Online Parallel Paging with Optimal Makespan

Kunal Agrawal
Washington University in St. Louis
St. Louis, USA
kunal@wustl.edu

Michael A. Bender
Stony Brook University
Stony Brook, USA
bender@cs.stonybrook.edu

Rathish Das
University of Waterloo
Waterloo, Canada
rathish.das@uwaterloo.ca

William Kuszmaul
MIT
Cambridge, USA
kuszmaul@mit.edu

Enoch Peserico
Università degli Studi di Padova
Padova, Italy
enoch@dei.unipd.it

Michele Scquizzato
Università degli Studi di Padova
Padova, Italy
scquizza@math.unipd.it

ABSTRACT

The classical paging problem can be described as follows: given a cache that can hold up to k pages (or blocks) and a sequence of requests to pages, how should we manage the cache so as to maximize performance—or, in other words, complete the sequence as quickly as possible. Whereas this sequential paging problem has been well understood for decades, the parallel version, where the cache is shared among p processors each issuing its own sequence of page requests, has been much more resistant. In this problem we are given p request sequences $R^1, R^2, \ldots, R^p$, each of which accesses a disjoint set of pages, and we ask the question: how should the paging algorithm manage the cache to optimize the completion time of all sequences (i.e., the makespan). As for the classical sequential problem, the goal is to design an online paging algorithm that achieves an optimal competitive ratio, using $O(1)$ resource augmentation.

In a recent breakthrough, Agrawal et al. [SODA '21] showed that the optimal (deterministic) competitive ratio C for this problem is in the range $\Omega(\log p) \leq C \leq O(\log^2 p)$. This paper closes that gap, showing how to achieve a competitive ratio $C = O(\log p)$. Our techniques reveal surprising combinatorial differences between the problem of optimizing makespan and that of optimizing the closely related metric of mean completion time; and yet our algorithm manages to be simultaneously asymptotically optimal for both tasks.

CCS CONCEPTS

• **Theory of computation → Caching and paging algorithms**.

KEYWORDS

Paging; parallel paging; multicores; online algorithms

ACM Reference Format:
Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. 2022. Online Parallel Paging with Optimal Makespan. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3490148.3538577

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9146-7/22/07.
https://doi.org/10.1145/3490148.3538577

1 INTRODUCTION

This paper considers the problem of ***parallel paging*** [2, 12, 14, 16, 19]: p processors share a fast memory (or ***cache***) that can hold up to k pages. When a processor accesses a location contained in a page that is in cache, the access cost is small (the access is a ***hit***); when it accesses a location contained in a page that is not in cache, the access cost is large (the access is a ***miss*** or a ***fault***). As with prior work on parallel paging [2, 14, 19], we assume that all p processors access distinct sets of memory locations.[1] The paging algorithm decides which ***pages*** (or ***blocks***) remain in cache at any point in time or, in other words, which page(s) to evict when a new page is brought into cache. The goal is to share the cache among the processors in a way that minimizes some objective function of processors' completion times.

Sequential vs. parallel paging. Whereas sequential paging (i.e., the paging problem with one processor) has been well understood for decades [4, 5, 20, 24], the problem of obtaining tight bounds for parallel paging has proved to be much more elusive [3, 6, 11, 12, 15–17, 21]. The goal is to determine the best possible online competitive ratio achievable using $O(1)$ resource augmentation.[2]

Part of what makes parallel paging difficult is that the scheduler must predict dynamically which processors are going to benefit more from having more/less cache and then must solve the online optimization problem of allocating memory optimally based on those predictions. Some processors may benefit greatly from having access to additional memory, while others might not. For each individual processor, the *marginal benefit* of having access to $i + 1$ space of cache, rather than i space, may be a non-monotonic function in i; and these marginal benefits may also fluctuate unpredictably over time as processors go through their respective request sequences. The parallel-paging algorithm must also be careful to not be too "sporadic" in its memory allocations; if a processor is allocated a large amount of cache, but only for a short period of time, that cache may be useless to the processor.

[1] This restriction represents the condition where each processor is running a distinct program and these programs do not share pages with each other.
[2] Constant-factor resource augmentation means that the algorithm's cache is a constant-factor larger than the cache given to the optimal offline algorithm OPT. Even in the sequential setting, resource augmentation is necessary, since otherwise any deterministic algorithm has competitive ratio $\Theta(k)$ [24].

A second challenge is that the decisions made by the scheduler can have unpredictable downstream effects on how the processors interact later in their access sequences. If some processor x is given more cache now, then it will proceed through its accesses faster than another processor y that is given less cache now. This means that, later on, the *alignment* between the two processors and their respective access sequences will be shifted due to allocation decisions that were made earlier. Even if an allocation decision seems beneficial in the short term, it is difficult to assess whether it will be a competitive decision in the long run.

The interactions between these two challenges make it so that even the offline version of this problem is difficult—in fact, it is NP-hard [19].

Finally, an interesting feature of the parallel paging problem (as opposed to the sequential version) is that there is no single best objective function. In the language of traditional scheduling metrics, there are two natural objective functions: ***makespan*** (i.e., maximum completion time) and ***mean completion time***. Some applications must wait on every processor to complete (e.g., parallel threads with a synchronization barrier) before the application can be considered finished, meaning that the objective is to minimize makespan; other applications are more concerned with how long the average processor spends on its task, meaning that the objective to minimize is mean completion time. What makes these two objective functions especially interesting is that they yield fundamentally different notions of what it means to allocate cache efficiently to processors.

Past work on parallel paging. First articulated by Fiat and Karlin in 1995 [12], the problem of achieving optimal competitive ratios for parallel paging has remained open for nearly three decades [2, 14, 16, 19]. In the online setting, most of the early work focused on simplifications of the problem [3, 6, 11, 12, 16, 17, 21] in which the rate at which each processor progresses is fixed, rather than being affected by how many hits and misses the processor incurs; that is, a processor that incurs all hits is treated as progressing through its access sequence at the same rate as if it incurred all misses. The downside of this assumption is that it "sequentializes" the interleaving between the access sequences, thereby removing the interactions that occur between the scheduler's decisions and the interleavings between the processors.

Recent work has focused on understanding the more complete version of the paging model in which the speed at which a processor progresses is affected by whether it incurs hits or misses [9, 10, 14, 19]. Agrawal et al. [1, 2, 8] achieved the first general-purpose results in the online setting: for mean completion time, they deterministically achieved a competitive ratio of $O(\log p)$ and showed that no deterministic algorithm can do better.

The problem of determining the optimal deterministic competitive ratio C for makespan has continued to be elusive, however, with the best asymptotic bounds establishing that $\Omega(\log p) \le C \le O(\log^2 p)$ [1, 2]. The current paper closes this gap, introducing a new parallel-paging algorithm that achieves an optimal ratio of $C = O(\log p)$.

Before we describe our technical results in detail, it is helpful to understand the structure of the recent results on mean completion time [1, 2]. Indeed, a key technical element of this paper will be to study the (somewhat surprising) ways in which these ideas do (and do not!) extend to the makespan setting.

Understanding the recent progress on mean completion time. The significant insight by [1, 2] was that the parallel paging problem is tightly connected to the seemingly unrelated ***green paging problem***.[3] In green paging, there is a *single processor*, and there is a paging algorithm that decides the amount of cache allocated to the processor over time. The goal is to service the processor's request sequence while minimizing the integral of the processor's cache capacity over time—a quantity known as ***memory impact***. (Notice that it is not optimal to minimize the cache size at all times, since this can increase cache misses, which increases the integral of cache size over time.)

There turns out to be an intimate connection between parallel paging and green paging. Indeed, [1, 2] proved via reductions that the best achievable deterministic competitive ratio for green paging is asymptotically the same as that for parallel-paging mean completion time. By analyzing green paging, they were able to then deduce that the optimal competitive ratio for both problems is $\Theta(\log p)$.

The main algorithmic takeaway from [1, 2] is that, if one wishes to optimize parallel-paging mean completion time, one should instead focus on optimizing the green-paging performance of each individual processor: the competitive ratio that one achieves for the latter problem directly translates to the former.

This paper: Obtaining tight bounds for makespan. Our first result establishes a counter-intuitive lower bound—that green paging algorithms *cannot*, in general, be transformed into optimal parallel-paging makespan algorithms without a loss in competitive ratio. Specifically, Theorem 4 establishes that, even if we are given an *omniscient* green paging algorithm that is *constant-competitive*, then any black-box construction of a parallel-paging algorithm that allocates cache using this green paging algorithm must incur a competitive ratio of $\Omega(\log p / \log\log p)$.[4]

This lower bound establishes that, when we consider makespan, there are two separate sources that individually force a roughly logarithmic loss in competitiveness—the first source is the lower bound of $\Omega(\log p)$ for green paging [2], and the second source is the conversion of green paging to parallel paging, which contributes a factor of $\Omega(\log p / \log\log p)$.

