Parallel Algorithms Align with Neural Execution

Valerie Engelmayer[*]
University of Augsburg
valerie.engelmayer@gmail.com

Dobrik Georgiev
University of Cambridge
dgg30@cam.ac.uk

Petar Veličković
Google DeepMind
petarv@google.com

Abstract

Neural algorithmic reasoners are parallel processors. Teaching them sequential algorithms contradicts this nature, rendering a significant share of their computations redundant. Parallel algorithms however may exploit their full computational power, therefore requiring fewer layers to be executed. This drastically reduces training times, as we observe when comparing parallel implementations of searching, sorting and finding strongly connected components to their sequential counterparts on the CLRS framework. Additionally, parallel versions achieve (often strongly) superior predictive performance.

1 Motivation

Classical algorithms often pose a bottleneck to information processing [1]. They are usually designed to deal with consistent, totally ordered, abstract quantities, while in reality, we need to reason about noisy, high-dimensional data. Machine learning and neural networks (NNs) in particular enable machines to extract useful features from such inputs, but if their outputs need to be composed with a non-differentiable algorithm, they cannot learn from direct feedback via backpropagation. Moreover, compressing information in a way that makes the algorithm applicable loses a lot of potentially relevant detail. Breaking this bottleneck by teaching NNs how to execute algorithms is the objective of *neural algorithmic reasoning* [1–3]. First applications to real-world data are promising [4–6], but extrapolation still has room for improvement even on highly elaborate architectures [7, 8]. Therefore, there is a clear need to investigate neural networks' information processing capabilities more closely.

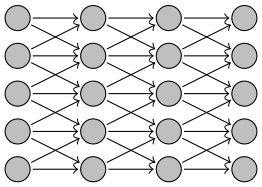
When executing algorithms, NNs act as computational machines. In graph neural networks (GNNs), graph nodes take on the role of storage space (interpreting edge labels as nodes adjacent to its endpoints throughout this paper), while edges indicate which ways information may flow. The update function of choice defines the set of constant (neural) time operations. But note how nodes update their features *in parallel*, each one acting as a processor of its own rather than sheer memory.

The parallel nature of neural networks is widely known. Running them in parallel fashion on processing devices like GPUs and TPUs drastically saves computational resources [9, 10]. It seems natural that this translation between computational models would also hold the other way around. And indeed, Loukas [11] proves how GNNs are analogous to distributed computational models under certain assumptions. Kaiser and Sutskever [12] exploit the advantages of parallel processing in their Neural GPU. The differentiable sorting algorithms by Petersen et al. [13] operate in parallel. Freivalds et al. [14] derive their architecture from the parallel computational model of Shuffle-Exchange Networks. Xu et al. [15] observe how their model learns to compute a shortest path starting from both ends in parallel when executing the Bellman-Ford algorithm. Veličković et al. [3] and Veličković et al. [16] hint at the favourability of using parallelized computations whenever possible.

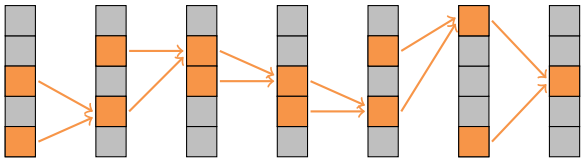
It is time the parallel processing capabilities of NNs are exploited systematically, and this paper takes a relevant step in that direction. Theory on parallel computational models and algorithms explicitly designed for them are abundant [17–19]. Their trajectories are shorter and align more closely with neural architectures, as illustrated in Figure 1. Hinting at these during training teaches NN to execute algorithmic tasks much more efficiently than when providing hints for sequential algorithms, as

[*]Corresponding author.

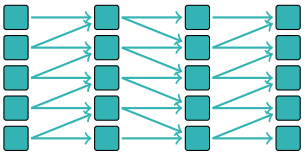
V. Engelmayer et al., Parallel Algorithms Align with Neural Execution. *Proceedings of the Second Learning on Graphs Conference (LoG 2023)*, PMLR 231, Virtual Event, November 27–30, 2023.



(a) Neural processing is highly parallel.



(b) Trajectories of sequential algorithms are sparse.



(c) Trajectories of parallel algorithms are denser and shorter.

Figure 1: Trajectories of sequential and parallel algorithms, as well as neural processing.

we demonstrate in Section 5 for the examples of searching, sorting and finding strongly connected components. While it is common practice to modify the neural architecture for better alignment [7, 15, 20–22], it seems promising to narrow the gap from the other side, by choosing algorithms that naturally align with neural execution.

2 Parallel Computing

Fundamentally, the parallel computational models addressed here assume multiple processors collaborating to solve a task. The line between parallel and distributed computing is blurry and depends on how controlled the interactions between processors are. We assume a fixed and known interconnection graph, uniquely identified processors and a common clock to govern computation. Therefore, we choose to speak of parallel computing.

2.1 Parallel Computational Models

Processor Arrays. Communication may take place via hard-wired channels between the processors. These induce an interconnection graph that may in principle take any shape. At every time step, each processor executes some computation based on the contents of its local memory and the information received from its neighbours in the previous step, and may, in turn, send out a tailored message through any of its channels.

