Red-Blue Pebbling with Multiple Processors: Time, Communication and Memory Trade-offs

Toni Böhnlein[0009−0001−2152−022X]✉, Pál András Papp[0009−0005−6667−802X]✉, and Albert-Jan N. Yzelman[0000−0001−8842−3689]

Computing Systems Lab, Huawei Zurich Research Center, Zurich, Switzerland
toni.boehnlein,pal.andras.papp,albertjan.yzelman@huawei.com

Abstract. The well-studied red-blue pebble game models the execution of an arbitrary computational DAG by a single processor over a two-level memory hierarchy. We present a natural generalization to a multiprocessor setting where each processor has its own limited fast memory, and all processors share unlimited slow memory. To our knowledge, this is the first thorough study that combines pebbling and DAG scheduling problems, capturing the computation of general workloads on multiple processors with memory constraints and communication costs. Our pebbling model enables us to analyze trade-offs between workload balancing, communication and memory limitations, and it captures real-world factors such as superlinear speedups due to parallelization.

Our results include upper and lower bounds on the pebbling cost, an analysis of a greedy pebbling strategy, and an extension of NP-hardness results for specific DAG classes from simpler models. For our main technical contribution, we show two inapproximability results that already hold for the long-standing problem of standard red-blue pebbling: (i) the optimal I/O cost cannot be approximated to any finite factor, and (ii) the optimal total cost (I/O+computation) can only be approximated to a limited constant factor, i.e., it does not allow for a polynomial-time approximation scheme. These results also carry over naturally to our multiprocessor pebbling model.

Keywords: Red-blue pebble game · Scheduling · Parallel computing · Limited memory · Approximation

1 Introduction

The computational requirements of modern applications, ranging from scientific simulations to artificial intelligence, necessitate parallel data processing. However, developing parallel algorithms that efficiently divide (sub)tasks, manage memory, and organize communication between processors presents significant difficulties to computer scientists. One particular bottleneck for the performance of parallel computations is due to data locality, i.e., the memory of modern hardware features a hierarchical structure, and to process data, it has to be stored in the lowest layer. Since the layers' memory is limited in size, data movements between them (called I/O operations) are necessary and impact the execution time of computations significantly. The phenomenon becomes especially noticeable when dealing with tasks involving basic operations on large amounts of input data, e.g., the training of neural networks.

In this paper, we introduce a model offering improved insights into the challenges regarding I/O costs and limited memory in the context of parallel computing. Our focus is directed toward the efficient execution of a specific computation by several processors (rather than the design of a parallel algorithm tailored for a particular problem). We consider directed acyclic graphs (DAGs) as the model for a computation. The nodes correspond to single operations, while the directed edges express that the output data of a node is required as input for another node, enforcing precedence constraints on their order of execution. We are concerned with devising effective and efficient schedules for a given DAG and number of processors with limited working memory.

Hong and Kung [21] introduced the red-blue pebble game to study the I/O complexity of computations when executed by a *single* processor with a two-level memory hierarchy. The processor has *fast memory* (working memory) of limited capacity. Computing a node (and storing its output data in fast memory) requires that its predecessors' output data is stored in fast memory. To free up space in fast memory, I/O operations can transfer data to (and back from) *slow memory*, which has unlimited capacity.

In this model, available data is indicated by placing a *pebble* on the corresponding node. Data stored in fast memory is represented by a red pebble, and data in slow memory by a blue pebble. The game starts with an empty DAG. A red pebble can be placed on a node (i.e., computing the node) if all its predecessors have red pebbles on them. A red pebble can be placed on a node if it already has a blue pebble, and vice versa (I/O operations). Any pebble can be removed for free. The goal is to place pebbles on the sinks (outputs of the computation) while minimizing the number of I/O operations. The number of red pebbles used may not exceed a given fast memory capacity at any time.

Red-blue pebbling studies trade-offs between I/O costs and memory size. In our paper, we extend this model to a setting where several processors compute in parallel, and communication is organized via shared memory; this allows us to study trade-offs between time, communication and memory size.

To familiarize ourselves with pebbling, consider the example DAG depicted in Figure 1. First, assume that we pebble it using only 3 red pebbles on a single

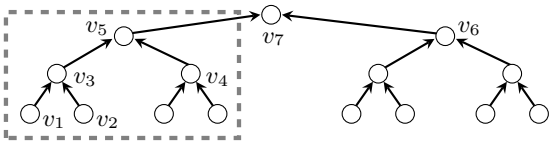

Fig. 1. A simple example DAG for pebbling.

processor. We may start by placing red pebbles on nodes v_1 and v_2, which then permits us to place a red pebble on node v_3. The red pebbles on v_1 and v_2 are no longer needed and can be removed. Observe that to pebble node v_4, we again need all 3 red pebbles. Since v_3 has an outgoing edge to node v_5, which we did not pebble yet, we place a blue pebble on v_3 (casting our first I/O operation), and then remove the red pebble from v_3. Next, v_4 (and its predecessors) can be pebbled analogously to v_3, leading to a configuration where we have a red pebble on v_4 and a blue pebble on v_3. We use another I/O operation to put a red pebble back on v_3, since it has a blue pebble. Now, we can place a red pebble on node v_5, and then remove the red pebbles from v_3 and v_4. Note that we pebbled the sub-graph in the dashed box with 2 I/O operations. Another I/O operation places a blue pebble on v_5. We pebble the subtree rooted v_6 in an identical way to the one rooted at v_5. To finish, we bring a red pebble back to v_5 with an I/O step, and then place a red pebble on v_7.

In contrast, consider pebbling the DAG with multiple processors for an intuitive description of our model. We use two processors p_1 and p_2 each having their own set of 3 red pebbles. To place a red pebble on a node, its predecessors require red pebbles of the *same* processor. We pebble the left and the right subtree of node v_7 with red pebbles of p_1 and p_2, respectively, using the strategy described above for a single processor. All steps are executed in parallel, reducing both the compute and I/O operations by a factor 2. The result is a configuration with a red pebble of p_1 and p_2 on nodes v_5 and v_6, respectively.