The main result of this paper is that it is possible to design a parallel-paging algorithm in which these two logarithmic factors *add*, instead of *multiplying* (as in [1, 2]).

We begin by constructing a randomized parallel-paging algorithm with competitive ratio $O(\log p)$. This algorithm uses randomization to "hide" any vulnerabilities that the parallel-paging algorithm might have based on when it gives which processors large/small amounts of cache. Each individual processor is allocated cache using a specially-designed randomized green paging

[3] We remark that, although this paper studies green paging primarily for its relationship to parallel paging, green paging has also been studied as a problem of independent interest [7, 13, 18, 22, 23], since it closely relates to the amount of energy that a processor consumes in its cache usage.

[4] This result requires a more careful definition of what an omniscient green paging algorithm is able to do—Section 4 provides a natural definition.

algorithm, and then the allocations are interleaved in such a way that we do not pay an additional $\log p$ factor in competitive ratio.

Finally, we show that it is possible to *deterministically* emulate our randomized algorithm in such a way that we still achieve a competitive ratio of $O(\log p)$. Rather than being a direct derandomization, our deterministic algorithm is instead obtained by formally capturing the properties of the randomized algorithm that enable its analysis, and showing how to achieve the same properties deterministically. Our deterministic competitive ratio of $O(\log p)$ matches a lower bound by [1, 2].

Perhaps surprisingly, our $O(\log p)$ competitive ratio for makespan is achieved without any loss of competitive ratio on average completion time. This results in a single algorithm that simultaneously achieves the optimal deterministic competitive ratio of $O(\log p)$ for both objectives.

An interesting feature of all of the algorithms in this paper (as well as those in [1, 2]) is that they are ***oblivious***, meaning that they do not adapt to the specific request sequences given to them, but instead use a "universal" strategy for how to allocate cache to processors efficiently (each processor individually uses LRU on the cache that it is allocated). This means that, rather than predicting the behavior of individual processors, the algorithm must allocate cache in such a way that it is $O(\log p)$-competitive *regardless* of the processors' access sequences. The fact that such an algorithm can achieve a competitive ratio of $O(\log p)$ (and that this is optimal) is perhaps the most surprising takeaway of this line of work.

Outline. Section 2 presents preliminary definitions and modeling. Section 3 presents our upper bounds for online parallel paging. Finally, Section 4 presents our lower bound for parallel-paging algorithms that make use of green paging as a black-box primitive.

2 PRELIMINARIES

This paper follows the same set of conventions and definitions as in recent previous work [1, 2]. For completeness, we give a brief overview of those definitions here.

The parallel paging model. The parallel paging model features p processors connected to a shared cache of size $k > p$. Each processor has a dedicated channel to the cache and to a main memory of unlimited size. Each processor i issues a sequence of requests for pages $R^i = r^i_0, r^i_1, r^i_2, \ldots$, where r^i_j is the j-th page request by the i-th processor. The requests of each processor are served in order, that is, request r^i_{j+1} can be requested only after r^i_j has been served. The request sequences are assumed to be disjoint, that is, for all $i \neq j$ and $\forall q, s$, $r^i_q \neq r^j_s$.

When a processor requests a page, if that page is in cache then it takes 1 unit of time to service this request. That is, if a page is requested at time t and it is in cache, then the request is served at time $t + 1$. If the requested page is not in cache, then it takes $s > 1$ units of time to transfer this page from main memory to cache. Therefore, this request is served at time $t + s + 1$ or later. Note that the paging algorithm does not necessarily serve the block *exactly* at time $t + s + 1$ because the paging algorithm can choose to hold off retrieving the block from memory if it deems that the cache should be used to hold other blocks.

Parallel paging. A ***parallel-paging algorithm*** controls when pages are evicted from cache—that is, whenever a new page is brought into cache, the paging algorithm chooses which of the pages currently in cache must be evicted (unless the cache is underfull). The algorithm is also permitted to stall a processor if it wishes, that is it can refuse to serve a page request until a later time.

The choices made by the parallel-paging algorithm determine which processors incur hits/misses on their requests, and how long each processor takes to complete. In this paper, the goal of a paging algorithm will be to minimize ***makespan***, which is the amount of time that it takes for *all* of the processors to complete their request sequences. We will use $T_{OPT}(k)$ to denote the optimal offline makespan on a cache of size k, and $T_{\mathcal{A}}(\ell)$ to denote the makespan of a given algorithm $\mathcal{A}$ on a cache of size ℓ. We will be interested in ***online algorithms***, meaning that the algorithm does not get to know what pages each processor will access until the requests are made.

To analyze an online parallel-paging algorithm, we use competitive analysis: an online algorithm $\mathcal{A}$ is said to achieve ***competitive ratio*** C with resource-augmentation ξ if for all request sequences $R^1, \ldots, R^p$ such that $\max_i |R^i|$ is sufficiently large as a function of p, k, we have that $T_{\mathcal{A}}(\xi k) \leq C \cdot T_{OPT}(k)$. In general, when proving upper bounds, we will assume the resource-augmentation parameter ξ to be $O(1)$. In this context, we will use as a shorthand T_{OPT} to denote $T_{OPT}(k)$ and $T_{\mathcal{A}}$ to denote $T_{\mathcal{A}}(\xi k)$. We remark that, even in the setting of $p = 1$, resource augmentation is needed if one wishes to achieve a competitive ratio better than the trivial $\Theta(k)$ for deterministic algorithms [24].

A useful tool: Compartmentalized box profiles. Past work [1, 2] has shown that several simplifying assumptions can be made without loss of generality about how the optimal parallel-paging algorithm OPT behaves. Each individual processor can be assumed to use LRU-eviction (i.e., evict the least-recently-accessed page) on the portion of the cache that it is allocated, meaning that the parallel paging problem is fundamentally about choosing *how much* cache to allocate each processor at a time.

One can further assume that each individual processor is allocated memory in ***boxes***, where a ***box of height $j \leq k$*** means that the processor has access to j pages in cache for $s \cdot j$ time steps (recall that s is the time to transfer a page from main memory to cache in the event of a cache miss). Moreover, without loss of generality, every box can be assumed to have a power-of-two height. Finally, boxes can also be assumed to be ***compartmentalized***, meaning that whenever a processor is first allocated a box of height j, any pages that were formerly in those j positions in cache are initially evicted (even if they would have been useful to keep around). As we shall see, the assumption that each processor is allocated cache in compartmentalized boxes significantly streamlines the discussion of parallel-paging algorithms.

Green paging. Finally, we conclude the section by briefly defining the related problem of ***green paging***. In the green paging problem, a single processor must service a single request sequence R using a cache whose size changes dynamically over time within the range $[k/p, k]$, where p is a given parameter. A ***green-paging algorithm*** dictates how the cache size evolves, as well as the replacement

policy. The goal of the algorithm is to minimize the integral of the cache size over time—a quantity called ***memory impact***. That is, if c_i is the amount of cache allocated at time i, then we wish to complete the request sequence R while minimizing $\sum_i c_i$.

As for parallel paging, one can analyze an online green-paging algorithm using competitive analysis with $\xi = O(1)$ resource augmentation—this means that the algorithm allocates a cache with size between $\xi k/p$ and ξk and is compared to an optimal offline algorithm OPT that allocates a cache with size between k/p and k. Also like parallel paging, one can assume without loss of generality [1, 2] that the processor manages its cache with LRU; that OPT allocates memory to the processor using compartmentalized boxes; and that the boxes have normalized sizes, so that each box has a height of the form $k/p \cdot 2^j$ for $j \in [\log p]$.

We will be interested in green paging primarily for its use as a tool to solve parallel paging. We study this relationship in depth in Section 4, where we also extend the definition of green paging to the setting where k and p evolve over time.

3 UPPER BOUNDS

In this section we present a online parallel-paging algorithms that achieves an $O(\log p)$-competitive makespan. We break the section into three parts. As a motivation, we begin by constructing a new green-paging algorithm which uses a very simple random process to generate a memory profile that is $O(\log p)$-competitive with optimal. We then show how to use a similar random process for parallel paging without any additional loss in the competitive ratio. Finally, we show how to derandomize our algorithm to obtain optimal deterministic online parallel paging, with a competitive ratio of $O(\log p)$ for makespan.

3.1 Randomized Online Green Paging

As a warm-up, we present a new algorithm for the closely related problem of green paging. This algorithm, which we call RAND-GREEN, achieves the same $O(\log p)$ competitive ratio as in past work [2], but with a remarkably simple randomized approach—this same approach will play an important role in our strategy for *parallel* paging in subsequent sections.