PRAM Models. Alternatively, communication may be realised by reading from and writing to global memory, giving rise to *PRAM* (parallel random access machine) models [17]. Submodels allowing for *concurrent* reading and writing by multiple processors are referred to as *CRCW* PRAM. Different conventions exist on whether attempting to concurrently write different values is permitted, and if so, how to decide who succeeds. In the most powerful model, the *priority* CRCW PRAM, the value from the processor with the lowest index taking part in the concurrent write will be taken on.

2.2 Efficiency

Since multiple steps can be carried out at the same time, the required number of operations in a parallel algorithm does not impose a lower bound to its run time as in the sequential case, but the product of time and processor number. *Optimal speedup* is achieved if the use of n processors speeds up computation by a factor of n. This gives rise to a notion of efficiency frequently used in parallel computing [17].

Definition 1. The *efficiency* of a parallel algorithm solving a task of sequential complexity C on p processors in time t is defined as

$$\frac{C}{pt}.$$

It is not hard to see that optimal speedup entails an efficiency of $\Omega(1)$.

2.3 Examples of Parallel Algorithms

Searching. For a simple parallel search for value x in a descending list of n items, assume a priority CRCW PRAM with n processors. Distribute the first item to processor 1, the second to processor 2 etc., while x is stored in the global memory. If a processor's item is $\geq x$, it tries to write its index to a designated location in the global memory. Since the one with the smallest index will succeed, the location now contains the desired position of x. The run time is independent of the input size[2], so the time-processor-product is $\Theta(n)$, missing optimal speed-up as sequential searching can be done in $O(\log n)$.

Sorting. Habermann [23] proposes a simple parallel sorting algorithm for a linear array of processors called Odd-Even Transposition Sort (OETS). Each processor holds one item. In an odd (even) round, all neighbouring pairs starting at an odd (even) index swap their items if they are out of order. The two types of rounds take turns for at most n rounds total when n items are to be sorted, yielding $O(n^2)$ operations when accounting for the n processors. Again, this is not optimal for comparison-based sorting, which may be done in $O(n \log n)$.

Strongly Connected Components. Fleischer et al. [24] propose a Divide-and-Conquer algorithm for computing strongly connected components (SCC) of a digraph, which they call DCSC. First, find all descendants and predecessors of an arbitrary node, e.g. by carrying out a breadth-first search (BFS) in the graph and its reversed version. The intersection of both sets constitutes a SCC. Observe how each further SCC has to be completely contained in either the descendants, the predecessors or the undiscovered nodes, such that the described routine may be called recursively for start nodes in each subset independently until each vertex is assigned to a SCC. They prove an expected serial time complexity of $O(n \log n)$ for graphs on n nodes whose degrees are bounded by a constant. This is not optimal, but parallelization of the two searches per vertex, as well as the recursive calls, may significantly speed up execution.

2.4 Analogy to Neural Networks

Loukas [11] formally establishes an analogy between models like processor arrays and GNN by identifying processors with graph nodes and communication channels with edges. Therefore, the width of a GNN corresponds to p, and its depth to t. Loukas coins the term *capacity* for the product of width and depth of a GNN, reflecting the time-processor product of parallel algorithms. The shared memory of a PRAM finds its neural analogue in graph-level features. Since the computation of a graph feature may take into account positional encodings of the nodes, we may assume a priority CRCW PRAM, encompassing all other PRAM models.

3 Efficiency of Executing Algorithms Neurally

Inspired by the definition of efficiency in parallel computing, we define the efficiency of a neural executioner as follows.

Definition 2. Let $\mathcal{G}$ be a GNN with capacity $c(n)$ executing an algorithm $\mathcal{A}$ of sequential complexity $C(n)$. Define its *node efficiency* as

$$\eta(\mathcal{G}, \mathcal{A}) \coloneqq \frac{C(n)}{c(n)}.$$

This definition implies an important assumption we make throughout this paper.

Assumption 1. When executing an algorithm on a GNN, one constant-time operation is to be executed per node per layer.

This is not entirely unproblematic as discussed in section 6, but often expected when providing hints and helps to identify theoretical properties. Under this assumption, node efficiency denotes the share

[2]Distributing values to processors can be done in constant time by routing over the shared memory. We neglect distributing/returning in-/outputs from/to a host computer in the following as it is omitted in neural execution.

of nodes doing useful computations throughout the layers. Since the computational cost of a GNN also scales with the number of messages that are being sent, it is insightful to study the share of edges that transport relevant information as well.

Definition 3. Let $\mathcal{G}$ be a GNN operating over a graph $G = (V, E)$, $m \coloneqq |E|$, to execute an algorithm $\mathcal{A}$. Then we call an edge $(i, j) \in E$ *active* at layer t for a certain input x, if the operation to be executed by node j at time t involves information stored at node i at time $t - 1$. Let $a(t)$ be the number of active edges at time t, and T the total number of time-steps. Then define *edge efficiency* as worst case share of active edges when processing inputs x_n of size n,

$$\epsilon(\mathcal{G}, \mathcal{A}) \coloneqq \min_{x_n} \frac{1}{T} \sum_{t=1}^{T} \frac{a(t)}{m}.$$

Note how neural efficiencies are defined relative to the *algorithm* they are executing as opposed to the *task* they solve. This allows for a neural executioner to be efficient in executing an algorithm that is itself not efficient in solving a task.