Recall that with one processor, we needed two I/O operations to convert the red pebble on v_5 to blue and then back to red. In contrast, with two processors we need I/O operations to ensure that we have red pebbles of the same processor on v_5 and v_6. The required communication between the processors is done via the shared memory (blue pebbles): We replace the red pebble of p_1 on v_5 by a blue pebble, and then the blue pebble by a red pebble of p_2. To finish the computation, we place a red pebble of p_2 on v_7.

Our Contribution. We present the multiprocessor red-blue pebbling game (MPP), which is a natural generalization of red-blue pebbling. MPP essentially combines the areas of DAG pebbling problems (single-processor computation with limited memory) and DAG scheduling problems (multiprocessor computation, but with unlimited memory). The result is a simple yet expressive model that captures computational costs, I/O costs incurred by memory limitations, and inter-processor communication costs. Unlike earlier attempts to generalize red-blue pebbling, MPP naturally combines these aspects into a single cost function,

allowing for a convenient study of the trade-offs between these factors. These trade-offs have been analyzed in detail for many concrete computations before (e.g., matrix multiplication [25,24,40]), but not for arbitrary DAGs.

We first discuss the basic properties of MPP, including simple bounds on the pebbling cost, NP-hardness for specific DAG classes, and the analysis of a simple greedy approach. We also analyze how adding more processors (with the same, or reduced amount of fast memory) can affect the pebbling problem, i.e., how the optimal pebbling cost can change in the same DAG.

Then as our main technical result, we present two hardness results for approximating the optimal strategy in red-blue pebbling problems. Both of these proofs already apply to single-processor red-blue pebbling, thus presenting novel results for a problem that has been studied for several decades. Firstly, we show that the minimal number of I/O steps in a pebbling cannot be approximated in polynomial time to any finite multiplicative factor, or any additive $n^{1-\varepsilon}$ term (for any $\varepsilon > 0$), unless P=NP. Secondly, we show that the minimal total cost (including both I/O and computation steps) can only be approximated to a limited factor, i.e., there is a $\delta > 0$ such that the optimum cannot be approximated to a $(1+\delta)$ factor in polynomial time, unless P=NP; in other words, the problem is APX-hard, allowing no PTAS. Both of these results carry over naturally to our multiprocessor pebbling model.

2 Related Work

Pebble games on graphs are used to capture various aspects of computation. For instance, the standard (black) pebble game [12,30] models general time-memory trade-offs, with the results including bounds on achievable trade-offs [22,26,32] and the complexity of computing pebbling strategies [18,13,36,2]. Other pebble games aim to capture different properties of computing, such as non-determinism or reversibility [3,11,15,6].

Hong and Kung [21] introduce the red-blue pebble game to study I/O complexity, and derive lower bounds based on a DAG partition technique. It is applied to specific computations in [17,31]. Further bounds on I/O costs were derived with different methods in [23,35,19]. Other works [33,34,16] extend the red-blue pebble game to single and multiple processors over memory hierarchies. The work of [24,25] refines the technique of [21] and derive improved I/O lower bounds for special DAGs. The bounds are extended to a multiprocessor setting where workload is perfectly balanced. In contrast to MPP, these works do not consider trade-offs between computation and I/O.

As for the complexity of computing optimal pebbling strategies, Demaine and Liu [14] show that standard red-blue pebbling is PSPACE-complete, and propose variations that are shown NP-complete. NP-hardness for red-blue pebbling with computation costs is also shown in [7]. The work of [28] shows an inapproximability to a $(2-\varepsilon)$ factor, assuming the unique games conjecture. The work of [9] presents a bi-criteria approximation for the optimal I/O cost to

a poly $\log(n)$ factor, provided that it is allowed to violate the memory bound by a poly $\log(n)$ factor.

Naturally, there are also several more realistic models that capture both parallelization and memory limitations, but these exhibit key differences from MPP. The work of [5,10] studies the cost components (computation, different kinds of I/O) separately, mostly assuming that I/O steps cannot be parallelized. Other works assume simpler models regarding computation costs: they are not considered at all in [4], and are handled with artificial balance constraints in [37,19]. Moreover, these models are primarily used for studying specific computations (e.g., Matrix multiplication, FFT) rather than arbitrary DAGs.

Valiant [39] introduces the bulk-synchronous parallel (BSP) computing model with explicit synchronisation, which was extended later [38,27] to include data locality and communication via shared memory. The model was further generalized to capture multi-processor/core architectures with multi-level memory [40]. These models describe practical compute systems more closely, and also match our MPP model. There are also theoretical studies on DAG scheduling in BSP [29], but without memory restrictions.

3 Model Definition

Our model for a computation is a directed acyclic graph (DAG) $G = (V, E)$. We use the notation $n = |V|$ for the number of nodes. The in/out-degree of a node is the number of its incoming/outgoing edges. We call nodes with in/out-degree 0 the source/sink nodes. Let Δ_{in} denote the highest in-degree in our DAG, i.e., the highest number of inputs to an operation in our computation; several previous works on pebbling assume that Δ_{in} is small, e.g., a constant [14,28]. Moreover, we denote the set of positive integers by $\mathbb{Z}^+$, and use the notation $[a] = \{1, ..., a\}$, for $a \in \mathbb{Z}^+$. Our goal is to execute a computational DAG on k processors, each having limited fast memory of size r, for parameters $k, r \in \mathbb{Z}^+$.

3.1 Single Processor Red-Blue Pebbling

Before introducing our parallel model, we briefly recap the single-processor red-blue pebble game (SPP) [21], where we have a single processor with fast memory of size r and unlimited slow memory. Nodes with red pebbles on them (denoted $R \subseteq V$) and nodes with blue pebbles on them (denoted $B \subseteq V$) correspond to the output data that is currently saved in fast and slow memory, respectively. In order to pebble a DAG, the following rules can be applied:

(R1-S) Place a red pebble on a node that has a blue pebble,
(R2-S) Place a blue pebble on a node that has a red pebble,
(R3-S) Place a red pebble on a node if all its predecessors have red pebbles,
(R4-S) Remove a (red or blue) pebble.

