ODD-EVEN MERGING NETWORK: A SOFTWARE IMPLEMENTATION FOR ARBITRARY LENGTHS

A PREPRINT

Antoine Gansel
Chair for Data Security and Cryptography
University of Regensburg
Regensburg, Germany
gansela.research@gmail.com

March 6, 2025

ABSTRACT

Sorting networks are of significant interest in the field of Multi-Party Computation (MPC), having the potential to decrease execution time and costs. In particular, Batcher's Odd-Even merge sort is believed not to be far from optimal regarding its computational complexity. However, and despite being widely known and used for some 60 years, one cannot find any existing software implementation or algorithm for this merging (and, by extension, sorting) network that would be suitable for lists of arbitrary length. This short paper fixes the gap in the literature and briefly discusses the interest of Batcher's sorting network for MPC.

Keywords Odd-Even Merge, Sorting Networks, Multi-Party Computation

1 Introduction

Sorting networks are a class of sorting algorithms with constant execution patterns, which is thus of particular interest in hardware implementation. Because they are often highly contextual (i.e., optimised or available only for specific sizes) or accept relatively high complexities, sorting networks have been mainly absent from most of computer science. This family of sorting algorithms could however be of significant interest in the field of Multi-Party Computation (MPC), as their constant execution pattern (i.e., independent from the content of the list) eliminates the need for additional third parties when trying to obliviously sort a list, thus efficiently limiting the costs associated with the protocol execution as well as the strain put on the network.

In particular, the Odd-Even Merge Sort introduced, by K.E Batcher in 1968 [1], presents a computational complexity of $\mathcal{O}(n\log_2(n)^2)$, which is believed to be close to optimal for generalisable sorting networks [5] (i.e., sorting networks which logic can be extended to all lengths). However, despite being initially presented as compatible with lists of arbitrary lengths (and despite the frequent claims of obviousness one can find online), it is impossible to find any software implementation or algorithms for this network that would accept lengths that are not in the power of 2. Indeed, all implementation works seem to refer, directly or indirectly, to [8]. However, we do denote a hardware implementation by C. Kuo and Z. Huang [6] able to sort any list of size lower or equal to N, for an arbitrary N specified during implementation.

Our Contribution - In this paper, we present a recursive software implementation of Batcher's Odd-Even merge sort, running without additional memory for any list of arbitrary size. We prove the correctness of our algorithm and give an intuition of its security in the semi-honest model [7], provided that the "compare and swap" operation (c.f., Ideal Functionality 1) is correctly initialised in the Arithmetic Black Box model [2, 9]. The Rust implementation of the presented algorithms is publicly available on GitHub[1].

[1] https://github.com/A-GX/Generalised_Odd-Even-Merge

Organisation - In Section 2, we present the functionality one needs to provide to our program beforehand. We then proceed, in Section 3, with presenting our algorithms and formally proving their correctness. Finally, we conclude by briefly discussing how to prove the privacy of the merge sort using the Arithmetic Black Box model in Section 4, and by discussing a possible optimisation for our protocol in Section 5.

2 Preliminaries

The execution of a sorting network requires the existence of a valid ordering notion over the list elements, as well as the possibility of exchanging (swapping) those elements when necessary. In this article, when we speak of the "compare-swap" operation we refer to the Ideal Functionality 1. This also implies that, when we mention the computational complexity of our algorithm, we are making direct reference to how often the "compare-swap" functionality is called.

Note that we assume the functionality to be implemented by a trusted third party as anticipation of the discussion in Section 4. This will indeed allow us to claim it is executed by the Oracle of the Arithmetic Black Box.

Ideal Functionality 1 $\mathcal{F}_{CompareSwap}(list, a, b)$: Compare elements at indices a and b in $list$, and swap them if the ordering is not correct (Here, we aim for an ascending ordering)

Inputs: Any list of data $list$ with two valid indexes a, b with $a \leq b$
Output: List $list$ whose elements at indexes a, b are in order : $list[a] \leq list[b]$

Ideal trusted third party $\mathcal{T}_1$ executes :

1. $\mathcal{T}_1$ compares elements $list[a]$ and $list[b]$
2. If $list[a] > list[b]$, $\mathcal{T}_1$ swaps both elements
3. Otherwise, $\mathcal{T}_1$ does not modify the ordering.

Out. $\mathcal{T}_1$ output the updated list $list$.

3 Odd-Even Merge Sorting Network

It is straightforward to see that the correctness of the sorting network (c.f., Protocol 1) directly follows from the correctness of the merging protocol Π_{merge} it references (c.f., Protocol 2). As such, in this section, we exclusively focus on proving the validity of Theorem 2 by a recursive argument. Before presenting our solution, we recall the ideal Odd-Even merge functionality, as originally stated by Batcher in 1968.

