Decentralized Online Scheduling Of Malleable NP-hard Jobs

Peter Sanders[0000−0003−3330−9349] and Dominik Schreiber[0000−0002−4185−1851]

Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
{sanders,dominik.schreiber}@kit.edu

Abstract. In this work, we address an online job scheduling problem in a large distributed computing environment. Each job has a priority and a demand of resources, takes an unknown amount of time, and is malleable, i.e., the number of allotted workers can fluctuate during its execution. We subdivide the problem into (a) efficiently determining the fair share of all current jobs and (b) assigning each job to a fair subset of all workers. Our approach is fully decentralized, uses lightweight communication, and arranges each job as a binary tree of workers which can grow and shrink as necessary. Using the NP-complete problem of propositional satisfiability (SAT) as a case study, we experimentally show on up to 128 machines (6144 cores) that our approach leads to near-optimal utilization, imposes minimal computational overhead, and performs fair scheduling of incoming jobs within few milliseconds.

Keywords: Malleable job scheduling · Load balancing · SAT

1 Introduction

A parallel task is called *malleable* if it can handle a fluctuating number of workers during its execution. In the field of distributed computing, malleability has long been recognized as a powerful paradigm which opens up vast possibilities for fair and flexible scheduling and load balancing [17, 13]. While most previous research on malleable job scheduling has steered towards iterative data-driven applications, we want to shed light on malleability in a very different context, namely for NP-hard tasks with unknown processing times. For instance, the problem of propositional satisfiability (SAT) is of high practical relevance and an important building block for many applications including automated planning [25], formal verification [18], and cryptography [20]. We consider malleable scheduling of such tasks highly promising: On the one hand, the description of a job can be relatively small even for very difficult problems, and the successful approach of employing many combinatorial search strategies in parallel can be made malleable without redistribution of data [26]. On the other hand, the limited scalability of these parallel algorithms calls for careful distribution of computational resources. We believe that a cloud-like on-demand system for resolving NP-hard problems has the potential to drastically improve efficiency and productivity for many organizations and environments. Using malleable job

scheduling, we believe that it is possible to schedule new jobs within a few milliseconds, resolve trivial jobs in a fraction of second, and rapidly resize more difficult jobs to a fair share of all resources – as far as the job can make efficient use of these resources.

To meet these objectives, we propose a fully decentralized scheduling approach which guarantees fast, fair, and bottleneck-free scheduling of resources without any knowledge on processing times. In previous work [26], we briefly outlined initial algorithms for this purpose while focusing on our award-winning scalable SAT solving engine which we embedded into our system. In this work, we shed more light on our earlier scheduling algorithms and proceed to propose significant improvements both in theory and in practice.

We address two subproblems. The first problem is to let m workers compute a fair number of workers v_j for each active job j, accounting for its priority and maximum demand, which result in optimal system utilization. In previous work [26] we outlined this problem and employed a black box algorithm to solve it. The second problem is to assign v_j workers to each job j while keeping the assignment as stable as possible over time. Previously [26], we proposed to arrange each job j as a binary tree of workers which grows and shrinks depending on v_j, and we described and implemented a worker assignment strategy which routes request messages randomly through the system. This protocol requires suboptimal utilization or can lead to high scheduling latencies otherwise.

In this work, we describe fully distributed and bottleneck-free algorithms for both problems with appealing asymptotic properties (solving the first problem in $\mathcal{O}(\log m)$ span and $\mathcal{O}(m \log m)$ work and the second problem in $\mathcal{O}(\log m)$ span and $\mathcal{O}(m)$ work). Our new approaches can achieve consistent optimal utilization. Furthermore, we introduce new measures to preferably reuse existing (suspended) workers for a certain job rather than initializing new workers. We then present our scheduling platform *Mallob* which features simplified yet highly practical implementations of our approaches. Experiments on up to 128 nodes (6144 cores) show that our system leads to near-optimal utilization and schedules jobs with a fair share of resources within tens of milliseconds. We consider these results of both theoretic and practical nature as promising contributions towards resolving NP-hard tasks in large-scale environments more efficiently.

2 Preliminaries

We now establish important preliminaries and discuss work related to ours.