The game starts with an empty DAG; rule (R3-S) allows us to place red pebbles on source nodes. A pebbling strategy is a sequence of the rules which

places pebbles on the sink nodes in the end, while the number of red pebbles does not exceed r at any step. The I/O costs are the total number I/O operations, i.e., applications of rules (R1-S) or (R2-S). The goal is to pebble a DAG with minimum I/O costs. Most works on SPP analyze lower bounds on I/O cost for a given DAG and memory limit r.

Additional variations of SPP have also been proposed to simplify SPP, make it more realistic, or resolve the fact that this base SPP variant is not even in NP, but rather PSPACE-complete. This is because the optimal pebbling sequence can be super-polynomially long in extreme cases, since repeatedly deleting and recomputing the same node incurs no cost. Notable SPP variants include:

- *One-shot SPP:* (R3-S) can be applied only once for each node [9],
- *No-deletion SPP:* (R4-S) is not allowed [14],
- *SPP with computation costs:* (R3-S) also incurs a small cost of ε [28,7].

The first two variants feature somewhat artificial restrictions to prohibit the deletion and then recomputation of a node entirely. On the other hand, SPP with computation costs is more realistic, discouraging this in a natural way by ensuring that computation steps also incur some cost, as in practice. This last SPP variant is also the closest one our multiprocessor pebbling model.

We note that besides this, there are also smaller aspects of the definition that vary over different previous works. For instance, some papers assume instead that source nodes already have a blue pebble initially, or that sink nodes specifically need to have a blue pebble in the end; this is also the case in the original work of [21]. For most proof constructions, these model variants can be reduced to each other with some simple tricks; we refer to [28] for a summary. Similarly, some works assume in rule (R1-S) that the new red pebble replaces the blue pebble, and in rule (R2-S) vice versa [28,19]; our SPP-related claims also carry over easily to this variant.

3.2 Multiple Processor Red-Blue Pebbling

We introduce the multiprocessor red-blue pebble game (MPP) which extends red-blue pebbling to a setting where k processors compute a DAG in parallel. Data stored in a processor's fast memory is represented by a red pebble of its *shade*, i.e., there are red pebbles of k different shades. The number of red pebbles of each shade is limited by r. The processors share unlimited slow memory that is used to (i) store data that cannot be kept in fast memory, and (ii) to communicate data between processors. Data available in the slow memory is represented by blue pebbles. Note that we assume that several pebbles of different shade/color can be placed on a node at the same time.

In our parallel version of the game, the rules allow multiple processors to simultaneously either transfer data between fast and slow memory, or compute nodes. More formally, define set $R^j \subseteq V$ as the set of *red pebbles* of *shade* j, for $j \in [k]$, and set $B \subseteq V$ as the set of *blue pebbles*. For $m \in \mathbb{Z}^+$ such that $m \leq k$, we call an injective function $f_m : [m] \to [k]$ a *shaded selection*, and extend the transition rules as follows:

(R1-M) For a shaded selection f_m and vertices $v_1, v_2, \ldots, v_m$ such that $v_i \in R^{f_m(i)}$, add v_i to B, for $i \in [m]$,
(R2-M) For a shaded selection f_m and vertices $\{v_1, v_2, \ldots, v_m\} \subseteq B$, add v_i to $R^{f_m(i)}$, for $i \in [m]$,
(R3-M) For a shaded selection f_m and vertices $v_1, v_2, \ldots, v_m$ such that for all predecessors u of v_i we have $u \in R^{f_m(i)}$, for $i \in [m]$, add v_i to $R^{f_m(i)}$, for $i \in [m]$,
(R4-M) Remove a (red or blue) pebble.

A parallel version of (R4-M) could also be defined; we use this simpler version since this rule will incur no cost. Note that any single rule places at most k pebbles, and that processors can be idle.

A *configuration* is a $(k+1)$-tuple $C_i = (R_i^1, R_i^2, \ldots, R_i^k, B_i) \subseteq V^{k+1}$ defining red and blue pebbles placed on the DAG. A configuration C_i is *valid* if $|R_i^j| \leq r$, for $j \in [k]$, i.e., it respects the fast memory size. An initial configuration C_0 sets $R_0^j = \emptyset$, for $j \in [k]$, and $B_0 = \emptyset$. Let $S \subseteq V$ be the sink nodes. We say configuration C_i is terminal, if $S \subseteq B_i \bigcup_{j=1}^{k} R_i^j$ holds. A *pebbling strategy* $(C_0, C_1, \ldots, C_T)$, for $T \in \mathbb{Z}^+$, is a sequence of valid configurations such that (i) C_i is obtained by applying a transition rule to C_{i-1}, for $i \in [T]$, and (ii) C_0 and C_T are initial and terminal, respectively. Equivalently, the pebbling strategy can be represented by a sequence of transition rules $(t_1, t_2, \ldots, t_T)$, where $t_i \in \{$(R1-M), (R2-M), (R3-M), (R4-M)$\}$ such that C_i is the result of applying t_i to C_{i-1}, for $i \in [T]$. To assign costs to a pebbling strategy, we assign costs to each rule. Let $g \in \mathbb{Z}^+$ be a parameter specifying the cost of an I/O step. We define the cost function for the transition rules as follows:

- $c(t_i) = g$ if $t_i =$ (R1-M) or (R2-M),
- $c(t_i) = 1$ if $t_i =$ (R3-M),
- $c(t_i) = 0$ if $t_i =$ (R4-M).

Then, the cost of a strategy $C(t_1, t_2, \ldots, t_T) = \sum_{i=1}^{T} c(t_i)$ is the total costs of its moves. Given a DAG G and parameters $k, r, g \in \mathbb{Z}^+$, the goal of MPP is to find a minimum cost pebbling strategy. We denote the cost of the optimum pebbling strategy by `OPT`.

3.3 Model Discussion

We first note that as in SPP, the rules of MPP allow for *recomputation*. Indeed, when I/O is expensive, it can sometimes be beneficial to compute the same node more than once; see Section 4 for an example.