3.1 Parallel Algorithms Entail Higher Efficiency

Contradicting a GNN's parallel nature by teaching it to execute sequential algorithms artificially impedes the task. Training to solve tasks in parallel instead is more efficient, which may also simplify the function to learn.

Shorter Trajectories. As observed by Loukas [11], the complexity of an algorithm lower bounds the capacity of a GNN executing it. If the number of processors is one, the depth alone needs to match the complexity, while the width might theoretically be set to one. But in practice, the width has to scale with the input size n to ensure applicability to different n. Therefore, *training sequential algorithms forces overspending on capacity by a factor of* n.

Setting the width to n, as is often done to distribute one unit of information over each node, entails n available processors. Making use of them may shorten the trajectory of an algorithm by a factor of up to n in the case of optimal speedup, which allows the capacity to take on its lower bound. The capacity of a GNN directly translates to the time needed to train and execute it. Additionally, long roll-outs give rise to an issue Bansal et al. [25] refer to as *overthinking*, where many iterations degenerate the behaviour of a recurrent processor.

Less Redundancy. Neural efficiencies denote the share of nodes and edges involved in useful computations. Redundant computations not only harm run times but may also interfere with the algorithmic trajectory. Parameterising them correctly to prevent this can complicate the function to learn. Assuming the redundant nodes (grey in figure 3(b)) need to preserve their information to be processed or put out later, their self-edges should execute an identity, while the additional incoming messages need to be ignored, i.e. mapped to a constant. In practice, this will be hard to do, which could entail a temporal variant of oversmoothing, where relevant information gets lost throughout the layers [26]. Oyedotun et al. [26] highlight how skip connections help to avoid the issue, Ibarz et al. [7] introduce a gating mechanism to leave information unchanged, Bansal et al. [25] let their architecture recall the original input.

So let's explore the efficiency of executing sequential and parallel algorithms.

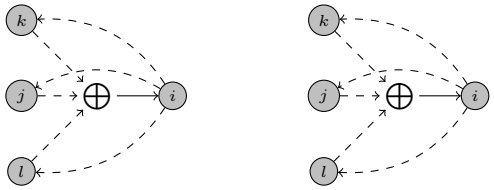
Corollary 1. Let $\mathcal{G}$ be a scalable GNN operating over a graph with n nodes and m edges. Further let $\mathcal{S}$ be a sequential, and $\mathcal{P}$ an efficient parallel algorithm on n processors, both of complexity C. Then executing $\mathcal{S}$ and $\mathcal{P}$ on $\mathcal{G}$, respectively, entails efficiencies

$$\eta(\mathcal{G}, \mathcal{S}) = O\left(\frac{1}{n}\right), \qquad \epsilon(\mathcal{G}, \mathcal{S}) = O\left(\frac{1}{m}\right),$$
$$\eta(\mathcal{G}, \mathcal{P}) = O(1), \qquad \epsilon(\mathcal{G}, \mathcal{P}) = O\left(\frac{n}{m}\right).$$

Proof. As observed above, the capacity c of a GNN executing a sequential algorithm of complexity C has to be $c \geq nC$, while it may be $c = C$ in the case of optimal speedup. Node efficiencies follow

(a) A single processor P may arbitrarily access any memory unit at any time, so there is no need to adhere to any pattern in which cells exchange information.

(b) In a parallel processing array, communication may look different at different times, but is always limited to the hard-wired channels ($J_i^t = \{k, l\}$, $J_i^{\tilde{t}} = \{j, l\}$).



(c) Permutation invariant recurrent neural processing is not adapted to node indices or time steps by default. Dashed edges indicate the influence of g.

Figure 2: Local view on information flow in different computational models at two different time steps t and $\tilde{t}$.

immediately. Since one processor can read only so much information, only a constant number of edges can be active at each layer during sequential processing, while up to a multiple of n edges can be active during parallel algorithms. This yields the stated edge efficiencies. □

Therefore, the share of nodes avoiding redundant computation cannot exceed $1/n$ when executing sequential algorithms, whereas it may reach up to 1 for efficient parallel algorithms. At the same time, the number of redundant messages is reduced by a factor of n. Removing the artificial bottleneck of a single processor prevents data from having to be stored until the processor gets to it. Allowing nodes to carry out meaningful computation frees them of the dead weight of acting as memory.

Local Exchange of Information. In neural networks, information exchange is inherently local. The feature h_i^t of node i at time t may only depend on itself and its neighbours $\mathcal{N}_i$. E.g. for permutation invariant MPNN [27],

$$h_i^t = f\big(h_i^{t-1}, \bigoplus_{j \in \mathcal{N}_i} g(h_i^{t-1}, h_j^{t-1})\big) \tag{1}$$

This paradigm is often not respected by classical algorithms, as depicted in figure 2(a). In the RAM model, the state $h_{i_t}^t$ of register i_t updated at time t may depend on any two registers j_t and k_t:

$$h_{i_t}^t = f_i^t(h_{k_t}^{t-1}, h_{j_t}^{t-1}),\ j_t, k_t \text{ arbitrary.} \tag{2}$$