2.1 Malleable Job Scheduling

We use the following definitions [10]: A *rigid* task requires a fixed number of workers. A *moldable* task can be scaled to a number of workers at the time of its scheduling but then remains rigid. Finally, a *malleable* task is able to adapt to a fluctuating number of workers *during* its execution. Malleability can be a highly desirable property of tasks because it allows to balance tasks continuously

to warrant fair and optimal utilization of the system at hand [17]. For instance, if an easy job arrives in a fully utilized system, malleable scheduling allows to shrink an active job in order to schedule the new job and significantly decrease its response time. Due to the appeal of malleable job scheduling, there has been ongoing research to exploit malleability, from shared-memory systems [13] to HPC environments [8, 27], even to improve energy efficiency [24].

The effort required to transform a moldable (or rigid) algorithm into a malleable algorithm depends on the application at hand. For iterative data-driven applications, redistribution of data is necessary if a task is expanded or shrunk [8]. In contrast, we demonstrated in previous work [26] for the use case of propositional satisfiability (SAT) that basic malleability is simple to achieve if the parallel algorithm is composed of many independent search strategies: The abrupt suspension and/or termination of individual workers can imply the loss of progress, but preserves completeness. Moreover, if workers periodically exchange knowledge, the progress made on a worker can benefit the job even if the worker is removed. For these reasons, we have not yet considered the full migration of application processes as is done in adaptive middlewares [16, 8] but instead hold the application itself responsible to react to workers being added or removed.

Most prior approaches rely on known processing times of jobs and on an accurate model for their execution time relative to the degree of parallelism [5, 23] whereas we do not rely on such knowledge. Secondly, most approaches employ a centralized scheduler, which implies a potential bottleneck and a single point of failure. Our approach is fully decentralized and uses a small part of each processes' CPU time to perform distributed scheduling, which also opens up the possibility to add most general fault-tolerance to our work in the future. For instance, this may include continuing to schedule and process jobs correctly even in case of network-partitioning faults [2], i.e., failures where sub-networks in the distributed environment are disconnected from each another. Other important aspects of fault-tolerance include mitigation of simple node failures (i.e., a machine suddenly goes out of service) and of Byzantine failures [6] (i.e., a machine exhibits arbitrary behavior, potentially due to a malicious attack).

2.2 Scalable SAT Solving

The propositional satisfiability (SAT) problem poses the question whether a given propositional formula $F = \bigwedge_{i=1}^{k} \left(\bigvee_{j=1}^{c_i} l_{i,j} \right)$ is satisfiable, i.e., whether there is an assignment to all Boolean variables in F such that F evaluates to `true`. SAT is the archetypical NP-complete problem [7] and, as such, a notoriously difficult problem to solve. SAT solving is a crucial building block for a manifold of applications such as automated planning [25], formal verification [18], and cryptography [20]. State-of-the-art SAT solvers are highly optimized: The most popular algorithm named Conflict-Driven Clause Learning (CDCL) performs depth-first search on the space of possible assignments, backtracks and restarts its search frequently, and derives redundant *conflict clauses* when encountering a dead end in its search [19]. As these clauses prune search space

and can help to derive unsatisfiability, remembering important derived clauses is crucial for modern SAT solvers' performance [3].

The empirically best performing approach to parallel SAT solving is a so-called *portfolio* of different solver configurations [14] which all work on the original problem and periodically exchange learned clauses. In previous work, we presented a highly competitive portfolio solver with clause sharing [26] and demonstrated that careful periodic clause sharing can lead to respectable speedups for thousands of cores. The malleable environment of this solver is the system which we present here. Other recent works on decentralized SAT solving [15, 21] rely on a different parallelization which generates many independent subproblems and tends to be outperformed by parallel portfolios for most practical inputs [11].

2.3 Problem Statement

We consider a homogeneous computing environment with a number of interconnected machines on which a total of m *processing elements*, or PEs in short, are distributed. Each PE has a *rank* $x \in \{0, \ldots, m-1\}$ and runs exclusively on $c \geq 1$ cores of its local machine. PEs can only communicate via message passing.