Furthermore, the transition rules assume that the processors compute and access memory *synchronously*. This simplifies the analysis greatly, enabling us to formulate the cost function as a linear term of I/O and compute costs. However, in some practical settings, one processor may be computing while another one is accessing memory. This could be modelled by allowing each processor in a step to execute one of the SPP rules independently; however, assigning costs to a pebbling strategy then becomes an intricate matter. We expect that most

of the general reasoning about pebbling strategies also carries over to such an asynchronous setting; it has been shown that the improvements from a non-synchronous schedule are limited to a factor 2 [29]. Synchronization of communication is also natural in some hardware architectures, and several parallel programming models, like BSP [39], also feature synchronization steps.

While parallel computing models typically study trade-offs between computation time and communication, SPP studies trade-offs between I/O costs and memory size. Combining these in MPP allows us to study the three-fold trade-off between computation time, communication, and memory size. The I/O steps in MPP can happen either due to (i) communicating data between processors, or (ii) saving data to slow memory to free up space in fast memory.

We note that several models in previous works are closely related to MPP; we discuss these in detail in the full version of the paper [8]. For instance, [19] also outlines a generalization of SPP to multiple processors; however, in contrast to MPP, their work captures computation costs via an artificial balance constraint, which imposes heavy limitations on the model. We also note that with $r = \infty$ and minor adjustments, MPP also becomes equivalent to DAG scheduling in the BSP model [29]. This shows that with small variations, MPP is indeed a generalization of both SPP and DAG scheduling problems.

4 Fundamental Properties of MPP

Straightforward bounds. Similarly to SPP, if $r \leq \Delta_{in}$, then there can be no valid pebbling strategy, since a node of in-degree Δ_{in} requires $(\Delta_{in} + 1)$ red pebbles of the same shade to be computed: Δ_{in} on its in-neighbors, and one more on the node itself. Thus we always implicitly assume $r \geq \Delta_{in} + 1$.

However, if $r \geq \Delta_{in}+1$, there is indeed always a valid pebbling. Consider the nodes in any topological order. We can always select any processor p to compute the next node v, load all the (already computed) in-neighbors of v from slow memory to p (at a cost of at most $\Delta_{in} \cdot g$), compute the node on p (at a cost of 1), save the value of v to slow memory (at a cost of g), and then remove the red pebbles from v and its in-neighbors (for free). This strategy incurs a cost of at most $(\Delta_{in} + 1) \cdot g + 1$ for each node. On the other hand, since each (R3-M) step can compute at most k nodes, the number of compute steps is at least $\frac{n}{k}$. This shows the following simple bounds for the optimal cost.

Lemma 1. *For any instance of MPP, we have* $\frac{n}{k} \leq \texttt{OPT} \leq (g \cdot (\Delta_{in} + 1) + 1) \cdot n$.

A simple example gadget. We briefly discuss a simple example, the *zipper gadget*, which highlights many vital aspects of pebbling problems, and different variants of it are used throughout our proofs. We only describe the gadget and these properties here on a high level, leaving the details to the full version [8]. The gadget, shown in Figure 2, consists of two input groups S_1, S_2 of d nodes each, and a *main chain* of n_0 nodes, with the input groups having edges to the main chain nodes in an alternating fashion. We typically have $n_0 = n - O(1)$ to make S_1 and S_2 asymptotically irrelevant.

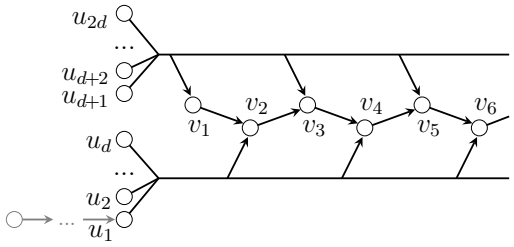

Fig. 2. Zipper gadget consisting of 2 input groups $S_1 = \{u_1, u_2, \ldots, u_d\}$ and $S_2 = \{u_{d+1}, \ldots, u_{2d}\}$, and a main chain $v_1, \ldots, v_{n_0}$. The edges going from the input groups are combined into a single arrow for simplicity. The extension to discourage recomputation is only illustrated for u_1 in gray.

- The gadget was used in [28] to analyze trade-offs in SPP. E.g. for $r = 2d+2$, we can always keep red pebbles on both S_1 and S_2, and use the last 2 red pebbles to compute the main chain without any I/O steps. However, for $r = d+2$, we still need 2 red pebbles to compute along the main chain, so we repeatedly need to move the other d red pebbles between S_1 and S_2, yielding a much higher cost.
- This example with $r = d+2$ also highlights the role of recomputation: we repeatedly need to move d red pebbles between S_1 and S_2, and doing this with I/O steps (repeatedly loading from slow memory) incurs a cost of $d \cdot g$ for each main chain node. However, since the nodes in S_1 and S_2 are sources, we can also simply compute them again with (R3-M) steps anytime, at a much lower cost of d per main chain node. To ensure that such a recomputation is suboptimal, we can also attach a chain of length $2g$ in front of each u_i: then recomputing u_i from a source node requires $2g + 1$ compute steps, whereas saving u_i to slow memory and loading it back later costs at most $2g$.
- Considering the gadget in MPP, if $k = 1$ and $r = d+2$, we again need to keep moving the d red pebbles between S_1 and S_2, which incurs a cost of $(d \cdot g + 1)$ per main chain node. However, with $k = 2$ processors and $r = d+2$, we can keep S_1 and S_2 in the fast memory of different processors, compute the main chain alternatingly, and only communicate these main chain nodes. This incurs a cost of $(2 \cdot g + 1)$ per chain node, and thus results in a superlinear speedup for larger d values.

NP-Hardness. Regarding the complexity of MPP, it is not surprising that the problem is NP-hard, since it generalizes SPP. However, MPP is already NP-hard on rather simple DAGs.

Lemma 2. *MPP is already NP-hard on the following subclasses of DAGs:*

- *2-layer DAGs (where the longest path has length* 1*),*
- *in-trees (where every out-degree is at most* 1*).*

The lemma follows from [29], where the same results are established for BSP scheduling; however, it requires some further work to adapt these proof constructions to MPP.