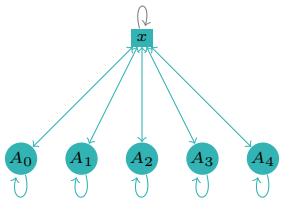
Not being able to restrict which nodes have to communicate may render it advisable for a GNN to operate over a complete graph to make sure all necessary information is available at all times (see e.g. [7]). The situation is different in the setting of interconnected processing arrays, see figure 2(b). For example, OETS only ever requires neighbouring processors to compare their items. In general, at time t, the memory state h_i^t of processor i is computed by

$$h_i^t = f_i^t(h_i^{t-1}, \mathop{\Big\|}_{j \in J_i^t} h_j^{t-1}),\ J_i^t \subseteq \mathcal{N}_i, \tag{3}$$

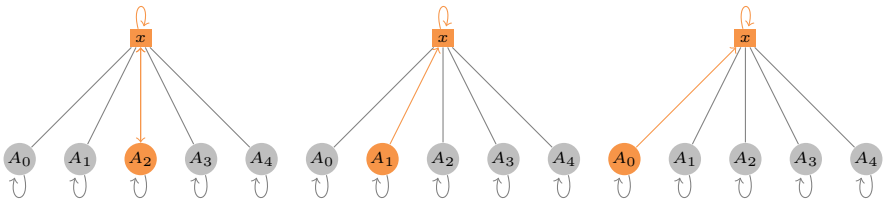
where concatenation indicates how i may tell apart its neighbours. Therefore it suffices for the GNN to only rely on edges present in the interconnection graph. To emulate a PRAM algorithm, an empty graph would in principle be enough, though it might not deem advantageous to route all communication over the graph feature in practice. Restricting the number of edges further reduces the use of resources and may help performance, since fewer unnecessary messages are being passed. Interconnection graphs are mostly chosen to be sparse, enabling maximum edge efficiency.

Table 1: Worst case asymptotic capacity c, node efficiency η and edge efficiency ϵ in our experiments.

	Searching		**Sorting**		**SCC**	
	Seq.	Par.	Seq.	Par.	Seq.	Par.
c	$n \log n$	n	n^3	n^2	$n(n+m)$	n^3
η	n^{-1}	1	n^{-1}	1	n^{-1}	n^{-1}
ϵ	n^{-2}	1	n^{-2}	n^{-1}	m^{-1}	m^{-1}



(a) Parallel Search



(b) Binary Search when $rank_A(x) = 1$

Figure 3: Necessary information flow when searching x in $A = [A_0, \ldots, A_4]$ using different algorithms. Active nodes and edges in color.

4 Methodology

For our experiments, we use the CLRS framework for neural algorithmic reasoning [3]. The default hidden size is 128, but we include experiments with smaller sizes in appendix A. The train data samples have input sizes 4, 7, 11, 13 and 16, while testing is done on input size $n = 64$.[3]

4.1 The CLRS Framework

CLRS follows the encode-process-decode paradigm. After encoding the input, a recurrent GNN denoted as *processor* network carries out the main computation, until finally its output is routed through the decoder network. To help performance and distinguish between different algorithms solving the same task, not only the final output is evaluated, but also the intermediate states. The ground truths of these are referred to as *hints*. For further details, we kindly refer the reader to [3].

4.2 Considered Algorithms

To test the hypothesis, we consider the two elementary tasks of searching and sorting, as well as computing SCC as an example of a graph algorithm. The parallel algorithms are chosen from section 2.3; as sequential counterparts, we use binary search, bubble sort and Kosaraju's SSC algorithm from the CLRS-30 benchmark [3]. Key data of the GNN we use are listed in table 1. We compare performances across various processor networks, namely the wide-spread architectures of DeepSets [28], GAT [29], MPNN [27], and PGN [20]. The trajectories of the new algorithms are encoded for the CLRS framework as follows below. Note that in every case, randomized positional information, as proposed by Mahdavi et al. [22] and standard on CLRS, is provided as part of the input, to emulate the situation of uniquely identified processors.

[3]Earlier versions of this paper report performance when training only on samples of size 16.

4.2.1 Searching

Parallel Search. The hints for parallel search of x in A closely resemble its template. As to be seen in figure 3(a), each item A_i of A is represented by one node of an empty graph. A node `mask` indicates whether $A_i \leq x$. The position $rank_A(x)$ of x in A is predicted by the graph feature as a categorical variable over the nodes (`pointer` in [3]). Therefore we introduce an extra node carrying x as a placeholder to allow for as many categories as possible positions of x.

To perfectly predict the outcome in this setting, the graph nodes may be updated by

$$h_i = ReLU(A_i - x),$$

yielding $h_i = 0$ if and only if $A_i \leq x$.

So the graph feature may be computed by

$$rank_A(x) = \min\{i = 1, \ldots, n : h_i = 0\}.$$

These steps closely align with the considered neural update functions, especially since the function updating the graph level possesses its own set of parameters. Additionally, the roll-out has a constant length, leaving room for only a constant number of redundant edges, see figure 3(a) and table 1. Altogether, we expect high performance on parallel search.

Binary Search. Opposed to parallel search, binary search has an optimal complexity of $O(\log n)$. But given the need for n nodes, it still requires an enhanced capacity of $O(n \log n)$, yielding low node efficiency. In CLRS-30, binary search is executed on a complete graph (whose edges are omitted in figure 3(b) to avoid clutter), impairing edge efficiency, see table 1. Low efficiency is visible in figure 3(b) by the amount of grey components.