Jobs are introduced over an interface connecting to some of the PEs. Each job j has a job description, a priority $p_j \in \mathbb{R}^+$, a *demand* $d_j \in \mathbb{N}^+$, and a *budget* b_j (in terms of wallclock time or CPU time). If a PE participates in processing a job j, it runs an execution environment of j named a *worker*. A job's demand d_j indicates the maximum number of parallel workers it can currently employ: d_j is initialized to 1 and can then be adjusted by the job after an initial worker has been scheduled. A job's priority p_j may be set, e.g., depending on who submitted j and on how important they deem j relative to an average job of theirs. In a simple setting where all jobs are equally important, assume $p_j = 1\ \forall j$. A job is cancelled if it spends its budget b_j before finishing. We assume for the active jobs J in the system that the number $n = |J|$ of active jobs is no higher than m and that each PE employs at most one worker at any given time. However, a PE can preempt its current worker, run a worker of another job, and possibly resume the former worker at a later point.

Let T_j be the set of active workers of $j \in J$. We call $v_j := |T_j|$ the *volume* of j. Our aim is to continuously assign each $j \in J$ to a set T_j of PEs subject to:

(C1) (Optimal utilization) Either all job demands are fully met or all m PEs are working on a job: $(\forall j \in J : v_j = d_j) \ \vee \ \sum_{j \in J} v_j = m$.

(C2) (Individual job constraints) Each job must have at least one worker and is limited to d_j workers: $\forall j \in J : 1 \leq v_j \leq d_j$.

(C3) (Fairness) Resources allotted to each job j scale proportionally with p_j except if prevented by C2:
For each $j, j' \in J$ with $p_j \geq p_{j'}$, there are *fair assignments* $\omega, \omega' \in \mathbb{R}^+$ with $\omega/\omega' = p_j/p_{j'}$ and some $0 \leq \varepsilon \leq 1$ such that $v_j = \min(d_j, \max(1, \lfloor \omega + \varepsilon \rfloor))$ and $v_{j'} = \min(d_{j'}, \max(1, \lfloor \omega' \rfloor))$.

Due to rounding, in C3 we allow for job volumes to deviate by a single unit (see $\varepsilon \leq 1$) from a fair distribution as long as the job of higher priority is favored.

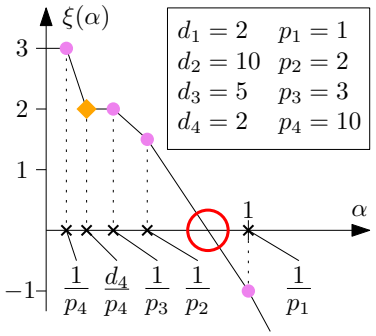


Fig. 1. Volume calculation example with four jobs and $m = 7$. Five of the eight points where $\xi(\alpha)$ is evaluated are depicted, three more (d_3/p_3, d_1/p_1, and d_2/p_2) are omitted. In the interval $[1/p_2, 1/p_1]$ we find $\alpha_0 = 0.8$ (red circle) where $\xi(\alpha) = 0$. Job 4 is capped at its demand ($v_4 = 2$) and job 1 is raised to $v_1 = 1$. The real-valued shares $\alpha_0 p_2 = 1.6$ and $\alpha_0 p_3 = 2.4$ are rounded to $v_2 = 1$ and $v_3 = 3$ as job 3 has the higher priority.

3 Approach

We subdivide the problem at hand into two subproblems: First, find *fair volumes* v_j for all currently active jobs $j \in J$ subject to C1–C3. Secondly, identify pairwise disjoint sets T_j with $|T_j| = v_j$ for each $j \in J$. In this section, we present fully decentralized and highly scalable algorithms for both subproblems. In Section 4.1 we describe how our practical implementation differs from these algorithms.

To assess our algorithms, we consider two important measures from parallel processing. Given a distributed algorithm, consider the dependency graph which is induced by the necessary communication among all PEs. The *span* (or *depth*) of the algorithm is the length of a critical path through this graph. The *local work* is the complexity of local computations summed up over all PEs.

3.1 Calculation of Fair Volumes

Given jobs J with individual priorities and demands, we want to find a fair volume v_j for each job j such that constraints C1–C3 are met. Volumes are recomputed periodically taking into account new jobs, departing jobs, and changed demands. In the following, assume that each job has a single worker which represents this (and only this) job. We elaborate on these representants in Section 3.2.