Ideal Functionality of the Merging Network in [1] - An s by t merging network can be built by presenting the odd-indexed numbers of the two input lists to one small merging network (the odd merge), presenting the even-indexed numbers to another small merging network (the even merge) and then comparing the outputs of these small merges with a row of comparison elements. The lowest output of the odd merge is left alone and becomes the lowest number of the final list. The i^{th} output of the even merge is compared with the $i + 1^{th}$ output of the odd merge to form the $2i^{th}$ and $2i + 1^{th}$ numbers of the final list for all applicable i's. This may or may not exhaust all the outputs of the odd and even merges; if an output remains in the odd or even merge it is left alone and becomes the highest number in the final list.

Intuitively, the small odd and even merging networks called by the protocol essentially behave as smaller versions of the said protocol. Their inputs are split into odd and even indexes again and fed to smaller mergers, whose outputs will then be recombined by comparing adjacent pairs of even/odd elements. Considering a list with a size of the form 2^N with two sorted halves, the implementation is fairly straightforward (c.f. Figure 1.a.). Notably, the step between every neighbouring element in the list received by a small odd or even merger will always be of 2^n, with $n - 1$ the number of mergers the list went through. This implies that the functionality can easily be implemented by a recursive algorithm, that doubles its step size each time it goes down in the recursion (which corresponds to [8], as well as all implementations readily available online). For all other list sizes, however, we quickly run into an issue as the step between each element won't always be constant anymore (c.f. Figure 1.b.). To solve this, we decided in Protocol 2 to keep track of the starting index of both sublists across the recursion. This way we can correctly split the sorted sublists into their odd and even elements, and adaptatively step over one sublist to the other when recombining the outputs of the odd/even mergers.

Theorem 1 (Correctness of Π_{sort})**.** *The protocol Π_{sort} (Protocol 1) correctly sorts any list of arbitrary size given that the correctness of Π_{merge} holds.*

Theorem 2 (Correctness of Π_{merge})**.** *The protocol Π_{merge} (Protocol 2) correctly implements the ideal functionality specified by Batcher [1] (i.e., correctly merges) for any two lists A, B of arbitrary sizes l, q respectively.*

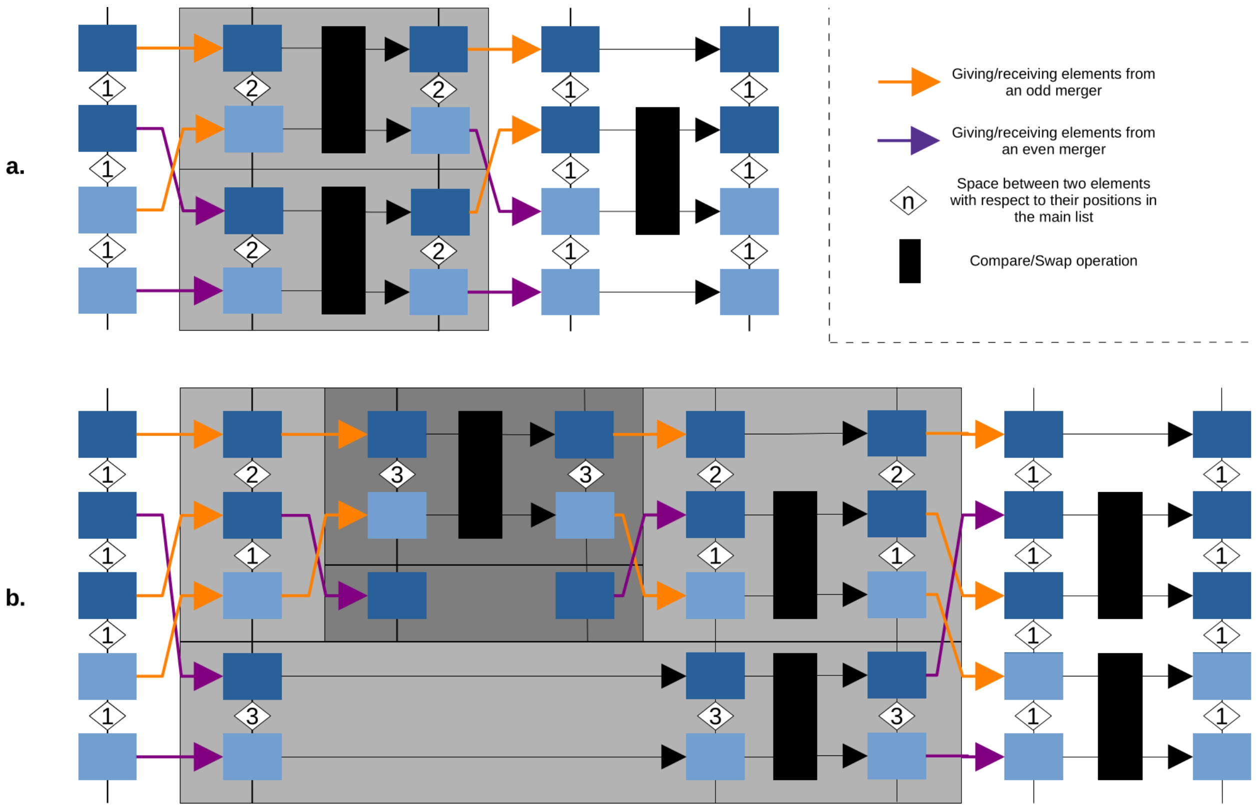
a.
b.
Giving/receiving elements from an odd merger
Giving/receiving elements from an even merger
Space between two elements with respect to their positions in the main list
Compare/Swap operation
n
1
2
3