4.2.2 Sorting

OETS. Swapping the `scalar` items would require making numerical predictions. Instead, we predict changing predecessors as `pointers`, following preimplemented examples. To still provide edges between nodes holding items to compare, we have to operate on a complete graph, sacrificing edge efficiency (see table 1), since only $\Theta(n)$ edges are active in each round, so $\epsilon = n/n^2$. As hints, we feed for each round the current predecessors along with an edge `mask` indicating whether two nodes have to switch their role, and a graph-level `mask` with the parity of the round, serving as a rudimentary clock.

Bubble Sort. Though Bubble Sort induces the same amount of operations $O(n^2)$ as OETS, it requires a larger network to be executed on (table 1). Again, along with operating over a complete graph, this entails low efficiencies.

4.2.3 Strongly Connected Components

DCSC. We input the undirected adjacency matrix as edge `mask`, along with the directed one as `scalar`. Parallelizing the recursive calls of DCSC on multiple disjoint sets would require an extra feature dimension for every search that is going on. Therefore we only let the two BFS starting from the same source node be executed in parallel, which we each encode as is standard in CLRS-30. Additionally, a binary `mask` on each node is flipped to 1 as soon as it is discovered from both directions, indicating it belongs to the currently constructed SCC (this is reset at the start of every new search). At the same time, it receives a `pointer` to the source, which in the end constitutes the output. Throughout, we keep track of undiscovered nodes in another node `mask`. We choose the node with the smallest index from this set as the next source.

DCSC spends most of its time on the repeated BFS, a subroutine known to be learned well even on relatively simple architectures [16], as it aligns well with neural execution [30]. Note how they let each node consider all its incoming edges in parallel, as is done on CLRS-30. This not only allows the trajectory to be shortened from $O(n+m)$ to $O(n)$ but also prevents redundant computations from having to be handled explicitly. Except for the source, each node can carry out the same computation at each step (see [16] for details) – just that this will only change its state whenever information flowing from the start node reaches it. DCSC only has to pass the index s of the source node instead of computing predecessor pointers, so computation looks like depicted in figure 4, closely resembling

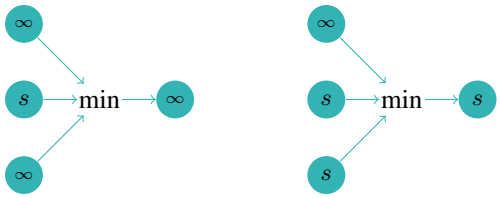


Figure 4: Consecutive steps of passing the source node index s during a BFS of DCSC. Note how repeating the computation would not change the state of the rightmost node, so redundant computations do not require to be parameterised differently.

Table 2: Out-of-distribution micro-F1 scores after 2000 iterations of training sequential versus parallel algorithms on different processor networks, averaged over 3 seeds.

	Searching		**Sorting**		**SCC**	
Arch.	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel
DeepSets	67.2%±10.2	**100%± 0.0**	57.5%±4.5	**77.8%±5.3**	26.2%±8.7	**41.1%±14.4**
GAT	3.4%±1.2	**100%±0.0**	28.6%± 13.6	**34.3%±20.0**	28.9%±2.5	**76.6%±4.9**
MPNN	79.8%±5.8	**100%± 0.0**	34.5 %±9.3	**52.4%±23.4**	34.0%±1.1	**75.0%±6.2**
PGN	77.2%±9.2	**100%± 0.0**	50.9 %±17.8	**62.0%±25.9**	35.1%±3.1	**82.3%±2.2**

the situation in figure 2(c). Therefore, efficiency is expected to be less important for predictive performance in this special case. An obvious upper bound to DCSC's run time is $O(n^2)$, accounting for one (two-sided) BFS per node, resulting in the big capacity reported in table 1. There is also no guarantee for more than one node and edge being active per step per BFS, resulting in low efficiencies. But this represents edge cases at best, such that the average trajectories will be much shorter and more efficient, as experiments will show. The core of DCSC aligning so well with neural execution promises good results.

Kosaraju. The skeleton of Kosaraju's algorithm as implemented in CLRS-30, on the other hand, is formed by a depth-first search (DFS), which is more challenging for neural executioners [3]. As opposed to the closely related BFS, it is hard to parallelize. When relying on lexicographic ordering for tie-braking, it is considered an *inherently sequential* algorithm [31]. Since nodes have to wait for the search to retract from its siblings, the computation cannot be carried out as in figure 4, so processing needs to be timed correctly. The total run time is $O(n + m)$, entailing the capacity and efficiencies reported in table 1.