We defined our problem such that $n = |J| \leq m$. Similarly, we assume $\sum_{j \in J} d_j > m$ since otherwise we can trivially set $v_j = d_j$ for all jobs j. Assuming real-valued job volumes for now, we can observe that for any parameter $\alpha \geq 0$, constraints C2–C3 are fulfilled if we set $v_j = v_j(\alpha) := \max(1, \min(\alpha p_j, d_j))$. By appropriately choosing α, we can also meet the utilization constraint C1: Consider the function $\xi(\alpha) := m - \sum_{j \in J} v_j(\alpha)$ which expresses the unused resources for a particular value of α. Function ξ is a continuous, monotonically decreasing, and piece-wise linear function (see Fig. 1). Moreover, $\xi(0) = m - n \geq 0$ and $\xi(\max_{j \in J} d_j/p_j) = m - \sum_{j \in J} d_j < 0$. Hence $\xi(\alpha) = 0$ has a solution α_0 which represents the desired choice of α that exploits all resources, i.e., it also fulfills constraint C1. Once α_0 is found, we need to round each $v_j(\alpha)$ to an integer. Due to C1 and C3, we propose to round down all volumes and then increment the volume of the $k := m - \sum_j \lfloor v_j(\alpha_0) \rfloor$ jobs of highest priority. More precisely, we

identify $J' := \{j \in J : v_j(\alpha) < d_j\}$, sort J' by job priority in descending order and select the first k of them.

We now outline a fully distributed algorithm which finds α_0 in logarithmic span. We exploit that the gradient of ξ changes at at most $2n$ values of α, namely when $\alpha p_j = 1$ or $\alpha p_j = d_j$ for some $j \in J$. Since we have $m \geq n$ PEs available, we can try these $\mathcal{O}(n)$ values of $\xi(\alpha)$ in parallel. We then find the two points with smallest positive value and largest negative value using a parallel reduction operation. Lastly, we interpolate ξ between these two points to find α_0.

The parallel evaluation of ξ is still nontrivial since a naive implementation would incur quadratic work – $\mathcal{O}(n)$ for each value of α. We now explain how to accelerate the evaluation of ξ. For this, we rewrite ξ as

$$\begin{aligned}\xi(\alpha) &= m - |\{j : 1/p_j > \alpha\}| - \sum_{j\,:\,d_j/p_j<\alpha} d_j - \alpha\left(\sum\nolimits_j p_j - \sum_{j\,:\,1/p_j>\alpha} p_j - \sum_{j\,:\,d_j/p_j<\alpha} p_j\right)\\ &\equiv m - A(\alpha) \qquad\qquad - \ B(\alpha) \quad - \alpha\left(\sum\nolimits_j p_j - \ C(\alpha) \ - \ D(\alpha) \ \right)\end{aligned}$$

Let us explain the terms one by one: m is part of the definition of ξ; $A(\alpha)$ (which can be viewed as $\sum_{j\,:\,1/p_j>\alpha} 1$) subtracts the resources of jobs whose v_j is raised to 1; $B(\alpha)$ subtracts the resources of jobs whose v_j is capped at their demand d_j; $\alpha \sum_j p_j$ subtracts the resources of *all* jobs. The terms $\alpha \cdot C(\alpha)$ and $\alpha \cdot D(\alpha)$ cancel the summands in $\alpha \sum_j p_j$ that have been subtracted wrongly because they are already covered by $A(\alpha)$ and $B(\alpha)$.

This new representation can be computed efficiently. $\sum_j p_j$ is independent of α and can be computed using a parallel reduction operation in logarithmic time. All other terms can be computed using prefix sums: We construct a key-value pair (α, v_α) for each α where the gradient of ξ changes, sort all pairs by α, and chose vectors v_α in such a way that an element-wise prefix sum over all v_α yields the desired terms $A(\alpha)$, $B(\alpha)$, $C(\alpha)$, and $D(\alpha)$ at each α. We achieve this with two key–value pairs for each job j, $\underline{t} = (1/p_j, (1, 0, p_j, 0))$ and $\overline{t} = (d_j/p_j, (0, d_j, 0, p_j))$. $\underline{t}$ represents the smallest value $\alpha = 1/p_j$ where v_j no longer needs to be raised to 1 and $\overline{t}$ represents the largest value $\alpha = d_j/p_j$ where v_j not yet needs to be cut to d_j. We sort these $2n$ tuples by their keys, accumulating key–value pairs $(\alpha, v_1), \ldots, (\alpha, v_k)$ with identical keys into a single key–value pair[1] $(\alpha, \sum_{i=1}^{k} v_i)$. We then calculate the prefix sum components σ_α:

$$\begin{aligned}\sigma_\alpha &:= \left(\sum_{j\,:\,1/p_j\leq\alpha} 1, \quad \sum_{j\,:\,d_j/p_j<\alpha} d_j, \quad \sum_{j\,:\,1/p_j\leq\alpha} p_j, \quad \sum_{j\,:\,d_j/p_j<\alpha} p_j \right)\\ &= \left(\ n - A(\alpha), \quad B(\alpha), \quad \sum\nolimits_j p_j - C(\alpha), \quad D(\alpha) \ \right)\end{aligned}$$

[1] This accumulation can be performed after sorting: A binary tree reduction reduces the minimum and maximum key of a subtree, their two accumulated values, and two intervals of PE ranks with these keys. A completed accumulation of several values is broadcast to the according interval of PEs in order to update the local elements.

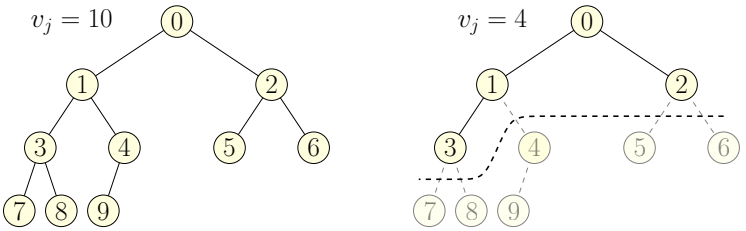


Fig. 2. Left: Job tree T_j has grown to ten workers $\{w_j^0, w_j^1, \ldots, w_j^9\}$ due to the volume $v_j = 10$ assigned to j. Right: Volume update $v_j = 4$ arrives. Consequently, all workers with index ≥ 4 are preempted and the corresponding PEs can adopt another job.

Note that our prefix sum is inclusive ("$\leq \alpha$") for the first and third element and exclusive ("$< \alpha$") for the second and fourth element. As such, at each evaluated α we can extract all four terms of interest from σ_α as depicted.

Computing prefix sums and sorting $\mathcal{O}(n)$ elements in parallel on $m \geq n$ PEs is possible in logarithmic time[2]. Selecting the k jobs to receive additional volume after rounding down all volumes can be reduced to sorting as well.

3.2 Assignment of Jobs to PEs

We now describe how the fair volumes computed as in the previous section translate to an actual assignment of jobs to PEs.

Basic Approach. We begin with our basic approach as introduced in [26].

For each job j, we address the k current workers in T_j as $w_j^0, w_j^1, \ldots, w_j^{k-1}$. These workers can be scattered throughout the system, i.e., their *job indices* $0, \ldots, k-1$ within T_j are not to be confused with their ranks. The k workers form a communication structure in the shape of a binary tree (Fig. 2). Worker w_j^0 is the root of this tree and represents j for the calculation of its volume (Section 3.1). Workers w_j^{2i+1} and w_j^{2i+2} are the left and right children of w_j^i. Jobs are made malleable by letting T_j grow and shrink dynamically. Specifically, we enforce that T_j consists of exactly $k = v_j$ workers. If v_j is updated, all workers w_j^i for which $i \geq v_j$ are suspended and the corresponding PEs turn idle. Likewise, workers without a left (right) child for which $2i+1 < v_j$ ($2i+2 < v_j$) attempt to find a child worker w_j^{2i+1} (w_j^{2i+2}). New workers are found via *request messages*: A request message $r = (j, i, x)$ holds index i of the requested worker w_j^i as well as rank x of the requesting worker. If a new job is introduced at some PE, then this PE emits a request for the root node w_j^0 of T_j. All requests for w_j^i, $i > 0$ are emitted by the designated parent node $w_j^{\lfloor (i-1)/2 \rfloor}$ of the desired worker.

[2] Asymptotically optimal sorting on communication networks [1] is of mostly theoretical value due to the large constant values involved. However there are quite practical algorithms when $n \in \mathcal{O}(\sqrt{m})$ or when spending $\mathcal{O}(\log^2 n)$ time is acceptable [4].