Figure 1: Construction of **(a)** a "2 by 2" and **(b)** a "3 by 2" merging network

Protocol 1 $\Pi_{sort}(list, start, len)$: Recursively split the given list in two (not necessarily equal) halves until it obtains lists of size 1 (i.e., sorted), before recombining them by calling the merging procedure.

Inputs:
- $list$ a vector
- $start$ start index of sub-vector
- len length of sub-vector

Output: $\forall a, b \in \{i \mid \exists list[i]\}, a \leq b \implies list[a] \leq list[b]$

```
1: $nl \leftarrow \lceil len/2 \rceil$
2: $nr \leftarrow \lfloor len/2 \rfloor$
3: $\Pi_{sort}(list, start, nr)$
4: $\Pi_{sort}(list, start + nr, nl)$
5: $\Pi_{merge}(list, nl, nr, start, start + nl, 1)$
```

Proof of Theorem 2. We follow the sketch made by K.E. Batcher in his original work [1]: Given $A = [a_0, ..., a_{l-1}]$ and $B = [b_0, ..., b_{q-1}]$ two ordered list, we note $O = [o_0, o_1, ...]$ ($E = [e_0, e_1, ...]$) the result of their odd (resp. even) merge. The final expected result $C = [c_0, ..., c_{l+q-1}]$ requires :

$$\begin{aligned} c_0 &= e_0 \\ c_i &= min(o_{i-1}, e_i), \forall i \geq 1,\ i\ odd \\ c_{i+1} &= max(o_{i-1}, e_i), \forall i \geq 1,\ i\ odd \end{aligned} \tag{1}$$

We now prove through a recursive argument that the output list, at every step of our recursion, does satisfy Equation (1).

Iterative Assumption - Assuming $l, q \leq 1$, it is clear that we do output an ordered list C by comparing and swapping the elements, independently of our depth in the recursion. As such, we make the iterative assumption that we can correctly merge any two correctly ordered lists of at most $n - 1$ elements.

Protocol 2 $\Pi_{merge}(list, nl, nr, sr, sl, step)$: Given the start and end position of two sorted sublists in $list$, recursively merge the odd and even indexes of the sublists before compare-swapping every Odd-Even adjacent element (see the ideal functionality mentioned earlier)

```
Inputs:
  • $list$ a vector
  • $nl$ ($nr$) # elements in left (right) sub-vector
  • $sl$ ($sr$) start index of left (right) sub-vector
  • $step$ # of index between two neighbouring elements of a same sub-vector
  • $\forall a, b \in \{i \mid i \in [sr, sr + nr * step], i \equiv sr \mod step\}$ (resp $\forall a, b \in \{i \mid i \in [sl, sl + nl * step], i \equiv sl \mod step\}$), $a \leq b \implies list[a] \leq list[b]$
Output: $\forall a, b \in \{i \mid \exists list[i]\}, a \leq b \implies list[a] \leq list[b]$
1:  if $(nl \geq 1$ & $nr > 1) \parallel (nl > 1$ & $nr \geq 1)$ then
2:      $n\_even\_l \leftarrow \lceil nl/2 \rceil$
3:      $n\_even\_r \leftarrow \lceil nr/2 \rceil$
4:      $n\_odd\_l \leftarrow \lfloor nl/2 \rfloor$
5:      $n\_odd\_r \leftarrow \lfloor nr/2 \rfloor$
6:      $ns \leftarrow step * 2$
7:      $\Pi_{merge}(list, n\_even\_l, n\_even\_r, sl, sr, ns)$
8:      $\Pi_{merge}(list, n\_odd\_l, n\_odd\_r, sl + step, sr + step, ns)$
9:      $i \leftarrow 1$
10:     while $i + 1 < nr + nl$ do
11:         if $i \geq nl$ then
12:             $\Pi_{CompareSwap}(list, sr + (i - nl) * step, sr + (i - nl + 1) * step)$
13:         else if $i + 1 == nl$ then
14:             $\Pi_{CompareSwap}(list, sl + i * step, sr)$
15:         else
16:             $\Pi_{CompareSwap}(list, sl + i * step, sl + (i + 1) * step)$
17:         end if
18:         $i \leftarrow i + 2$
19:     end while
20: else if $(nl = 1)\&(nr = 1)$ then
21:     $\Pi_{CompareSwap}(list, s\_l, s\_r)$
22: end if
```