5 Results

Predictive performance is reported in table 2. Parallel search achieves perfect results and converges very quickly (see figure 6). Meanwhile, training time on sample size 16 is reduced by a factor of more than 2 as compared to binary search (see figure 5). Parallel sorting outperforms its sequential opponent as well, though performance on both is subject to big standard deviations. Despite both

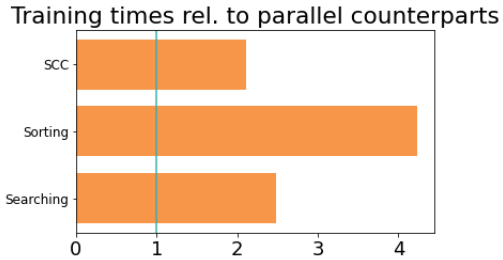
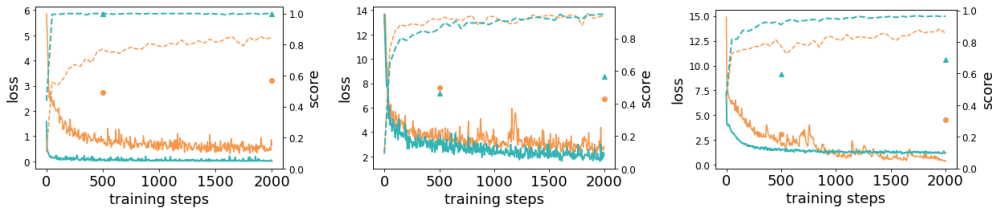


Figure 5: Training times of sequential algorithms with samples of input size $n = 16$, relative to their respective parallel counterparts.



(a) Searching (b) Sorting (c) SCC

Figure 6: Losses (solid lines) and validation scores (dashed lines) over time on different tasks. Performance on sequential algorithms in orange, on parallel ones slightly thicker in turquoise. Test scores after 500 and 2000 steps as orange points and turquoise triangles for sequential and parallel algorithms, respectively.

algorithms requiring the same asymptotic number of operations, training OETS takes less than a quarter of the time needed for bubble sort (figure 5. Despite DCSC's only partial parallelization and the asymptotically optimal linear run time of its sequential opponent, training time is more than halved for the SCC task as well. At the same time, predictions become up to more than twice as accurate.

6 Discussion

Neural efficiency only loosely correlates with predictive performance when comparing tables 1 and 2. This is not too surprising, since correctly parameterising redundant computations is only one of many aspects that make a function hard to learn. We propose a rather one-sided relationship, where low efficiencies can harm accuracy (if not circumvented as in BFS, see section 4.2.3), but high efficiencies do not necessarily enhance learning success.

We would like to highlight the importance of taking the perspective on neural networks as computational models when executing algorithms, as it opens access to the rich theory of computational complexity. E.g. the classes of NC (efficiently parallelizable) and P-complete problems (mostly thought of as inherently sequential) [18] inform us on which tasks may be hard to execute neurally, to tackle them more effectively. However, in doing so, it is important to keep in mind the gap between the respective sets of constant time operations, with none being strictly more powerful than the other. On the one hand, a single RAM instruction may need to be approximated by entire subnetworks. On the other hand, one neural step suffices to process all incoming edges of a node during the execution of BFS [16]. This breaks up the strict correspondence between time-processor product and capacity.

7 Conclusion

As suggested in section 3.1, parallel algorithms prove to be a lot more efficient to learn and execute on neural architectures than sequential ones. Often, OOD predictions on algorithmic tasks are significantly improved as well, suggesting that higher node and edge efficiency can help learning. Future work has to show how performance is impacted for other tasks, on more elaborate architectures like in [7, 8], and in generalist settings.

Author Contributions

Valerie Engelmayer:	Conceptualization, Formal Analysis, Investigation, Methodology, Software, Visualization, Writing – original draft
Dobrik Georgiev:	Resources, Validation
Petar Veličković:	Supervision, Writing – review & editing

Acknowledgements

We would like to thank Razvan Pascanu and Karl Tuyls for their valuable comments, as well as Pietro Liò for insightful discussions and Torben Hagerup for the support he provided.

References

[1] Petar Veličković and Charles Blundell. Neural Algorithmic Reasoning. *Patterns*, 2(7):100273, July 2021. ISSN 26663899. doi: 10.1016/j.patter.2021.100273. URL http://arxiv.org/abs/2105.02761. arXiv:2105.02761 [cs, math, stat]. 1

[2] Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. Neural Execution Engines: Learning to Execute Subroutines. Technical Report arXiv:2006.08084, arXiv, October 2020. URL http://arxiv.org/abs/2006.08084. arXiv:2006.08084 [cs, stat] type: article.

[3] Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The clrs algorithmic reasoning benchmark. In *International Conference on Machine Learning*, pages 22084–22102. PMLR, 2022. 1, 6, 7, 8

[4] Lovro Vrček, Petar Veličković, and Mile Šikić. A step towards neural genome assembly. *arXiv preprint arXiv:2011.05013*, 2020. 1

[5] Petar Veličković, Matko Bošnjak, Thomas Kipf, Alexander Lerchner, Raia Hadsell, Razvan Pascanu, and Charles Blundell. Reasoning-modulated representations. In *Learning on Graphs Conference*, pages 50–1. PMLR, 2022.