A Greedy Algorithm. It is also natural to wonder if we can obtain good solutions e.g. by following a greedy pebbling strategy. We can derive a naive upper bound on this approach, using the long known results that in DAG scheduling without memory limits or communication costs, any non-idle greedy strategy is a 2-approximation of the optimum [20]. Employing such a greedy approach for the computations, and assuming the worst-case strategy discussed for Lemma 1, we get the following bound.

Lemma 3. *Any pebbling where the compute steps follow a non-idle greedy schedule gives a* $2 \cdot (g \cdot (\Delta_{in} + 1) + 1)$*-factor approximation of the optimum.*

Unsurprisingly, greedy strategies can also return rather bad solutions. In general, it is even non-trivial to precisely define a greedy strategy for MPP, since pebbling consist of various aspects. Here we make observations for a general class of greedy strategies: we only assume that processor p always picks the next node to compute as the yet uncomputed node with the largest number (or largest fraction) of in-neighbors having a red pebble of p. We show a lower bound for any such greedy strategy, regardless of how compute steps are parallelized, how ties are broken, or how I/O steps are applied to compute the chosen node.

Lemma 4. *There exist DAGs where any such greedy pebbling algorithm is worse than the optimum*

- *by a* $\frac{1}{5} \cdot \Delta_{in} - 1$ *factor asymptotically (for any* $\Delta_{in} = O(1)$*),*
- *by a* $\frac{2}{3} \cdot g + 1$ *factor asymptotically (for any* $g \geq 2$*).*

Lower Bounds on Pebbling Costs. We now discuss how we can apply lower bounds on the number of required I/O steps from SPP to bound the optimum cost in MPP. The key insight is that a pebbling strategy for MPP can be implemented using a single processor (SPP) with fast memory of size $r \cdot k$. Specifically, each parallel rule can be simulated using k sequential rules.

Lemma 5. *Let* G *be a DAG such that an SPP pebbling strategy with fast memory of size* $k \cdot r$ *requires at least* L *steps of I/O, for some* $L \in \mathbb{Z}^+$*. Then, an MPP pebbling strategy with* k *processors, each having fast memory of size* r*, requires at least* L/k *steps of I/O.*

Corollary 1. *Let* G *be a DAG such that an SPP pebbling strategy with fast memory of size* $k \cdot r$ *requires at least* L *steps of I/O, for some* $L \in \mathbb{Z}^+$*. Then, an MPP strategy with* k *processors and fast memory of size* r *each has cost of at least* $g \cdot L/k + n/k$*.*

It follows that we can translate lower bounds for a single processor to k processors. Indeed, many previous works [21,25] derive I/O lower bounds for specific

computations in SPP, which are obtained utilizing a special DAG partition. For example, Hong and Kung [21] derive a lower bound of $\frac{n \log n}{\log(rk)}$ on the number of required I/Os in SPP (with fast memory size $r \cdot k$) for the n-point FFT DAG. This translates to a lower bound of $\frac{n}{k} \cdot (g \cdot \frac{\log n}{\log(rk)} + 1)$ on the cost in MPP for the same DAG. Another well-studied computation is matrix-matrix multiplication. Kwasniewski et al. [25] improve the technique of [21] and derive a lower bound of $\frac{2n^3}{\sqrt{rk}} + n^2$ in the single processor case, resulting in a lower bound of $\frac{n}{k} \cdot (g \cdot (\frac{2n^2}{\sqrt{rk}} + n) + 1)$ on the costs of matrix-matrix multiplication in MPP.

Finally, we show that there are instances where this bound is essentially tight.

Lemma 6. *For any n, there is a DAG construction with $\mathit{OPT} \leq g \cdot L/k + n/k + O(1)$ in MPP.*

5 The Impact of More Processors: Trade-offs Between k, r and OPT

We next analyze how more available processors can affect the optimal pebbling strategy for a DAG, which captures fundamental trade-offs for parallel computing. For convenience, we compare the simplest case of 1 processor to k processors, but our proofs are easy to carry over to any k-fold increase in the number of processors. We use $\mathtt{OPT}^{(k)}$ as a short notation for the optimal pebbling cost with k processors.

There are two natural ways to do this comparison. Let r_0 denote the amount of fast memory in the 1-processor case. One option is to compare this to a setting with k processors and $r := \frac{r_0}{k}$ fast memory on each; we call this the *fair* comparison, as the total fast memory over all processors remains unchanged. Another option is to compare to a setting with k processors and simply $r := r_0$ for each, i.e., all processors having the same fast memory r_0 as before. This *practical* comparison is more relevant for applications, where computations are often parallelized by simply employing more processors of the same kind.

"Fair" Comparison. We first compare the case of 1 processor with $r = r_0$, to the case of k processors with $r = \frac{r_0}{k}$. This is an interesting setting: on the one hand, the k processors allow for parallelization of computations and I/O, but on the other hand, processors have less fast memory, possibly resulting in further I/O steps to save and reload data. We first show that the optimum cost can decrease by a factor k at most; intuitively, this is because in the fair case, any pebbling strategy with k processors can be transformed into a 1-processor schedule with at most k times the cost. The bound is tight, as can be seen in e.g. a DAG with k independent chains of length $\frac{n}{k}$.

Lemma 7. *In the fair case, we have $\frac{\mathit{OPT}^{(k)}}{\mathit{OPT}^{(1)}} \geq \frac{1}{k}$, and there are DAGs such that $\frac{\mathit{OPT}^{(k)}}{\mathit{OPT}^{(1)}} = \frac{1}{k}$.*

With the same fast memory scattered over k processors, the optimum can also increase notably.

Lemma 8. *In the fair case, there is a construction with*

$$\frac{\mathit{OPT}^{(k)}}{\mathit{OPT}^{(1)}} \geq \frac{k-1}{k} \cdot g \cdot (\Delta_{in} - 1) + 1 - o(1)\,.$$

Recall that $\mathtt{OPT}^{(1)} \geq n$ and $\mathtt{OPT}^{(k)} \leq (g \cdot (\Delta_{in} + 1) + 1) \cdot n$, so this bound is essentially tight for large k and Δ_{in}. Another construction shows that the optimum can also be non-monotonic in k.

Lemma 9. *In the fair case, we can have* $\mathit{OPT}^{(2)} \leq \mathit{OPT}^{(1)}$ *and* $\mathit{OPT}^{(2)} \leq \mathit{OPT}^{(4)}$.

