribution operation on this dedicated hardware with moderate clock frequencies achieving processing times that recently required high-performance server clusters.

6. REFERENCES

[1] Tiancheng Li, Miodrag Bolic, and Petar M Djuric, "Resampling methods for particle filtering: classification, implementation, and strategies," *IEEE Signal processing magazine*, vol. 32, no. 3, pp. 70–86, 2015.

[2] M. F. Bugallo, V. Elvira, L. Martino, D. Luengo, J. Míguez, and P. M. Djuric, "Adaptive importance sampling: The past, the present, and the future," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 60–79, 2017.

[3] Leonardo Vanneschi and Riccardo Poli, *Genetic Programming — Introduction, Applications, Theory and Open Issues*, pp. 709–739, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[4] Petar M Djuric, Jayesh H Kotecha, Jianqui Zhang, Yufei Huang, Tadesse Ghirmai, Mónica F Bugallo, and Joaquin Miguez, "Particle filtering," *IEEE signal processing magazine*, vol. 20, no. 5, pp. 19–38, 2003.

[5] Arnaud Doucet, Adam M Johansen, et al., "A tutorial on particle filtering and smoothing: Fifteen years later," *Handbook of nonlinear filtering*, vol. 12, no. 656-704, pp. 3, 2009.

[6] O. Cappé, A. Guillin, J. M. Marin, and C. P. Robert, "Population Monte Carlo," *Journal of Computational and Graphical Statistics*, vol. 13, no. 4, pp. 907–929, 2004.

[7] V. Elvira, L. Martino, D. Luengo, and M. F. Bugallo, "Improving Population Monte Carlo: Alternative weighting and resampling schemes," *Signal Processing*, vol. 131, no. 12, pp. 77–91, 2017.

[8] Víctor Elvira, Luca Martino, David Luengo, and Mónica F Bugallo, "Population monte carlo schemes with reduced path degeneracy," in *2017 IEEE 7th International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*. IEEE, 2017, pp. 1–5.

[9] Caleb Miller, Jem N Corcoran, and Michael D Schneider, "Rare events via cross-entropy population monte carlo," *IEEE Signal Processing Letters*, vol. 29, pp. 439–443, 2021.

[10] Víctor Elvira and Emilie Chouzenoux, "Optimized population monte carlo," *IEEE Transactions on Signal Processing*, vol. 70, pp. 2489–2501, 2022.

[11] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee, *A Field Guide to Genetic Programming*, Lulu Enterprises, UK Ltd, 2008.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Second Edition*, The MIT Press and McGraw-Hill Book Company, 2001.

[13] Randal Douc and Olivier Cappé, "Comparison of resampling schemes for particle filtering," in *ISPA 2005. Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, 2005.* IEEE, 2005, pp. 64–69.

[14] Alessandro Varsi, Lykourgos Kekempanos, Jeyarajan Thiyagalingam, and Simon Maskell, "Parallelising particle filters with deterministic runtime on distributed memory systems," in *IET 3rd International Conference on Intelligent Signal Processing (ISP 2017)*, 2017, pp. 1–10.

[15] Alessandro Varsi, Simon Maskell, and Paul G. Spirakis, "An o(log2n) fully-balanced resampling algorithm for particle filters on distributed memory architectures," *Algorithms*, vol. 14, no. 12, 2021.

[16] Aki Vehtari, Andrew Gelman, and Jonah Gabry, "Practical bayesian model evaluation using leave-one-out cross-validation and waic," *Statistics and Computing*, vol. 27, no. 5, pp. 1413–1432, Sep 2017.

[17] Aki Vehtari, Daniel Simpson, Andrew Gelman, Yuling Yao, and Jonah Gabry, "Pareto smoothed importance sampling," 2022.

[18] Hansheng Wang and Chih-Ling Tsai, "Tail index regression," *Journal of the American Statistical Association*, vol. 104, no. 487, pp. 1233–1240, 2009.