[6] Dobrik Georgiev, Ramon Vinas, Sam Considine, Bianca Dumitrascu, and Pietro Lio. Narti: Neural algorithmic reasoning for trajectory inference. 1

[7] Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyriacos Nikiforou, Mehdi Bennani, Róbert Csordás, Andrew Joseph Dudzik, Matko Bošnjak, Alex Vitvitskyi, Yulia Rubanova, et al. A generalist neural algorithmic learner. In *Learning on Graphs Conference*, pages 2–1. PMLR, 2022. 1, 2, 4, 5, 9

[8] Beatrice Bevilacqua, Kyriacos Nikiforou, Borja Ibarz, Ioana Bica, Michela Paganini, Charles Blundell, Jovana Mitrovic, and Petar Veličković. Neural Algorithmic Reasoning with Causal Regularisation, February 2023. URL http://arxiv.org/abs/2302.10258. arXiv:2302.10258 [cs, stat]. 1, 9

[9] Qianru Zhang, Meng Zhang, Tinghuan Chen, Zhifei Sun, Yuzhe Ma, and Bei Yu. Recent advances in convolutional neural network acceleration. *Neurocomputing*, 323:37–51, 2019. 1

[10] Amir Yazdanbakhsh, Kiran Seshadri, Berkin Akin, James Laudon, and Ravi Narayanaswami. An evaluation of edge tpu accelerators for convolutional neural networks. *arXiv e-prints*, pages arXiv–2102, 2021. 1

[11] Andreas Loukas. What graph neural networks cannot learn: depth vs width, January 2020. URL http://arxiv.org/abs/1907.03199. arXiv:1907.03199 [cs, stat]. 1, 3, 4

[12] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015. 1

[13] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Differentiable sorting networks for scalable sorting and ranking supervision. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8546–8555. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/petersen21a.html. 1

[14] Karlis Freivalds, Emīls Ozoliņš, and Agris Šostaks. Neural shuffle-exchange networks - sequence processing in o(n log n) time. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/9001ca429212011f4a4fda6c778cc318-Paper.pdf. 1

[15] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What Can Neural Networks Reason About? Technical Report arXiv:1905.13211, arXiv, February 2020. URL http://arxiv.org/abs/1905.13211. arXiv:1905.13211 [cs, stat] type: article. 1, 2

[16] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural Execution of Graph Algorithms, January 2020. URL http://arxiv.org/abs/1910.10593. arXiv:1910.10593 [cs, stat]. 1, 7, 9

[17] Alan Gibbons and Wojciech Rytter. *Efficient parallel algorithms*. Cambridge Univ. Press, Cambridge, reprinted edition, 1990. ISBN 978-0-521-38841-2 978-0-521-34585-9. 1, 2

[18] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, New York, 1995. ISBN 978-0-19-508591-4. 9

[19] Behrooz Parhami. *Introduction to parallel processing: algorithms and architectures*. Springer Science & Business Media, 2006. 1

[20] Petar Veličković, Lars Buesing, Matthew Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 2232–2244. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/176bf6219855a6eb1f3a30903e34b6fb-Paper.pdf. 2, 6

[21] Dobrik Georgiev, Pietro Barbiero, Dmitry Kazhdan, Petar Veličković, and Pietro Lió. Algorithmic Concept-Based Explainable Reasoning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(6):6685–6693, June 2022. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v36i6.20623. URL https://ojs.aaai.org/index.php/AAAI/article/view/20623.

[22] Sadegh Mahdavi, Kevin Swersky, Thomas Kipf, Milad Hashemi, Christos Thrampoulidis, and Renjie Liao. Towards Better Out-of-Distribution Generalization of Neural Algorithmic Reasoning Tasks, March 2023. URL http://arxiv.org/abs/2211.00692. arXiv:2211.00692 [cs]. 2, 6

[23] A Nico Habermann. Parallel neighbor-sort (or the glory of the induction principle). 1972. 3

[24] Lisa K. Fleischer, Bruce Hendrickson, and Ali Pınar. On Identifying Strongly Connected Components in Parallel. In José Rolim, editor, *Parallel and Distributed Processing*, volume 1800, pages 505–511. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-67442-9 978-3-540-45591-2. doi: 10.1007/3-540-45591-4_68. URL http://link.springer.com/10.1007/3-540-45591-4_68. Series Title: Lecture Notes in Computer Science. 3

[25] Arpit Bansal, Avi Schwarzschild, Eitan Borgnia, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. End-to-end algorithm synthesis with recurrent networks: Extrapolation without overthinking. *Advances in Neural Information Processing Systems*, 35:20232–20242, 2022. 4

[26] Oyebade K. Oyedotun, Kassem Al Ismaeil, and Djamila Aouada. Why is everyone training very deep neural network with skip connections? *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–15, 2022. doi: 10.1109/TNNLS.2021.3131813. 4

[27] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017. 5, 6

[28] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/f22e4747da1aa27e363d86d40ff442fe-Paper.pdf. 6

[29] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks, February 2018. URL http://arxiv.org/abs/1710.10903. arXiv:1710.10903 [cs, stat]. 6

[30] Andrew Dudzik and Petar Veličković. Graph Neural Networks are Dynamic Programmers. Technical Report arXiv:2203.15544, arXiv, June 2022. URL http://arxiv.org/abs/2203.15544. arXiv:2203.15544 [cs, math, stat] type: article. 7

[31] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, June 1985. ISSN 00200190. doi: 10.1016/0020-0190(85)90024-9. URL https://linkinghub.elsevier.com/retrieve/pii/0020019085900249. 8

Table 3: Out-of-distribution micro-F1 scores after 2000 iterations of training sequential versus parallel algorithms on different processor networks with hidden size 8, averaged over 3 seeds.