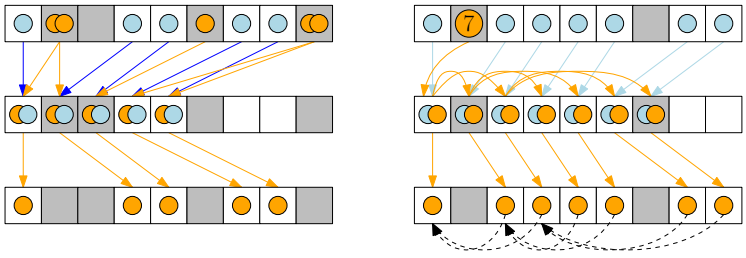


Fig. 3. Illustration for matching requests and idle PEs. White (gray) squares represent idle (busy) PEs, blue (orange) spheres represent idle tokens (requests). Left: A prefix sum (not depicted) numbers all requests and idle tokens, and each request (token) of index i is sent to rank i. Each PE with a matching pair sends the request to the idle PE. Right: A job j grows by multiple layers of T_j. Requests are sent along a tree structure and child-parent relationships of T_j are encoded into the distributed requests.

In [26], we proposed that each request performs a random walk through a regular graph of all PEs and is resolved as soon as it hits an idle PE. While this strategy resolves most requests quickly, some requests can require a large number of hops. If we assume a fully connected graph of PEs and a small share ϵ of workers is idle, then each hop of a request corresponds to a Bernoulli process with success probability ϵ, and a request takes an expected $1/\epsilon$ hops until an idle PE is hit. Consequently, to improve worst-case latencies, a small ratio of workers should be kept idle [26]. By contrast, our following algorithm with logarithmic span does not depend on suboptimal utilization.

Matching Requests and Idle PEs. In a first phase, our improved algorithm (see Fig. 3) computes two prefix sums with one collective operation: the number of requests q_i being emitted by PEs of rank $< i$, and the number o_i of idle PEs of rank $< i$. We also compute the total sums, q_m and o_m, and communicate them to all PEs. The q_i and o_i provide an implicit global numbering of all requests and all idle PEs. In a second phase, the i-th request and the i-th token are both sent to rank i. In the third and final phase, each PE which received both a request and an idle token sends the request to the idle PE referenced by the token.

If the request for a worker w_j^i is only emitted by its designated parent $w_j^{\lfloor (i-1)/2 \rfloor}$, then our algorithm so far may need to be repeated $\mathcal{O}(\log m)$ times: Repetition l activates a worker which then emits requests for repetition $l+1$. Instead, we can let a worker emit requests not only for its direct children, but for all *transitive* children it deserves. Each worker w_j^i can compute the number k of desired transitive children from v_j and i. The worker then contributes k to q_i. In the second phase, the k requests can be distributed communication-efficiently to a range of ranks $[x, \ldots, x+k-1]$: w_j^i sends requests for workers w_j^{2i+1} and w_j^{2i+2} to ranks x and $x+1$, which send requests for corresponding child workers to ranks $x+2$ through $x+5$, and so on, until worker index $v_j - 1$ is reached. To enable this distribution, we append to each request the values x, v_j, and the

rank of the PE where the respective parent worker will be initialized. As such, each child knows its parent within T_j (Fig. 3) for job-internal communication.

We now outline how our algorithm can be executed in a fully asynchronous manner. We compute the prefix sums within an In-Order binary tree of PEs [22, Chapter 13.3], that is, all children in the left subtree of rank i have a rank $< i$ and all children in the right subtree have a rank $> i$. This prefix sum computation can be made sparse and asynchronous: Only non-zero contributions to a prefix sum are sent upwards explicitly, and there is a minimum delay in between sending contributions to a parent. Furthermore, we extend our prefix sums to also include *inclusive* prefix sums q'_i, o'_i which denote the number of requests (tokens) at PEs of rank $\leq i$. As such, every PE can see from the difference $q'_i - q_i$ ($o'_i - o_i$) how many of its local requests (tokens) took part in the prefix sum. Last but not least, the number of tokens and the number of requests may not always match – a PE which receives either a request or an idle token (but not both) knows of this imbalance due to the total sums q_m, o_m. The unmatched message is sent to its origin and can re-participate in the next iteration.