Consider a green-paging instance with minimum memory size k/p and maximum memory size k. Recall that, without loss of generality (and with $O(1)$ resource augmentation), we can assume that k and p are powers of two and that boxes have heights $k/p, 2k/p, 4k/p, \ldots, k$.

The RAND-GREEN algorithm generates a sequence of boxes, where the height of each box is selected independently from a fixed probability distribution $\mathcal{D}$ designed so that the probability of a given box-height j being selected is inversely proportional to the memory impact sj^2 of that box. In more detail, whenever it is time to select a new box, the algorithm randomly selects a box height $j \in \{k/p, 2k/p, 4k/p, \ldots, k\}$ so that each j has probability $\Theta\left(k^2/j^2p^2\right)$.

The intuition behind the algorithm is as follows. If a box has height j, then it will contribute sj^2 to the memory impact incurred by the algorithm. By selecting each box height j to occur with probability $\sim 1/j^2$, we equalize the expected contribution to memory impact of all the different box heights. The result is that, if some part of the request sequence requires a box of some height j, then the expected memory impact that we will incur until we get a box of that size j is only $O(\log p) \cdot j^2$. We now prove this formally, and analyze the competitive ratio of the algorithm.

Lemma 1. *Let $j \in \{k/p, 2k/p, 4k/p, \ldots, k\}$, and consider a box whose height is j with probability $\Theta\left(k^2/j^2p^2\right)$. Let X be the indicator random variable for the event that the box has height j, and let Y be the memory impact of the box. Then*

$$\mathbb{E}[XY] = \Theta\left(\frac{k^2 s}{p^2}\right).$$

Proof.

$$\mathbb{E}[XY] = \Pr[X] \cdot \mathbb{E}[Y \mid X] = \Theta\left(\frac{k^2}{j^2p^2}\right) \cdot j^2 s = \Theta\left(\frac{k^2 s}{p^2}\right). \qquad \square$$

Theorem 1. *With $O(1)$ resource augmentation,* RAND-GREEN *is $O(\log p)$-competitive in expectation.*

Proof. With $O(1)$ resource augmentation, we can assume without loss of generality that OPT uses a compartmentalized box profile with power-of-two sized boxes, and that k and p are powers of two. Let S be the sequence of boxes that OPT uses, and let R be the sequence of boxes that RAND-GREEN uses. Observe that RAND-GREEN finishes the request sequence if S is a subsequence of R.

At each time t, let z be the dynamically changing variable that specifies the height of the next box in S. That is, at time $t = 0$, z is the height of the first box in S. Once RAND-GREEN allocates a box of height z, the value z changes to the height of the next box in S, and so on. The box that was allocated prior to z changing (i.e., the box of height z) is called a ***useful box***, and the memory impact of any useful box is said to be ***useful memory impact***.

We now calculate the expected memory impact of RAND-GREEN until RAND-GREEN finishes the request sequence. Let j be the height of a box chosen randomly by RAND-GREEN. If $j = z$, then the box is useful. Lemma 1 (with $j = z$) tells us that the expected useful memory impact of a randomly chosen box is $\Theta(k^2 s/p^2)$. On the other hand, the expected (useful and wasted) memory impact of a given box chosen by RAND-GREEN is $\Theta(k^2 s/p^2 \cdot \log p)$—which can be seen by summing Lemma 1 across all $\Theta(\log p)$ options for j.

Thus the expected memory impact that a given box contributes is precisely a $\Theta(\log p)$-factor larger than the expected *useful* memory impact that the box contributes. By linearity of expectation, it follows that the expected total memory impact of RAND-GREEN is at most an $O(\log p)$-factor larger than the total useful memory impact of RAND-GREEN. However, the latter quantity is deterministically at most the cost of OPT, implying that RAND-GREEN is $O(\log p)$-competitive in expectation. $\square$

3.2 Randomized Online Parallel Paging

We now give a randomized online parallel-paging algorithm RAND-PAR that achieves an $O(\log p)$ competitive ratio. Algorithm RAND-PAR has at most $\log p$ ***phases***, where phase i ends and phase $i+1$ begins once half of the processors that were ***active*** (i.e., still running) at the start of phase i finish.

Breaking each phase into algorithmic chunks. Let r be the number of active processors at the start of a chunk. Each phase proceeds in ***chunks***, where each chunk has a ***primary part*** followed by a ***secondary part***. The length ℓ_1 of the primary part is fixed, and only depends on r. In particular, the length of the primary part of the chunk is $\Theta(sk \log r/r)$ and in the primary part of the chunk, RAND-PAR gives each active processor exactly k/r memory for for its entire length ℓ_1. Another way of thinking about it is that $RAND - PAR$ gives each processor $\log r$ the minimum size boxes of height k/r.

In the secondary part, RAND-PAR randomly chooses a box size according to the randomized online green-paging algorithm RAND-GREEN, that is, the box's area (i.e., memory impact) is inversely proportional to the probability of it being chosen. Thus the probability that the box has height $j \in \{k/r, 2k/r, 4k/r, \ldots, k\}$ (where each of the heights is rounded up to the next power of two) is $\Theta\left(\frac{k^2}{j^2 r^2}\right)$. Then RAND-PAR allocates a box of this size to each processor. That is, if j is the randomly selected box height, then the secondary part of each chunk consists of one height-j box for each of the $\Theta(r)$ processors that remain. Note that the total memory impact of these boxes is $\Theta(srj^2)$. The length of the secondary part of the chunk is thus $\ell_2 = \Theta(srj^2/k)$.

Based on the way the primary and secondary parts of the algorithm are designed, we can make the following observation which simply states that the primary and secondary parts of each chunk have the same length and the same cache impact (in expectation).

Observation 1. *For any chunk* Π *the length of the primary part of each chunk* ℓ_1 *is equal to the expected length of the secondary part of the chunk* $E[\ell_2]$*. In addition, since* RAND-PAR *always makes use of at least a constant fraction of memory, the cache impact of its primary part is the same as the expected cache impact of its secondary part.*

Intuition for why the algorithm does well. We now describe the intuition for why RAND-PAR achieves a good competitive ratio.

We shall argue that there are essentially two possible modes that a given processor can be in at any given moment: the first mode is when the processor is time-bound, and any size box will suffice for the processor to make progress; the second mode is when the processor is memory-impact bound, and it needs a relatively large box in order to continue making substantial progress.

If most of the processes are in the first mode (i.e., time-bound), then we want to assign every process the minimum box size (this is what the primary part of each chunk does). On the other hand, if most of the processes are in the second mode (i.e., memory-impact bound) then we want to assign boxes to them using a green-paging algorithm (this is what the secondary part of each chunk does). Whenever a chunk is in the former case, we call it ***time-efficient***, and whenever a chunk is in the latter case, we call it ***impact-efficient***. Time-efficient chunks benefit from their primary parts, and impact-efficient chunks benefit from their secondary parts.

In other words, each chunk has one part (out of its primary and secondary parts) that is "useful" and one part that is not. This is why we design the two parts to have the same expected lengths as each other, that way the time spent on the non-useful part can be amortized to the time spent on the useful part.

Before we perform the formal analysis, let us remark on the important role that randomization plays in this algorithm. There are certain points in time when a parallel-paging algorithm should be viewed as "vulnerable", in particular, these are the times when the parallel-paging algorithm has selected a large box size for some processor. If the processor doesn't need that large box size, then the parallel-paging algorithm has just wasted a large amount of resources. In the context of our algorithm, if an adversary knew which chunks were going to allocate (very) large boxes within their secondary parts, then the adversary could choose those specific chunks to be time-efficient (rather than impact-efficient), thereby rendering the large boxes wasted. The purpose of randomization is to make it so that the adversary cannot predict which chunks will contain large boxes until it is too late for the adversary to exploit that knowledge. That is, randomization prevents the adversary from strategically choosing which chunks are time-efficient versus which chunks are impact-efficient in any way that could thwart the parallel-paging algorithm's effectiveness. As we shall see in the next section, this randomization can actually be eliminated by designing a paging algorithm with the property that it is never "too vulnerable".

Analysis. We now prove that RAND-PAR is $O(\log p)$ competitive.

First, we give some notation. With $O(1)$ resource augmentation, we can assume without loss of generality that OPT uses a compartmentalized box profile for each processor (and that the box sizes are powers of two). Let S_i be the sequence of boxes that OPT uses for processor i. Let R_i be the sequence of boxes that RAND-PAR uses for processor i. By construction, up until the time when processor i or processor j finishes, the box sequences for R_i and R_j are the same. Observe that RAND-PAR finishes the request sequence for processor i if S_i is a subsequence of R_i. In this case, some boxes in sequence R_i correspond to boxes in sequence S_i—these are called ***useful boxes***. The rest are called ***wasted boxes***. As in the previous subsection, we say that the memory impact from a useful box is ***useful memory impact***; the memory impact from a wasted box is ***wasted memory impact***.