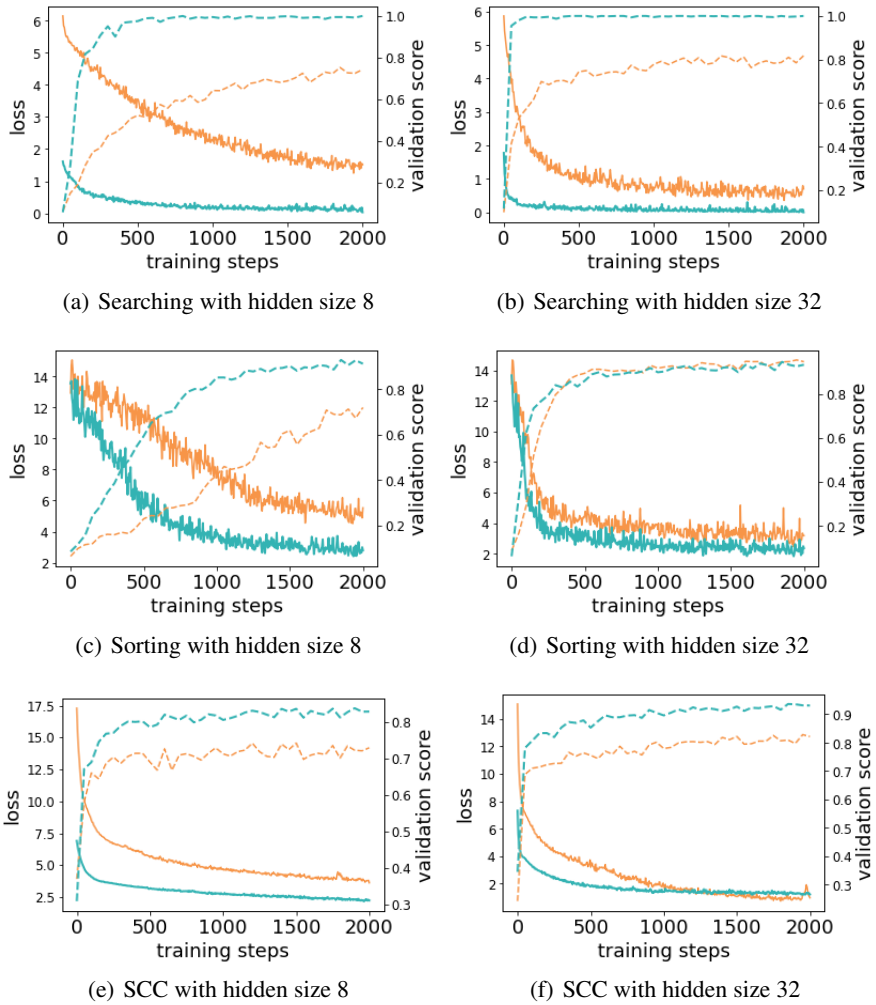
	Searching		**Sorting**		**SCC**	
Arch.	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel
DeepSets	40.8%±9.3	**97.9%± 2.9**	18.8%±7.2	**43.4%±6.8**	30.5%±2.5	**43.4%±7.2**
GAT	4.0%±0.7	**100%±0.0**	29.9%± 11.1	**36.0 %±16.3**	33.5%±0.7	**48.2%±2.8**
MPNN	31.9%±13.6	**99.0%± 1.5**	**40.8 %±24.5**	30.6%±18.5	29.2%±4.3	**46.2%±2.2**
PGN	36.4%±20.0	**98.0%± 2.9**	41.4 %±25.8	**51.0%±15.6**	30.7%±3.1	**45.9%±3.1**

Table 4: Out-of-distribution micro-F1 scores after 2000 iterations of training sequential versus parallel algorithms on different processor networks with hidden size 32, averaged over 3 seeds.

	Searching		**Sorting**		**SCC**	
Arch.	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel
DeepSets	65.1%±7.5	**100%± 0.0**	37.0%±3.2	**64.3%±2.1**	15.2%±4.9	**45.0%±5.5**
GAT	7.5%±1.6	**100%±0.0**	43.6%± 9.0	**64.6%±7.2**	14.8%±12.0	**51.1%±6.1**
MPNN	63.1%±10.9	**100%± 0.0**	57.6 %±4.8	**60.2%±9.4**	26.2%±4.2	**62.7%±6.9**
PGN	77.1%±2.5	**100%± 0.0**	44.0 %±8.3	**74.1%±5.6**	21.4%±4.3	**66.9%±5.9**

A Appendix

Better alignment of parallel algorithms may enhance performance on smaller processor networks. Indeed, we observe that decreasing the hidden size from 128 to 32 or even 8 is mostly slightly less impeding in the parallel setting, especially the searching task, see tables 3 and 4, along with figure 7.



(a) Searching with hidden size 8

(b) Searching with hidden size 32

(c) Sorting with hidden size 8

(d) Sorting with hidden size 32

(e) SCC with hidden size 8

(f) SCC with hidden size 32

Figure 7: Losses (solid lines) and validation scores (dashed lines) over time on different tasks with different hidden sizes. Performance on sequential algorithms in orange, on parallel ones slightly thicker in turquoise.