"Practical" Comparison. We now compare MPP with 1 and with k processors, with the processor(s) having the same $r = r_0$ in both cases. Here the larger k value only comes with advantages, so the optimum can never increase; on the other hand, it can easily decrease as before, e.g., for k independent chains. What makes this setting more interesting is that the optimum can decrease by a factor larger than k; such a *superlinear speedup* is a well-known (and highly desired) phenomenon in real-world systems. To our knowledge, MPP is the first DAG scheduling or pebbling model that naturally captures this behavior.

Lemma 10. *In the practical case, for any* $\varepsilon > 0$*, we can have* $\frac{\mathit{OPT}^{(1)}}{\mathit{OPT}^{(2)}} \geq \frac{\Delta_{in}-1}{2} - \varepsilon$.

The proof idea has already been outlined in Section 4 with the zipper gadget. With the appropriate Δ_{in}, the lemma can achieve a speedup of any constant factor, already for $k = 2$.

The number of I/O steps. We also briefly study the number of I/O steps separately: let $\mathtt{OPT}_{I/O}{}^{(k)}$ denote the number of steps (R1-M) and (R2-M) in the optimal pebbling with k processors. Our observations here hold for both the fair and the practical case.

Since $k = 1$ requires no communication, whereas $k = 2$ might, it is easy to construct a DAG where $\mathtt{OPT}_{I/O}{}^{(1)} = 0$, but $\mathtt{OPT}_{I/O}{}^{(2)} = \Theta(n)$. More surprisingly, we can also have a similar decrease in I/O, i.e., a DAG with $\mathtt{OPT}_{I/O}{}^{(1)} = \Theta(n)$, but $\mathtt{OPT}_{I/O}{}^{(2)} = 0$. Intuitively, this can happen when the computations can only be distributed in a very imbalanced way, so with 2 processors, it becomes more beneficial to do a lot of recomputations with one of the processors, instead of using I/O steps.

6 Inapproximability

Given the NP-hardness of MPP, a natural follow-up question is whether the optimum can be approximated to some factor in polynomial time. We show that MPP is also hard from this perspective.

Theorem 1. *MPP is APX-hard: there is a constant $\delta > 0$ such that no polynomial-time algorithm can approximate the optimum to a $(1+\delta)$ factor, unless $P = NP$.*

We prove this property already for the simplest case of $k = 1$, i.e., SPP with computation costs. The proof can then easily be extended to any k value.

Lemma 11. *SPP with computation costs is APX-hard.*

Proof sketch. Our proof is motivated by a construction in [28], which reduces one-shot SPP to vertex cover on a graph of N nodes to show another property. However, this proof considers SPP, where (R3-S) compute steps are free, and hence the proof is rather careless with the number of nodes, using very large gadgets to ensure asymptotic behavior. As a result, the I/O cost is only $\Theta(N) = o(n)$. With computation costs adding up to at least n in SPP, this makes the I/O costs asymptotically irrelevant. Hence the reduction does not work directly for pebbling models with computation cost.

To adapt this idea, we substantially decrease the size of gadgets to ensure that $n = O(N)$, and hence the I/O cost of any solution is in $\Theta(N) = \Theta(n)$. By design, in any reasonable solution, a specific part of this I/O cost is proportional to the size of a vertex cover in the underlying graph; as such, an approximation for the best pebbling could be transformed into an approximation of vertex cover. The main ingredients of the proof are as follows:

- we show how to modify the node gadgets in the construction of [28] to reduce their size to constant, thus ensuring that our construction has $n = \Theta(N)$ altogether,
- we execute some further changes in the gadgets to avoid some other undesired properties, which could be ignored in the original proof of [28] due to their asymptotic analysis,
- we then show that in any reasonable pebbling, the I/O cost has a linear term that is proportional to the size of a vertex cover in the underlying graph.

Altogether, this modified construction allows for an L-reduction to the vertex cover problem in 3-regular graphs, which is known to be APX-hard [1]. This implies that it is already NP-hard to find a pebbling of cost at most $(1 + \delta) \cdot \texttt{OPT}$ for an appropriate $\delta > 0$.

In general, the approximability of MPP is an intricate question, as the total cost is often dominated by computation costs, which are at least $\frac{n}{k}$. Thus, even if we have two solutions where I/O costs differ by a large factor, this can translate to only a negligible difference in total cost. To better capture these differences, we introduce an alternative cost metric.

Definition 1. *Given an MPP pebbling of cost C, its* surplus cost *is $C - \frac{n}{k}$.*

Intuitively speaking, surplus cost ignores this unavoidable cost of $\frac{n}{k}$, and instead only measures "imperfections" in the pebbling, such as I/O steps, work imbalance between the processors, and recomputations. In terms of this new metric, finding a good solution is much more challenging: it is not possible to

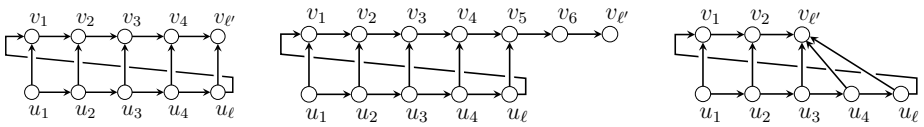

Fig. 3. Examples of consecutive levels. The 1st level (bottom row) always has size $\ell = 5$. The 2nd level (top row) has $\ell' = 5$ on the left side, $\ell' = 7$ in the middle, and $\ell' = 3$ on the right.

approximate the surplus cost in MPP to any finite factor. We again show this via proving an inapproximability result in standard one-shot SPP (i.e. minimizing I/O costs, with computations being free): we show that it is already NP-hard here to distinguish the cases when $\texttt{OPT} = 0$ and when $\texttt{OPT} \geq n^{1-\varepsilon}$, for any $\varepsilon > 0$.

Theorem 2. *In one-shot SPP, it is NP-hard to approximate the optimum to any finite multiplicative factor, and to any additive $n^{1-\varepsilon}$ term (for any $\varepsilon > 0$).*

Proof sketch. The main part of our proof is to develop a DAG construction where it is already NP-hard to decide whether one-shot SPP has $\texttt{OPT} = 0$ or $\texttt{OPT} \geq 1$. After this, some technical steps allow for extending the same result to an optimum of either 0 or $n^{1-\varepsilon}$.