At each time t, let z_i be the dynamically changing variable that specifies the height of the next box in sequence S_i. That is, at time $t = 0$, z_i is the height of the first box in S_i. Once RAND-PAR allocates a box of height z_i, the value z_i changes to the height of the next box in S_i, and so on.

We now divide the algorithm's chunks into two categories, namely, time-efficient and impact-efficient, as defined next. Consider a time-step in the primary part of a chunk. If more than $r/4$ of the processors i satisfy $z_i \leq k/r$, then we call the step ***time-efficient***. Otherwise, we call the step ***impact-efficient***. A chunk is ***time-efficient*** if all the time steps in its entire primary part are time-efficient; otherwise the chunk is ***impact-efficient***. Note here that the primary part of a chunk contains several boxes (in particular, $\Theta(\log r)$ boxes, each of size k/r) — therefore, it is possible for some steps in a chunk to be time-efficient while others are not. It is important to note that even if only some, but not all, steps are time-efficient, the chunk can still be impact efficient.

Since every chunk is either time-efficient or impact-efficient, we can bound the makespan of RAND-PAR by separately analyzing

the total length of the time-efficient and impact-efficient chunks. We first bound the total length of time-efficient chunks.

Lemma 2. *Let T_{OPT} be the makespan of the optimal algorithm OPT. Then the total length of all the time-efficient chunks of* RAND-PAR *is $O(\log p \cdot T_{OPT})$ in expectation.*

Proof. Each time-efficient step in RAND-PAR can be viewed as making one time-step of progress on each of at least $r/4$ processors in OPT. It follows that each phase can have at most $4T_{\text{OPT}}$ time-efficient steps. The total number of time-efficient steps in the full algorithm is therefore $O(\log p \cdot T_{\text{OPT}})$. This means that the sum of the lengths of the primary parts of the time-efficient chunks is at most $O(\log p \cdot T_{\text{OPT}})$. On the other hand, the expected length of the secondary part of each time-efficient chunk is equal to the length of the primary part. Thus the expected total length of all the time-efficient chunks is $O(\log p \cdot T_{\text{OPT}})$. □

We now bound the total length of the impact-efficient chunks. We first show that in any impact-efficient chunk Π, in expectation, the total *useful* memory impact that RAND-PAR makes in the chunk is at least a $\Theta(1/\log p)$-factor of the *total* memory impact.

Lemma 3. *Let Π be an impact-efficient chunk, and let r be the number of active processors at the start of the chunk. Then the expected useful memory impact that* RAND-PAR *makes in chunk Π is a $\Theta(1/\log r)$-factor of the expected total memory impact that* RAND-PAR *makes in chunk Π.*

Proof. The proof is similar to that of Theorem 1 in the way it compares useful memory impact to wasted memory impact. Consider an impact-efficient chunk, and let j be the height of the random boxes used by (all) the active processors in the secondary part of the chunk. Since the chunk is impact-efficient, at some point in the primary part of the chunk there is an impact-efficient time step, which means that there are at least $r/2 - r/4 = \Omega(r)$ processors i that all satisfy $z_i > k/r$ in that time step.

For each of those $\Omega(r)$ processors, if $j = z_i$ then the box contributes sz_i^2 useful memory impact. By Lemma 1 (with $j = z_i$), the expected useful memory impact that a given box makes for processor i is $\Theta(k^2 s/r^2)$. Since there are $\Omega(r)$ processors, the total expected useful memory impact of a randomly chosen box is $\Theta(k^2/r)$.

Since there are $\log r$ different box sizes, and each box size contributes expected memory impact $\Theta(k^2 s/r^2)$ to each processor (by Lemma 1), the expected memory impact (whether useful or wasted) of a randomly chosen box for a particular processor is $\Theta(\log r \cdot (k/r)^2)$. Hence, the total expected memory impact of all r processors in the (secondary part of the) chunk is $\Theta(\log r \cdot k^2/r)$.

From Observation 1, the impact of the primary part of a chunk is the same as the expected impact of its secondary part. Thus the expected total memory impact incurred by the algorithm across all processors is $\Theta(\log r \cdot k^2/r)$. □

Lemma 4. *Let T_{OPT} be the makespan of the optimal algorithm OPT. Then the total length of all the impact-efficient chunks of* RAND-PAR *is $O(\log p \cdot T_{OPT})$ in expectation.*

Proof. From Lemma 3, in any impact-efficient chunk Π, in expectation the total useful memory impact that RAND-PAR makes in the chunk is a $\Theta(1/\log p)$-factor of the total expected memory impact (whether useful or wasted). Summing over all the impact-efficient chunks, we get the expected total useful memory impact A that RAND-PAR makes is a $\Theta(1/\log p)$-factor of the total expected memory impact B that RAND-PAR makes. Notice, however, that A is deterministically equal to the memory impact of OPT, which is $O(ks \cdot T_{\text{OPT}})$. On the other hand, B is $\Theta(ks \cdot T_{RAND-PAR})$. Thus the total length of all the impact-efficient chunks is $O(\log p \cdot T_{\text{OPT}})$. □

Theorem 2. *Assuming $O(1)$ resource augmentation, the expected makespan of* RAND-PAR *is $O(\log p \cdot T_{OPT})$.*

Proof. It follows immediately from Lemmas 2 and 4, since each chunk of RAND-PAR is either time-efficient or impact-efficient. □

An interesting feature of our algorithm is that, although it achieves competitive ratio $O(\log p)$, it *also* fits into the mold described by Theorem 4, i.e., our algorithm uses a green-paging algorithm as a black box. Thus we have:

Corollary 1. RAND-PAR *is a randomized parallel paging algorithm that assigns boxes to each processor using a black-box $\Theta(\log p)$-competitive green paging algorithm, that uses a factor of $O(1)$ resource augmentation, and that achieves competitive ratio $O(\log p)$ for makespan.*

This corollary comes as a surprising complement to Theorem 4. Recall that Theorem 4 says that the use of a black-box green paging algorithm (even a clairvoyant $O(1)$-competitive green-paging algorithm) forces a competitive ratio of $\tilde{\Omega}(\log p)$, where notation $\tilde{\Omega}$ hides a $1/\text{polyloglog}(p)$ factor. Given that we are using a $\Theta(\log p)$-competitive green-paging algorithm (which in past work [2] has been shown to be the best possible online green-paging algorithm), it is tempting to assume that we must incur a competitive ratio of $\tilde{\Omega}(\log^2 p)$. Remarkably, this intuition ends up being wrong, and as shown by the previous corollary, the two $\tilde{\Omega}(\log p)$ factors can be made to add rather than multiply.

3.3 Deterministic Online Parallel Paging

In this section, we give a deterministic parallel-paging algorithm, called DET-PAR, that achieves competitive ratio $O(\log p)$ for makespan. The basic idea behind our algorithm is to derandomize the construction from the previous subsection, by first formalizing the necessary set of properties that the parallel-paging algorithm must satisfy in order to for it to deterministically achieve the same results as the randomized algorithm RAND-PAR, and by then showing how to achieve those properties deterministically.

As before, we will break our parallel-paging algorithm into ***phases*** such that the number of processors that are active at the end of each phase is half as large as the number of processors that are active at the beginning of each phase. Within each phase Q, define p_Q to be the number of processors active at the end of the phase, and define $b_Q = k/p_Q$ to be the ***base height*** for the phase.

A parallel-paging algorithm using $O(k)$ memory is ***well-rounded*** if two properties hold. The first is that, at any given moment in any given phase Q, every processor that is active is currently allocated a box with height at least b_Q. The second is that, at any given moment in time t, for any given processor x that is

still active, and for any given box height $z \geq b_Q$, at least one of the following holds:

- we are within the final $O(z^2 s/b_Q \log p)$ time steps of the current phase;
- we are within the final $O(z^2 s/b_Q \log p)$ time steps of x's life;
- x is currently allocated a box with height at least z;
- x will be allocated a new box with height at least z within the next $O(z^2 s/b_Q \log p)$ time steps.

A key insight is that, if an algorithm is well-rounded, then we can analyze the algorithm's makespan in a similar way to how we analyzed RAND-PAR in the previous subsection.

Lemma 5. *Any well-rounded parallel-paging algorithm $\mathcal{A}$ is $O(\log p)$-competitive for makespan.*

Proof. For each processor x, let σ_x be the box sequence that the optimal parallel-paging algorithm OPT allocates to x. For each box in σ_x (we refer to these as the ***OPT-boxes***), there is some set S of paging requests that OPT processes with that box. We say that $\mathcal{A}$ has ***completed*** an OPT-box once $\mathcal{A}$ has completed all of the paging requests associated with that box. We say that processor x is ***working*** on an OPT-box if some partial subset of the paging requests associated with the box are complete.