Iterative Argument - Building on the initialisation of the iterative rule above and on the assumptions that we received two order lists of size $l, q \leq n$ as input, we can assume we correctly create the lists O and E (which sizes are clearly at most $n - 1$). It thus remains to prove that the while-loop defined across lines 10 to 19 correctly enforces the condition expressed in Equation (1), no matter at what depth we are in the recursion.

Taking s_a, s_b the starting indexes of sublists A and B respectively and s the step to take to move from one element to the other inside our sublists (typically $s = 2^d$, with d the depth of our recursion), we observe the following relation:

$$\begin{aligned}[e_0, e_1, ...] &= [a_{s_a}, a_{s_a+2*s}, .., a_{s_a+(\lceil l/2 \rceil -1)*s}, b_{s_b}, b_{s_b+2*s}, .., b_{s_b+(\lceil q/2 \rceil -1)*s}] \\ [o_0, o_1, ...] &= [a_{s_a+1*s}, a_{s_a+3*s}, .., a_{s_a+(\lfloor l/2 \rfloor -1)*s}, b_{s_b+1}, b_{s_b+3*s}, .., b_{s_b+(\lfloor q/2 \rfloor -1)*s}]\end{aligned} \tag{2}$$

As such, we can rewrite Equation (1) depending on A and B, according to three cases:

- $\mathbf{i + 1 < l,\ i\ odd}$

$$\begin{aligned} c_0 &= a_{s_a} \\ c_i &= min(a_{s_a+i*s}, a_{s_a+(i+1)*s}), \\ c_{i+1} &= max(a_{s_a+i*s}, a_{s_a+(i+1)*s}), \end{aligned} \tag{3}$$

- $\mathbf{i + 1 = l,\ i\ odd}$

$$\begin{aligned} c_i &= min(a_{s_a+i*s}, b_{s_b}), \\ c_{i+1} &= max(a_{s_a+i*s}, b_{s_b}), \end{aligned} \tag{4}$$

- $\mathbf{i \geq l,\ i\ odd}$

$$\begin{aligned} c_i &= min(b_{s_b+(i-l)*s}, b_{s_b+(i-l+1)*s}), \\ c_{i+1} &= max(b_{s_b+(i-l)*s}, b_{s_b+(i-l+1)*s}), \end{aligned} \tag{5}$$

By observing that our index i is indeed always odd (c.f., Protocol 2, lines 9 and 18), one can easily see that Equation (3), (4) and (5) are correctly implemented by instructions at line 15, 13 and 11 respectively. As such, the output of our protocol is, at every point in the recursion (and assuming the validity of our step s), correct with respect to Batcher's specifications.

We can see the corectness of our step value s as follows: at depth 0, at the start of our recursion, all elements inside a same sublist are next to one another. As such, we need to call the protocol with argument $s = 2^0 = 1$. Each time we go down in the recursion, the element of the new sublists will be formed by the odd or even elements from the previous one. This implies we need to double the step value each time we enter a new recursive depth such as to obtain $s = 2^d$ at depth d. This behaviour is correctly implemented by line 6.

Note that we prove the merging network for sorting in the ascendant order without loss of generality. It is clear that the result derived here extends to any other sorting rule, as long as it is correctly implemented by the "compare-swap" operation.

□

4 Privacy in the Arithmetic Black Box Model

In this section, we briefly discuss how to show the privacy (i.e., security in the semi-honest model [7]) of our protocols given a correctly initialised Arithmetic Black Box [2, 9] implementing the Ideal Functionality 1. As the goal here is only to give an intuition, we do not extend the explanation to the inner mechanisms of the different frameworks and theorems used here, and instead refer to [7, 2, 9] for further information.