When looking for a solution of cost 0, pebbling becomes notably simpler: blue pebbles cannot be used at all, since (R1-S) and (R2-S) incur cost. The use of (R4-S) is also simple: without recomputation, a red pebble should be deleted exactly when all out-neighbors have been pebbled. As such, a pebbling is characterized by the order of the n computation steps.

We point out that pebbling with only (R3-S) and (R4-S) is essentially equivalent to one-shot *black pebbling*, which is long known to be NP-hard [36]; as such, the novelty of this part of our theorem is somewhat limited. However, [36] considers a slightly different variant of pebbling, where compute steps can also "slide" a pebble from an in-neighbor. It may also be possible to adapt the proof in [36] to our case with further work; instead, we present a novel, somewhat simpler reduction based on a different problem, and we also devise new gadgets that might be of independent interest for future works on pebbling.

Our construction is organized into consecutive chains of *level gadgets* that form *towers*. Intuitively, a pebbling strategy always needs to keep one level of each tower in fast memory, and it can 'proceed' to the next level, computing the nodes of the next level and deleting pebbles from the current level. There is no benefit to having partially pebbled levels in our DAG, so intuitively, each level can be considered a single entity in our analysis. To pebble the DAG correctly, we need to go through the towers of levels in a carefully designed order, to ensure that the current set of levels never requires more than r pebbles altogether. The level gadgets are shown in Figure 3, and discussed in the full version [8].

Using these gadgets, we present a reduction from finding a clique of size q in a graph G'. Our DAG has a main tower, which is used to control the number of available pebbles at all times. We also add smaller tower gadgets for each node

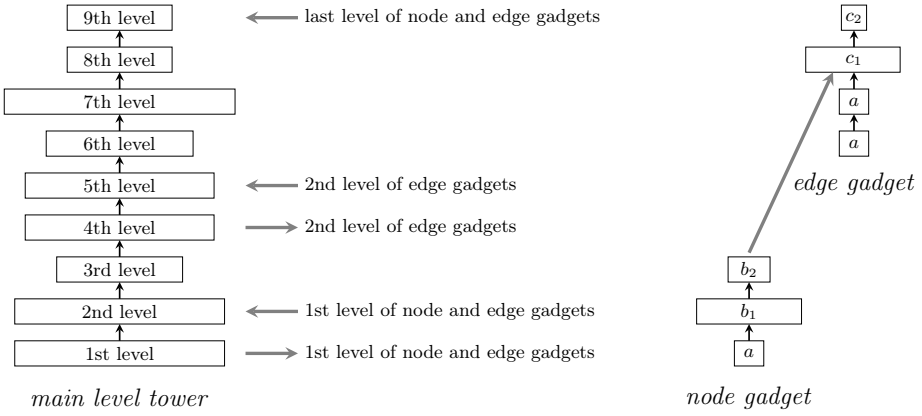

Fig. 4. High-level sketch of our construction: the main tower on the left, and node/edge gadgets on the right, with the level sizes and the dependency between incident node-edge pairs also shown.

and edge of G', as illustrated in Figure 4. Furthermore, we draw edges between specific levels of the node and edge gadgets whenever a node is incident to an edge, and also between the main tower and all node and edge gadgets.

In the beginning, we can only pebble the 1st level of the main tower, then the first level of each node/edge gadget, then the 2nd level of the main tower. Proceeding to the 3rd level of the main tower then frees up many pebbles; we can use these to proceed to the 2nd (and 3rd) level of node gadgets. However, due to $b_2 > a$, we can only move to the 3rd level of at most q node gadgets; otherwise, we would not have enough remaining pebbles to reach the 4th level of the main tower afterwards. Thus intuitively, any pebbling strategy corresponds to selecting at most q node gadgets.

The next levels of the main tower allow/require us to move to the 2nd level in edge gadgets. The 6th level again provides more free pebbles, so we can move to the 3rd (and then 4th) level in any edge gadget, provided that both of its incident nodes were chosen earlier. The next (7th) level then allows the fewest free pebbles: we can only proceed to this level if we reached the 4th level in at least $\binom{q}{2}$ edge gadgets. This is only possible if we found a set of q nodes in G' that span $\binom{q}{2}$ edges. Finally, after this point, the pebbling is easy to finish. Hence a pebbling of cost 0 exists if and only if G' has a clique of size q.

The theorem directly carries over to surplus cost in MPP: given the same DAG in MPP, a pebbling of cost 0 will translate to a surplus cost of 0, while a pebbling with $n^{1-\varepsilon}$ I/O steps (or the same amount of recomputation) will translate to a surplus cost of at least $n^{1-\varepsilon}$.

Corollary 2. *In MPP, it is NP-hard to approximate the optimum surplus cost to any finite multiplicative factor, and to any additive $n^{1-\varepsilon}$ term (for any $\varepsilon > 0$).*