Note that if a processor x is working on an OPT-box of some height z, and if the processor spends sz steps with a memory of size z or larger, then the processor is guaranteed to complete the OPT-Box. One consequence of this is that, if an algorithm $\mathcal{A}$ is well-rounded, then the maximum amount of time that it can spend working on any given OPT-box of some height z is at most

$$O(z^2 s/b_Q \log p + sz). \tag{1}$$

We will make use of this fact later in the proof.

Say that a processor is ***time-efficient*** during a given time step of a phase Q if, during that time step, the processor works exclusively on OPT-boxes with heights b_Q or smaller. Say that a time step is ***time-efficient*** if at least half of the remaining processors are time-efficient during that time step. Say that a phase is ***time-efficient*** if for at least half of the time steps in the phase are time efficient.

We will now argue that each time-efficient phase Q can take time at most $8T_{OPT}$. Since $\mathcal{A}$ is well-rounded, it must always allocate every processor a box with height at least b_Q. It follows that, whenever a processor x is working on an OPT-box with height b_Q or smaller, the processor is guaranteed to complete that box within the same amount of time that OPT completed it. Thus each processor x can be time-efficient during at most T_{OPT} time steps in Q. If $2p_Q$ is the number of processors active at the beginning of Q, then each time-efficient time step in Q requires at least a $p_Q/2$ of them to be time-efficient. The number of time-efficient time steps in Q is therefore at most $4T_{OPT}$. Since Q is itself time-efficient, at least half of its time steps must be time-efficient. Thus the length of Q is at most $8T_{OPT}$.

The total length of all time-efficient phases is therefore at most $8T_{OPT} \log p$. To complete the proof, we turn our attention to the phases Q that are not time-efficient.

For each processor x, and each time-inefficient phase Q, let $I_{x,Q}$ be the total memory impact across all OPT-boxes that x works on during Q. A key claim is that, if x is time-inefficient for $s_{x,Q}$ steps of some phase Q, then

$$s_{x,Q} \leq \frac{I_{x,Q} \log p}{b_Q}. \tag{2}$$

We can prove (2) using (1). In particular, for each OPT-box of some height z that x works on, it spends time at most

$$\begin{cases} 0 & \text{if } z < b_Q \\ O(z^2 s/b_Q \log p) + hs & \text{if } z \geq b_Q \end{cases} = O(z^2 s/b_Q \log p)$$

on that box. On the other hand, that same box contributes $z^2 s$ to $I_{x,Q}$, hence (2).

We now use (2) to relate the length of each time-inefficient phase Q to the sum $\sum_x I_{x,Q}$. During any time-inefficient phase Q of some length t_Q, we have that $\sum_x s_{x,Q} = \Theta(p_Q t_Q)$. It follows by (2) that

$$p_Q t_Q = O\left(\sum_x \frac{I_{x,Q} \log p}{b_Q}\right),$$

which, using that $b_Q = k/p_Q$, means that

$$t_Q = O\left(\sum_x \frac{I_{x,Q} \log p}{k}\right).$$

Summing over all time-inefficient phases, we get that their total length is at most

$$O\left(\sum_{x,Q} \frac{I_{x,Q} \log p}{k}\right).$$

Now let I_x be the total memory impact of all boxes in OPT's box sequence σ_x for processor x. Then,

$$\sum_{x,Q} I_{x,Q}/k \leq \sum_x I_x/k + \text{poly}(pk),$$

where the final term accounts for the fact that adjacent phases may overlap on up to p boxes that they work on. Notice, however, that $\sum_x I_x/k \leq T_{OPT}$. So, assuming that T_{OPT} is sufficiently large as a function of p and k,

$$\sum_{x,Q} I_{x,q}/k \leq T_{OPT} + \text{poly}(pk) = O(T_{OPT}).$$

Since the total length of $\mathcal{A}$'s time-inefficient phases is $O(\sum_{x,Q} I_{x,Q} \log p/k)$, the proof is complete. □

Next we construct a well-rounded parallel-paging algorithm.

Lemma 6. *There exists a deterministic well-rounded parallel-paging algorithm using $O(k)$ memory.*

Proof. Without loss of generality, we can focus on the task of constructing a single phase Q in which the number of processors falls from $2p_Q$ to p_Q.

Since we are only interested in a single phase, we will simplify the discussion by using p to denote p_Q and b to denote b_Q.

Our new task is the following. We wish to assign boxes with heights $b, 2b, 4b, 8b, \ldots, pb = k$ to p processors so that, at any given moment, the total height of all assigned boxes is at most $O(pb)$. Once a box of height z is assigned, it sticks around for time z. (Note that we can also feel free to allocate multiple boxes to a given processor if we want, while still assuming that OPT only allocates one at a time.) In order so that our algorithm is well-rounded, we

need two properties: (1) that every processor is always assigned a box (this is trivial, since we can use $pb = k$ extra memory to give out boxes of size b to whomever needs them); and (2) that, for each box height $z \geq b$, for each processor x, and for any point in time, x is guaranteed to be allocated a box of height $\geq z$ at some point within the next $O(z^2 s/b \log p)$ time steps (unless, of course, x is currently allocated such a box). We will ignore the first property, since it is trivial to achieve, and focus on achieving the second.

The boxes of heights $z > k/\log p = pb/\log p$ are easy to handle. Indeed, for each such height z, we only need a box of height z every

$$sz^2(\log p)/b > s(pb/\log p)z(\log p)/b = szp$$

time steps. So we can just allocate one box of height z at a time, and cycle through which processor is allocated that box. The boxes of heights $z \geq k/\log p$ therefore contribute at most $k + k/2 + \cdots + k/\log p = O(k)$ total allocated height at any given moment.

The boxes of heights $z \leq k/\log p = pb/\log p$ require a different approach. We cannot get away with allocating only one box of each height at a time. For each of the $O(\log p)$ heights $z \in \{b, 2b, 4b, \ldots, pb/\log p\}$, we allocate $k/\log p$ memory to boxes of that height z. We refer to this memory as the ***z-strip of memory***.

We use the z-strip of memory as follows: using the fact that $z \leq k/\log p$, we allocate $\frac{k}{z \log p}$ boxes of height z at a time, so that the total time needed to allocate p boxes is

$$\frac{spz}{k/(z \log p)} = \frac{sz^2 \log p}{b}.$$

The boxes allocated in the z-strip are assigned in a round-robin fashion to the p processors, so each processor gets a box of height z every $sz^2/b \log p$ time steps.

In summary, it is possible to allocate $O(k)$ memory at a time while ensuring that for each box size $z \in \{b, 2b, 4b, \ldots, k\}$, each processor gets a box of size z every $sz^2/b \log p$ time steps. This completes the proof of the lemma. □

We define DET-PAR to be the deterministic well-rounded algorithm from the previous lemma. We remark that DET-PAR is ***oblivious***, meaning that it does not adapt to the specific access sequence of processor: the only information that it uses is how many processors are present at any given moment.

Combining the previous two lemmas, we obtain the following:

Theorem 3. DET-PAR *is a deterministic parallel-paging algorithm that uses a factor of $O(1)$ resource augmentation and that achieves competitive ratio $O(\log p)$ for makespan.*

Connections to green paging, and other implications. We conclude the section by proving an interesting relationship between arbitrary well-rounded paging algorithms and the green-paging problem. This relationship will then allow us to obtain several interesting corollaries.

Call an algorithm ***balanced*** if (1) the parallel-paging algorithm always allocates at least a constant fraction of memory to processors; and (2) within each phase Q, the total amount of memory impact allocated to each of the (remaining) processors is always equal up to $\pm\text{poly}(pk)$. We now observe that any balanced well-rounded algorithm is necessarily green:

Lemma 7. *Any well-rounded balanced parallel paging algorithm $\mathcal{A}$ must have the following property: the box sequence allocated by $\mathcal{A}$ to a given processor x during a given phase Q is $O(\log p)$-competitive for green paging (with $O(1)$ resource augmentation).*

Proof. Consider a processor x during a phase Q, and assume that the length T_Q of Q is at least $\text{poly}(pk)$. Let us compare to a paging algorithm OPT that assigns each processor an optimal green-paging box sequence during phase Q (rather than optimizing for parallel paging). Because $\mathcal{A}$ is well-rounded, the total amount of time that x spends working on a given OPT-box of some height $z \in \{k/p_Q, 2k/p_Q, \ldots, k\}$ is at most $O(z^2 s/b_Q \log p + sz) = O(z^2 s/b_Q \log p)$. On the other hand, the box of height z incurs a memory impact of $\Theta(z^2 s)$. Thus, if we let $I_{x,Q}$ be the total memory impact across all OPT-boxes that x works on during Q, then

$$I_{x,Q} = \Omega(b_Q T_Q / \log p).$$

Since $\mathcal{A}$ is balanced, the total amount of memory impact ϕ that x incurs in phase Q is $O(T_Q k/p_Q)$. Thus

$$T_Q = \Omega\left(\phi p_Q/k\right).$$

Chaining together the inequalities,

$$I_{x,Q} = \Omega\left(\frac{b_Q \phi p_Q}{k \log p}\right) = \Omega(\phi/\log p).$$

This implies that the box sequence allocated by $\mathcal{A}$ to x during phase Q is $O(\log p)$-competitive for green paging. □

As an immediate consequence, we get a deterministic analogue of Corollary 1.