Theorem 3 (Privacy of Π_{merge})**.** *Protocol Π_{merge} (Protocol 2) is private, given an oracle access to a private implementation of $\mathcal{F}_{CompareSwap}$ (Ideal Funcitonality 1).*

Proof of Theorem 3. We need to show that the access pattern generated by our merging network is, assuming the "compare-swap" ideal functionality is successfully implemented in a correctly initialised Arithmetic Black Box [2, 9], computationally equivalent to a fully random one. To do so, we reason on the view a potential adversary would have of our protocol.

We can sum up the view of any of the parties as $(in, r, abb_1, ..., abb_x)$, setting in the party's input, r its random tape and $abb_1, ..., abb_x$ all the messages exchanged with the oracle of the Arithmetic Black Box when calling for the "compare-swap" functionality. Note that, because Π_{merge} is a merging network, its view is constant. As such, simulating this part of the party's view is trivial and can be ignored in the above representation, where it thus only remains the messages exchanged with the Arithmetic Black Box. We can thus argue, thanks to Goldreich's composition theorem [7], that the view of any of our parties can be efficiently simulated by anyone having access to an Oracle simulating the Arithmetic Black Box. Because we assume the Arithmetic Black Box is correctly initialised and correctly implemented, every message abb_n is computationally equivalent to random elements. As such, the privacy of the presented protocols follows from the privacy of the Arithmetic Black Box (i.e., its correct initialisation and implementation). □

As one can trivially prove the privacy of the sorting algorithm with the same argument, we only state the theorem as is.

Theorem 4 (Privacy of Π_{sort})**.** *Protocol Π_{sort} (Protocol 1) is private, given an oracle access to a private implementation of Π_{merge} (Protocol 2).*

Notice, however, that those proofs do not trivially extend to the security in the Malicious Model [7], for which our protocols need to be modified to ensure a potential malicious adversary would not deviate from the intended execution.

5 Future Work

The previous implementations of the Odd-Even merge and sorting networks [5, 8] have been widely studied to efficiently parallelise it [4, 10], reaching a time complexity of $\mathcal{O}(n \log_2(n))$ for the sorting network. The presented protocols should also accept such optimisation, though one still needs to verify to what extent can the existing works be applied to the protocols presented herein.

Notice, however, that parallelisation does not necessarily translate well to the field of MPC. In particular, for primitives such as Secret Sharing which asks for a significant amount of communication for every operation, one will often be limited by the available bandwidth. Because parallelisation leaves the computational complexity untouched (i.e., $\mathcal{O}(n \log_2(n)^2)$, the bandwidth limitation will likely render the improvement negligible.

This work develops a subsection of my master thesis that I effectuated at the University of Twente under the supervision of Dr. Ing. Florian Hahn and Yoep Kortekaas [3]. While the capabilities of the sorting network did not change, I extended the merging algorithm to accept two lists of size l and q for any tuple $(l, q) \in \mathbb{R}^2$, instead of lists of size l and $l + 1, \forall l \in \mathbb{R}$ like originally presented in the master thesis.

References

[1] K. E. Batcher. "Sorting Networks and Their Applications". In: New York, NY, USA: Association for Computing Machinery, 1968.

[2] Ivan Damgård et al. "Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation". In: *Theory of Cryptography*. 2006.

[3] Antoine Gansel. *Sub-quadratic Privacy Preserving Cohort Selection*. Aug. 2023.

[4] Peter Kipfer, Mark Segal, and Rüdiger Westermann. "UberFlow: A GPU-based particle engine". In: *Graphics Hardware 2004 - Eurographics Symposium Proceedings*. 2004.

[5] Donald E Knuth. *The art of computer programming: Volume 3: Sorting and Searching*. USA: Addison-Wesley Professional, 1998.

[6] C.J. Kuo and Z.W. Huang. "Modified odd-even merge-sort network for arbitrary number of inputs". In: *IEEE International Conference on Multimedia and Expo, 2001. ICME 2001.*

[7] Goldreich Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. 1st. USA: Cambridge University Press, 2009.

[8] R. Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching*. Pearson Education, 1998.

[9] Tomas Toft et al. "Primitives and applications for multi-party computation". In: *Doctoral dissertation, University of Aarhus, Denmark* (2007).

[10] Keliang Zhang et al. "GPU accelerate parallel Odd-Even merge sort: An OpenCL method". In: *Proceedings of the 2011 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. 2011.