References

1. Alimonti, P., Kann, V.: Some APX-completeness results for cubic graphs. Theoretical Computer Science **237**(1-2), 123–134 (2000)
2. Austrin, P., Pitassi, T., Wu, Y.: Inapproximability of treewidth, one-shot pebbling, and related layout problems. In: International Workshop on Approximation Algorithms for Combinatorial Optimization. pp. 13–24. Springer (2012)
3. Bennett, C.H.: Time/space trade-offs for reversible computation. SIAM Journal on Computing **18**(4), 766–776 (1989)
4. Bilardi, G., Scquizzato, M., Silvestri, F.: A lower bound technique for communication in BSP. ACM Transactions on Parallel Computing (TOPC) **4**(3), 1–27 (2018)
5. Blelloch, G.E., Chowdhury, R.A., Gibbons, P.B., Ramachandran, V., Chen, S., Kozuch, M.: Provably good multicore cache performance for divide-and-conquer algorithms. In: Proceedings of the 19th annual ACM-SIAM Symposium on Discrete Algorithms. pp. 501–510 (2008)
6. Blocki, J., Holman, B., Lee, S.: The parallel reversible pebbling game: Analyzing the post-quantum security of imhfs. In: Theory of Cryptography Conference. pp. 52–79 (2022)
7. Blocki, J., Ren, L., Zhou, S.: Bandwidth-hard functions: Reductions and lower bounds. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1820–1836 (2018)
8. Böhnlein, T., Papp, P.A., Yzelman, A.N.: Red-blue pebbling with multiple processors: Time, communication and memory trade-offs (full version). arXiv preprint arXiv:2409.03898 (2024)
9. Carpenter, T., Rastello, F., Sadayappan, P., Sidiropoulos, A.: Brief announcement: Approximating the I/O complexity of one-shot red-blue pebbling. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 161–163 (2016)
10. Chowdhury, R.A., Ramachandran, V.: Cache-efficient dynamic programming algorithms for multicores. In: Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 207–216 (2008)
11. Cook, S., Sethi, R.: Storage requirements for deterministic/polynomial time recognizable languages. In: Proceedings of the sixth annual ACM symposium on Theory of computing. pp. 33–39 (1974)
12. Cook, S.A.: An observation on time-storage trade off. In: Proceedings of the 5th annual ACM Symposium on Theory of Computing. pp. 29–33 (1973)
13. Demaine, E.D., Liu, Q.C.: Inapproximability of the standard pebble game and hard to pebble graphs. In: Workshop on Algorithms and Data Structures. pp. 313–324. Springer (2017)
14. Demaine, E.D., Liu, Q.C.: Red-blue pebble game: Complexity of computing the trade-off between cache size and memory transfers. In: Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 195–204 (2018)
15. Dymond, P.W., Tompa, M.: Speedups of deterministic machines by synchronous parallel machines. Journal of Computer and System Sciences **30**(2), 149–161 (1985)
16. Elango, V., Rastello, F., Pouchet, L.N., Ramanujam, J., Sadayappan, P.: On characterizing the data movement complexity of computational dags for parallel execution. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 296–306 (2014)

17. Elango, V., Rastello, F., Pouchet, L.N., Ramanujam, J., Sadayappan, P.: On characterizing the data access complexity of programs. In: Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 567–580 (2015)
18. Gilbert, J.R., Lengauer, T., Tarjan, R.E.: The pebbling problem is complete in polynomial space. In: Proceedings of the 11th annual ACM Symposium on Theory of Computing. pp. 237–248 (1979)
19. Gleinig, N., Hoefler, T.: The red-blue pebble game on trees and dags with large input. In: International Colloquium on Structural Information and Communication Complexity. pp. 135–153. Springer (2022)
20. Graham, R.L.: Bounds on multiprocessing timing anomalies. SIAM journal on Applied Mathematics **17**(2), 416–429 (1969)
21. Hong, J.W., Kung, H.T.: I/O complexity: The red-blue pebble game. In: Proceedings of the 13th ACM Symposium on Theory of Computing. pp. 326–333 (1981)
22. Hopcroft, J., Paul, W., Valiant, L.: On time versus space. Journal of the ACM (JACM) **24**(2), 332–337 (1977)
23. Jain, S., Zaharia, M.: Spectral lower bounds on the I/O complexity of computation graphs. In: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures. pp. 329–338 (2020)
24. Kwasniewski, G., Ben-Nun, T., Gianinazzi, L., Calotoiu, A., Schneider, T., Ziogas, A.N., Besta, M., Hoefler, T.: Pebbles, graphs, and a pinch of combinatorics: Towards tight I/O lower bounds for statically analyzable programs. In: Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures. pp. 328–339 (2021)
25. Kwasniewski, G., Kabić, M., Besta, M., VandeVondele, J., Solcà, R., Hoefler, T.: Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–22 (2019)
26. Lengauer, T., Tarjan, R.E.: Asymptotically tight bounds on time-space trade-offs in a pebble game. Journal of the ACM (JACM) **29**(4), 1087–1130 (1982)
27. McColl, W., Tiskin, A.: Memory-efficient matrix computations in the BSP model. Algorithmica **24**(3-4), 287–297 (1999)
28. Papp, P.A., Wattenhofer, R.: On the hardness of red-blue pebble games. In: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures. pp. 419–429 (2020)
29. Papp, P.A., Anegg, G., Yzelman, A.N.: DAG scheduling in the BSP model (2023), arXiv preprint arXiv:2303.05989
30. Paterson, M.S., Hewitt, C.E.: Comparative schematology. In: Record of the Project MAC conference on concurrent systems and parallel computation. pp. 119–127 (1970)
31. Ranjan, D., Savage, J., Zubair, M.: Upper and lower I/O bounds for pebbling r-pyramids. Journal of Discrete Algorithms **14**, 2–12 (2012)
32. Reischuk, R.: Improved bounds on the problem of time-space trade-off in the pebble game. Journal of the ACM (JACM) **27**(4), 839–849 (1980)
33. Savage, J.E.: Extending the Hong-Kung model to memory hierarchies. In: International Computing and Combinatorics Conference. pp. 270–281. Springer (1995)
34. Savage, J.E., Zubair, M.: A unified model for multicore architectures. In: Proceedings of the 1st international forum on next-generation multicore/manycore technologies. pp. 1–12 (2008)

35. Scott, J., Holtz, O., Schwartz, O.: Matrix multiplication I/O-complexity by path routing. In: Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures. pp. 35–45 (2015)
36. Sethi, R.: Complete register allocation problems. In: Proceedings of the 5th ACM Symposium on Theory of Computing. pp. 182–195 (1973)
37. Solomonik, E., Carson, E., Knight, N., Demmel, J.: Trade-offs between synchronization, communication, and computation in parallel linear algebra computations. ACM Transactions on Parallel Computing (TOPC) **3**(1), 1–47 (2017)
38. Tiskin, A.: The bulk-synchronous parallel random access machine. Theoretical Computer Science **196**(1-2), 109–130 (1998)
39. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM **33**(8), 103–111 (1990)
40. Valiant, L.G.: A bridging model for multi-core computing. Journal of Computer and System Sciences **77**(1), 154–166 (2011)