Corollary 2. DET-PAR *is a deterministic parallel-paging algorithm that assigns boxes to each processor using a black-box $\Theta(\log p)$-competitive green paging algorithm, that uses a factor of $O(1)$ resource augmentation, and that achieves competitive ratio $O(\log p)$ for makespan.*

A second corollary of Lemma 7 is that we can analyze the competitive ratio that DET-PAR incurs with respect to *average completion time*. Indeed, [2] showed that, in order for a deterministic parallel-paging algorithm (with $1 + \Theta(1)$ resource augmentation) to achieve an optimal competitive ratio of $O(\log p)$ for average completion time, it needs only to satisfy two requirements: (1) that it is balanced, as defined above; and (2) that it is green, meaning that within each phase it allocates memory to processors using an $O(\log p)$-competitive green-paging algorithm. On the other hand, Lemma 7 tells us that, if a parallel-paging algorithm is well-rounded and balanced, then green-ness is *automatic*. Thus we can conclude that DET-PAR achieves the same (optimal) competitive ratio for average completion time as it achieves for makespan.

Corollary 3. DET-PAR *achieves a competitive ratio of $O(\log p)$ for average-completion-time using $O(1)$ resource augmentation.*

4 A LOWER BOUND ON BLACK-BOX APPLICATIONS OF GREEN PAGING

In [2], the authors show how "good" green paging algorithms can be transformed into parallel paging algorithms with good performance under several metrics. In particular, starting from *any* green

paging algorithm that is c-competitive (including an algorithm e.g. 1-competitive because it is offline, or because it is only required to service "easy" request sequences) one can obtain a parallel paging algorithm with comparable resource augmentation that is $O(c)$-competitive for mean completion time. The black-box transformation involves using the green paging algorithm for each of the p sequences in the parallel paging problem. Without loss of generality, this produces for each processor a sequence of boxes. Any parallel paging algorithm that packs these boxes *fairly* (with no request sequence having had, at any point in time, more than $O(1)$ times the memory impact of any other uncompleted sequence) and *efficiently* (with boxes taking an $\Omega(1)$ fraction of the "available" impact) yields the requisite result.

This result does not apply to makespan. In general, to complete a fraction greater than $(1-\varepsilon)$ of all sequences, the black-box transformation yields a competitive ratio with an upper bound no better than $O(c \log 1/\varepsilon)$. Therefore the additional overhead is not a constant if $\varepsilon = o(1)$; in particular, for makespan $\varepsilon = 1/n$, yielding a multiplicative $\approx \log n$ overhead on top of that stemming from the suboptimality of online green paging. We now show that a significant portion of this overhead is inherent to using such a black-box approach: roughly speaking, if each processor is allotted *any* optimally or almost optimally green memory profile, then makespan on some sequences is inevitably suboptimal by a factor $\tilde{\Omega}(\log n)$.

A deeper look at green paging. To state more formally the lower bound we must first make the definition of green paging more precise. In general, the green paging problem is well defined only when we specify the maximum and minimum memory to service a sequence. Intuitively, the closer the two thresholds are, the fewer choices the algorithm has, and thus the more easily it can be optimal or almost optimal. Indeed, the competitive ratio achievable by a deterministic online green paging algorithm with constant resource augmentation is logarithmic in the ratio between the two thresholds [2]. Crucially, if the ratio is small because the minimum threshold is high, an algorithm can achieve better *relative* performance not because it can have lower impact, but because there are fewer possible choices – in other words, because the optimal strategy is forced to have higher impact.

When using green paging black-box to allocate memory to each sequence in a parallel paging algorithm, the threshold for the minimum memory allotted to each sequence grows over time – since when v sequences remain uncompleted, an extra factor 2 of resource augmentation allows each sequence to receive k/v memory at all times. In practice, this is easily addressed (both in previous sections and in earlier work [2]) with yet another factor 2 of resource augmentation by simply "rebooting" the green paging algorithm whenever the minimum threshold doubles – so that it is always effectively running with fixed thresholds.

Note that the memory thresholds encountered when servicing one sequence might be influenced by the choices of the algorithm for other sequences. Thus, one might in principle encounter a green paging algorithm used for parallel paging that appears greener by "cheating" as follows: it allots excessive memory early on to some sequences so as to end them as soon as possible, and thus becomes *comparatively* greener on the suffixes of the remaining sequences (and in aggregate over the whole execution) – not by being intrinsically greener on those suffixes, but simply by making the lowest-impact allocations not viable. As we are trying to prove that employing "really" green paging algorithms in parallel paging is a source of a logarithmic overhead, we would like to rule out this "greenwashing".

We stress that this issue is not present if considering green paging in isolation, when (ignoring constant factors) servicing a prefix of a sequence with higher impact can never lower the impact of the remaining suffix and thus of the sequence in its entirety; so that being *greedily* green is always the most efficient strategy. This is true even if the memory thresholds can change over time, as long as the changes do not depend on the algorithm's choices. Thus, even in the context of parallel paging we would like to consider only "greedily green" allocations – which is also the only option when dealing with online algorithms, which cannot know when a sequence might end.

Thus, we refine our definition of green paging as follows:

Definition 1. *A green paging algorithm ALG is g-greedily competitive for green paging if for some g' (which might depend on the minimum and maximum memory thresholds, and on the relative overhead s of faults), for any sequence σ, it services σ incurring on any prefix π of σ an impact no larger than $g \cdot c_{OPT}(\pi) + g'$, where $c_{OPT}(\pi)$ is the minimum offline cost to service π.*

Note that this is a very general definition. A green paging online algorithm that is c-competitive is necessarily greedily c-competitive since a sequence can end at any time (and thus the algorithms in [2] are exactly of this type). But a greedily competitive algorithm is not necessarily online: it can make decisions taking the entire sequence into consideration – in fact, in the context of parallel paging base its decisions on the entirety of *all* p sequences.

Being green forces a logarithmic makespan overhead. Consider a parallel paging algorithm PAR that uses a greedily green paging algorithm GREEN as a black box. As mentioned before, we can assume without loss of generality GREEN is simply a normalized and compartmentalized memory allocation strategy, that given a sequence of requests produces a sequence of boxes and services the sequence within those boxes e.g. with LRU (which is 2-competitive with resource augmentation 2). At any time PAR can allocate a memory box for one or more sequences not under execution, it does so (prioritizing among the possible choices, if necessary, in *some* fashion), and begins servicing those sequences until the end of the respective boxes, or of the sequences. In fact, with $O(1)$ resource augmentation, we can simply assume PAR keeps *every* sequence constantly in execution by simply allocating, to any that would receive no memory, a box of capacity k/v where v is the (smallest power of 2 at least as large as the) number of surviving sequences. Note that the parallel paging algorithms obtained in [2] are exactly of this type. We can then prove:

Theorem 4. *For any $c \geq 1$, and for any arbitrarily large number of request sequences p and memory size $k \geq p$, there exist sequences $\sigma_0, \ldots, \sigma_{p-1}$ and a parallel paging algorithm OPT that can serve these sequences with memory k which has the following property.*

If PAR is greedily competitive within a factor c, then for any $\varepsilon > 0$, $s > ck$, the time necessary for PAR to complete more than a fraction $(1-\varepsilon)$ of all sequences, even with memory $K = ck$, is a factor $\Omega(1+$

$\log(1/\varepsilon)/\log\log p)$ *larger than that necessary for OPT to complete* all *sequences.*

Let us briefly examine the theorem's statement. First, if c is a constant (i.e. GREEN is $O(1)$-competitive), PAR's competitive ratio for makespan is $\Omega(\log p/\log\log p)$. Second, note the order of quantifiers in Theorem 4, and in particular how the adversarial request sequences depend (obviously) on the memory and number of processors, and on the maximum "slack" we allow the green paging algorithm – but not on the green paging algorithm itself or its parallel scheduler. Thus, the sequences are universally bad even against offline (but sufficiently green) algorithms; i.e. the suboptimality factor stems not from ignorance of future requests, but from being green. We then have:

Corollary 4. *Any parallel paging algorithm with constant resource augmentation c, exploiting black-box a c−greedily competitive green paging algorithm, has a competitive ratio* $\Omega(1 + \log p/\log\log p)$.

Proof idea. The theorem's proof is short, but requires careful coordination of many parameters; so we first try to provide an intuition of where we are going and why. For simplicity, we focus on the case $\varepsilon = 1/n$, i.e. makespan.

Our generic sequence σ_i is the concatenation of a possibly empty ***prefix*** π_i and of a ***suffix*** ψ_i. Suffixes of all sequences have the same length; each of their pages is requested only once, so they run at the same speed however little memory they are given. And since even with minimal memory suffixes are designed to consume the bulk of impact, the key to optimality lies in executing them in parallel.

Prefixes of different sequences vary in length and are designed to achieve two goals. First, any "sufficiently green" memory allocation must be an almost minimal one: i.e. GREEN can allocate only few boxes of size $\gg K/v$ when v sequences are still uncompleted. Second, it is only slightly less green to execute a prefix with the largest possible memory (the size K box) than with the minimal one (the size K/v box). OPT takes advantage of this by allocating large boxes to prefixes so as to complete each very quickly without being too wasteful; it can then execute all suffixes in parallel. PAR, on the other hand, to be green must give roughly the same memory (the minimal one) to each executing prefix; since prefixes are of different length, this prevents PAR from executing all suffixes in parallel.

Let us now have a first look at the sequences. The generic prefix π_i accesses two types of pages: ***repeaters*** and ***polluters***. Polluters are pages that are accessed only once; thus, they replace in memory potentially useful data with data that will never be accessed again. Repeaters are $k-1$ pages that we keep cycling over — except that, every so often, instead of a repeater we access a polluter. Note that these are common access patterns, and not at all pathological.

If we had no polluters, for a sufficiently large s and sufficiently long prefixes, the greenest strategy would obviously be to allocate a $\approx k$ memory and only incur a negligible fraction of misses (on the very first cycle). By adding a fraction of polluters, we can increase the miss rate incurred by this allocation so that it is just high enough to have GREEN reject it. More specifically, denoting by v the number of sequences not yet completed by GREEN at a point in time, we want roughly every v/cth page to be a polluter. Then, the minimum memory GREEN can assign to a sequence is $\approx K/v$ – forcing a miss on most accesses, for a total impact on L requests of sKL/v. With memory k or larger, every v/cth access (involving a polluter) is a miss, for a total impact slightly larger than $cskL/p$. Since GREEN must be optimal within a factor c in terms of impact, it cannot choose the large boxes (nor mid-sized boxes that incur almost as many misses as the small box, but more impact).

OPT can then complete the prefixes one at a time with boxes of size k. Once all prefixes are completed, all suffixes can be executed in parallel. Suffixes involve roughly $\alpha k^2 \log\log p$ requests, with a sufficiently large α that the suffix cost dominates the entire computation, so OPT finishes in time $O(\alpha sk^2 \log\log p)$.

We want PAR to take significantly more time, which leads us to the next insight. Prefixes are of different lengths so that the following property (approximately) always holds: PAR runs over $\approx \log p$ *eras* of roughly the same duration, with the number of uncompleted prefixes halving each era until only suffixes remain. Roughly speaking, if $\approx v$ sequences remain uncompleted during an era, then at most a fraction $1/\log p$ of them are executing prefixes and all others are executing suffixes. Our sequences will force each era to take approximately $\approx \alpha sk^2$ time, for a total time of $\approx \alpha sk^2 \log p$. This is a factor $\approx \log p/\log\log p$ larger than OPT – due to the fact that suffixes, the dominant cost in both cases, for PAR spread over $\approx \log p$ eras rather than over $\log\log p$.

How do we design sequences to make the above happen? We divide all sequences into phases, such that, in general, each alive sequence executes one phase in each era. Each phase has about $X = \Theta(\alpha k^2)$ requests. All suffixes have the same number $\Theta(\log\log p)$ of phases; so every suffix has $\Theta(\alpha k^2 \log\log p)$ requests, all to polluters.

As for prefixes, those of most sequences are empty and only $p/\log p$ sequences are ***prefixed***. The prefixed sequences are divided into roughly $\log p - \log\log p$ families $F_0, F_1, \ldots$ where sequences in a family are isomorphic. Family F_i has 2^i sequences, but each sequence in this family has $\log p - \log\log p - i$ phases. That is, the number of sequences in the family increases geometrically with i but the number of phases that the sequences in the family have decreases linearly with i. Therefore, family F_0 has 1 sequence, but is very long — it has approximately $\log n - \log\log n$ phases. On the other hand, the last prefixed family has about half the prefixed sequences, but each of these prefixes has only 1 phase.

Remember that we are designing sequences so that GREEN (and therefore PAR) is forced to choose mostly small boxes for all the sequences that are in their prefix phases. To do so, we must ensure first, that there are at least $p/2^j$ sequences in the jth phase, and second, that the pollution level of the sequences executing their prefix in the jth phase is *just* high enough — in particular, the jth prefixed phase for all sequences (where it exists) has a pollution level of approximately $\approx 2^j c/p$. Now consider PAR's execution. In the first era, all sequences are alive. $p - p/\log p$ prefix-free sequences are executing their first suffixes and the remaining $p/\log p$ sequences are executing their first prefix phase. At this point, the smallest available box size is K/p and the pollution level is high enough that GREEN should choose small boxes for all the $p/\log p$ prefixed sequences. The first phase takes $O(\alpha sk^2)$ time since each sequence always misses with small boxes. After the first phase, about half the prefixed sequences enter their suffix phases, but all sequences are

still alive. The pollution level keeps doubling in each phase over the next $\log\log n$ phases, so GREEN must keep choosing small boxes.

At this point, the prefix-free sequences start completing. However, most of the prefixed sequences are now in their suffix phases — in particular, only a $1/\log p$ fraction of the originally prefixed sequences are still executing their prefixes, since suffixes last for about $\log\log p$ phases, and during each of those the number of surviving prefixes roughly halved. Also, the pollution level has increased enough by this time (for prefixed phases) that GREEN must keep choosing smallest available boxes (which are size about $K/(p/\log p)$ by now since only $p/\log p$ sequences are now alive. By carefully picking constants, we keep increasing the pollution level just enough as sequences complete so that GREEN keeps picking small boxes and each phase keeps taking $\approx \alpha sk^2$ time to complete. Since there are a total of $\Theta(\log p)$ phases, PAR takes $\Theta(\alpha sk^2 \log p)$ time to finish all sequences.

Let us go back to OPT now. It executes all the prefixed sequences one at a time giving it a box of size k. Therefore, its miss rate in the jth phase is about $c2^j/p$, and the jth phase only takes time $O(\alpha 2^j sk^2/p)$. Adding over all sequences, it can finish all the prefixes in only $O(\alpha sk^2)$ time. It can then execute all suffixes together and complete all sequences in $O(\alpha sk^2 \log\log p)$ time.

The actual proof requires us to carefully pick constants so that it all works exactly as needed. The details can be found in the appendix.

5 CONCLUSIONS

We settled the problem of minimizing makespan for online parallel paging when p processors access distinct sets of pages, with a single algorithm that simultaneously achieves the optimal deterministic competitive ratio of $O(\log p)$ for both makespan and mean completion time.

An obvious open question is whether our result can be improved (for either metric) via randomization; we conjecture this is not the case. Another direction for future research is to consider scenarios where the p sequences running on different processors can share pages (in the resource-augmentation framework). This problem appears to be difficult to solve in the general case; however, it may be tractable under limited conditions of sharing. An even more difficult problem would be that of sharing a cache among processors executing a parallel program with inter-thread dependencies, so that paging decisions interact with scheduling decisions.

Acknowledgments

The authors gratefully acknowledge support from the following grants.

Agrawal was supported by the Department of Computer Science and Engineering at Washington University in St. Louis as well as the NSF grants CCF-2106699, CCF-1733873, and SPX-1725647.

Bender was supported by NSF grants CCF-2118832, CCF-2106827, CSR-1763680, CCF-1716252, CNS-1938709, and CCF-1725543.

Das was supported by the Canada Research Chairs Programme and NSERC Discovery Grants.

Kuszmaul was funded by an NSF GRFP fellowship, a Fannie and John Hertz Fellowship; the research was also funded by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Peserico was supported, in part, by the University of Padova project "Internet of Things" (MIUR grant "Dipartimenti di Eccellenza" L. 232/2016).

Scquizzato was supported, in part, by the University of Padova under grant BIRD197859/19.

REFERENCES

[1] Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. Brief announcement: Green paging and parallel paging. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 493–495, 2020.

[2] Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. Tight bounds for parallel paging and green paging. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3022–3041, 2021.

[3] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. Application-controlled paging for a shared cache. *SIAM Journal on Computing*, 29(4):1290–1303, 2000.

[4] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[5] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[6] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In *Proceedings of the USENIX Summer 1994 Technical Conference (USTC)*, pages 171–182, 1994.

[7] Marek Chrobak. SIGACT news online algorithms column 17. *SIGACT News*, 41(4):114–121, 2010.

[8] Rathish Das. *Algorithmic Foundation of Parallel Paging and Scheduling under Memory Constraints*. PhD thesis, State University of New York at Stony Brook, 2021.

[9] Rathish Das, Kunal Agrawal, Michael A Bender, Jonathan Berry, Benjamin Moseley, and Cynthia A Phillips. How to manage high-bandwidth memory automatically. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 187–199, 2020.

[10] Daniel DeLayo, Kenny Zhang, Kunal Agrawal, Michael A Bender, Jonathan Berry, Rathish Das, Benjamin Moseley, and Cynthia A Phillips. Automatic hbm management: Models and algorithms. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2022.

[11] Esteban Feuerstein and Alejandro Strejilevich de Loma. On-line multi-threaded paging. *Algorithmica*, 32(1):36–60, 2002.

[12] Amos Fiat and Anna R. Karlin. Randomized and multipointer paging with locality of reference. In *Proceedings of the 27th annual ACM Symposium on Theory of Computing (STOC)*, pages 626–634, 1995.

[13] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Elastic caching. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 143–156, 2019.

[14] Avinatan Hassidim. Cache replacement policies for multicore processors. In *Proceedings of 1st Symposium on Innovations in Computer Science (ICS)*, pages 501–509, 2010.

[15] Shahin Kamali and Helen Xu. Beyond worst-case analysis of multicore caching strategies. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 1–15, 2021.

[16] Anil Kumar Katti and Vijaya Ramachandran. Competitive cache replacement strategies for shared cache environments. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 215–226, 2012.

[17] Ravi Kumar, Manish Purohit, Zoya Svitkina, and Erik Vee. Interleaved caching with access graphs. In *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1846–1858, 2020.

[18] Alejandro López-Ortiz and Alejandro Salinger. Minimizing cache usage in paging. In *Proceedings of the 10th Workshop on Approximation and Online Algorithms (WAOA)*, pages 145–158, 2012.

[19] Alejandro López-Ortiz and Alejandro Salinger. Paging for multi-core shared caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science conference (ITCS)*, pages 113–127, 2012.

[20] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
[21] Ishai Menache and Mohit Singh. Online caching with convex costs. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 46–54, 2015.
[22] Enoch Peserico. Paging with dynamic memory capacity. In *Proceedings of the 36th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 56:1–56:18, 2019.
[23] Michele Scquizzato. *Paging on Complex Architectures.* PhD thesis, University of Padova, 2013.
[24] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

APPENDIX

Let us now provide all the missing details of the proof of Theorem 4; in particular considering all the constants. Assume without loss of generality that $p = 2^{\ell+1} - 1$, $\varepsilon = 2^{-t}$, and $k = p2^{a-1}$ for some positive integers ℓ, t, a. In addition, assume $\gamma = 2k\alpha$. Each suffix has $4\log\ell$ phases, each in turn consisting of $(k-1)\gamma$ requests, all for new pages for a total number of requests equal to $4\gamma(k-1)\log\ell$.

The prefixed sequences are divided into $\ell - \log\ell$ families $F_0, \ldots, F_{\ell-\log\ell}$, with F_i containing 2^i isomorphic sequences (i.e. identical in structure but requesting different pages). The generic sequence in family F_i has $\ell - \log\ell - i + 1$ phases, namely, $\sigma_i^0, \ldots, \sigma_i^{\ell-\log\ell-i}$ — as i increases, the number of phases that the sequences in the family contain decreases. That is, family F_0 has only one sequence, but this sequence is long, while family $F_{\ell-\log\ell}$ has almost $p/\log p$ sequences, but the sequences have only one phase.

For all i, all σ_i^j are isomorphic to a single subsequence σ^j. The latter is formed by $\gamma = 2k\alpha$ cycles of requesets for the same $k-1$ repeater pages $\rho_1, \ldots, \rho_{k-1}$, replacing every $n^j = p/2^j$th request with one to a polluter. Note that the level of pollution increases as the computation advances. Thus, phase 0 sequence σ^0 for all prefixes sequences (for which it exists) requests a polluter page every p requests, σ^1 every $p/2$ requests, and $\sigma^{\ell-\log\ell}$ – the last subsequence of the longest prefix – every $p2^{-(\ell-\log\ell)} = \ell$ requests.

Lemma 8. *OPT can complete all p sequences in $O(sk^2\log\ell) = O(\alpha sk^2\log\log p)$ time.*

Proof. Phase j of any sequence (for which it exists), σ^j, can then be completed in isolation by first bringing the cycle's $k-1$ pages into memory (at cost $s(k-1)$), so that the subsequent $\gamma - 1$ cycles incur at most one fault every n_j requests. Remembering that $s \geq \gamma \geq p$, then σ^j can be completed in time no larger than

$$\begin{aligned} T^j &= \gamma(k-1) + (s-1)(k-1) + (s-1)(k-1)\gamma/n_j \\ &\leq k(\gamma + s + s\gamma/n_j) \leq sk(2^j\gamma/p + 2) = O(2^j sk\gamma/p). \end{aligned}$$

Recall that$2^{\ell-\log\ell-j}$ sequences have σ_j in the first place (all prefixed sequences have σ^0, but only about half have σ^1, and so on). Therefore, the total time for prefix j to complete for all sequences is $2^{\ell-\log\ell-j}T_j$. Also, all suffixes can execute in parallel in time $4\log\ell s(k-1)\gamma$ since they don't reuse any pages. Therefore, the total cost to complete all prefixes one after the other, followed by all suffixes in parallel, is at most

$$\begin{aligned} T_{OPT} &= \sum_{j=0}^{\ell-\log\ell} 2^{\ell-\log\ell-j}T_j + 4\log\ell s(k-1)\gamma \\ &\leq (\ell - \log\ell + 1)(p/\ell)sk(\gamma/p) + 4sk + 4\log\ell s(k-1)\gamma \\ &= O(sk\gamma\log\ell). \end{aligned}$$ □

Lemma 9. *PAR takes time $\Omega(\log\ell + \log(1/\varepsilon)sk\gamma)$ to complete a fraction at least $1-\varepsilon$ of all sequences.*

Proof. Intuitively, in PAR, there are always at least $\approx \ell$ times as many active suffixes as prefixes; this guarantees that any "sufficiently green" memory allocation must have service a fraction bounded away from 0 of each prefix's requests with minimal memory, causing each phase to take time $\Omega(sk\gamma)$ time.

More formally, say $S_j = js(k-1)\gamma/4$. We can easily prove by simultaneous induction on j that:

(1) No sequence serviced by PAR completes its $(j-1)^{th}$ stage and enters its j^{th} before S_j.
(2) There are at least $p2^{-j}$ *uncompleted* sequences at all times before S_{j+1}.
(3) For any sequence, at any given point in time before S_{j+1}, PAR must have serviced at least half the requests with the minimum available memory at that time.
(4) For any sequence, at any given point in time before S_{j+1}, PAR has incurred at least 1 page fault every 4 requests.

Note that (1) is trivially true for $j = 0$, and (for any given j) (2) follows immediately from (1), since there are at least $p2^{-j}$ sequences with at least j stages.

(3) follows from (2) since, as long as $u_j = 4c^2n_j$ sequences remain unfinished, with memory $K \leq ck$ GREEN must service at least half the requests of any prefix in F_i with blocks of capacity at most $2ck/u_i$. To see how this is the case, note that the minimum-cost green strategy is the one servicing the sequence with the strictly minimal blocks of size (at most) k/u_i, with a per-request cost of at most $sk/u_i \leq sck/u_i$. Larger blocks smaller than $k/2$ do not reduce faults by more than a factor of two, since cycles involve $k-1$ distinct pages. Blocks of size at least $k/2$ can incur fewer faults, but at least $1/n_j$ times as many as with minimal memory (due to new page requests); then, the per-request cost of any block with capacity at least $k/2$ is still at least $(k/2)(1 + (s-1)/n_j) \geq sk/(2n_j) \geq (2c)sck/u_j$, and thus no more than half of all requests can be serviced via such large blocks.

Then (4) immediately follows, and from it (1) follows for $j+1$ since $S_{j+1} - S_j = s(k-1)\gamma/4$ and every stage sports $(k-1)\gamma$ requests, at least half of which incur faults and thus take time s to service. □

Theorem 4 is implied from Lemmas 8 and 9.