Our matching algorithm has $\mathcal{O}(\log m)$ span and takes $\mathcal{O}(m)$ local work. The maximum local work of any given PE is in $\mathcal{O}(\log m)$ (to compute the above k), which is amortized by other PEs because at most m requests are emitted.

3.3 Reuse of Suspended Workers

Each PE remembers up to C most recently used workers (for a small constant C) and deletes older workers. Therefore, if a worker w_j^i is suspended, it may be resumed at a later time. Our algorithms so far may create a different worker on a different PE each time T_j shrinks and then re-grows. We now outline how we can increase the reuse of suspended workers.

In our previous approach [26], each worker remembers a limited number of ranks of its past (direct) children. A worker which desires a child queries them for reactivation one after the other until success or until all past children have been queried unsuccessfully, at which point a normal job request is emitted.

We make two improvements to this strategy. First, we remember past workers in a distributed fashion. More precisely, whenever a worker joins or leaves T_j, we distribute information along T_j to maintain the following invariant: Each current leaf w_j^i in T_j remembers the past workers which were located in a subtree below w_j^i. As such, past workers can be remembered and reused even if T_j shrinks by multiple layers and re-grows differently. Secondly, we adjust our scheduling to prioritize the reuse of existing workers over the initialization of new workers. In our implementation, each idle PE can infer from its local volume calculation (Section 4.1) which of its local suspended workers w_j^i are eligible for reuse, i.e., $v_j > i$ in the current volume assignment. If a PE has such a worker w_j^i, the PE will reject any job requests until it received a message regarding w_j^i. This message can either be a query which causes w_j^i to be resumed, or a notification that w_j^i has been initialized at a different PE instead. On the opposite side, a worker which desires a child begins to query past children according to a "most recently

used" strategy. If a query succeeds, all remaining past children are notified that they are unneeded. If all queries failed, a job request is emitted.

4 The Mallob System

In the following, we outline the design and implementation of our plattform named Mallob, short for **Mal**leable **Lo**ad **B**alancer. Mallob is a C++ application to be compiled with an implementation of the Message Passing Interface (MPI) [12]. Each PE of Mallob can be configured to accept jobs and return responses over the local file system, a Unix IPC socket, or an API within Mallob. The application-specific worker running on each PE is defined via an interface with a small set of methods. These methods define the worker's behavior if it is started, suspended, resumed, or terminated, and allow it to send and receive application-specific messages at will. Note that we outlined some of Mallob's earlier features in previous work [26] with a focus on our malleable SAT engine.

4.1 Implementation of Algorithms

Our system features practical and simplified implementations solving the volume assignment problem and the request matching problem. We now explain how and why these implementations differ from the algorithms provided in Section 3.

Volume Assignment. Our implementation computes job volumes similar to the algorithm outlined in Section 3.1. However, each PE computes the desired change of root α_0 of ξ locally. All events in the system (job arrivals, departures, and changes in demands) are aggregated and broadcast periodically such that each PE can maintain a local image of all active jobs' demands and priorities [26]. The local search for α_0 is then done via bisection over the domain of ξ. This approach requires more local work than our fully distributed algorithm and features a broadcast of worst-case message length $\mathcal{O}(n)$. However, it only requires a single all-reduction. At the scale we aimed for ($n < 10^3$ and $m < 10^4$), we expect in practice that our simplified approach performs better than our asymptotically superior algorithm which features several stages of collective operations. When targeting much larger configurations in the future, it may be beneficial to implement and employ our fully distributed algorithm instead.

Request Matching. We did not yet implement asynchronous prefix sums as described in Section 3.2. Instead, we route requests directly along a communication tree R of PEs. Each PE keeps track of the *idle count*, i.e., the number of idle PEs, in each of its subtrees in R. This count is updated transitively whenever the idle status of a child changes. Emitted requests are routed upwards through R until hitting an idle PE or until a hit PE has a subtree with a non-zero idle count, at which point the request is routed down towards the idle PE. If a large number of requests (close to n) are emitted, the traffic along the root of R may constitute

a bottleneck. However, we found that individual volume updates in the system typically result in a much smaller number of requests, hence we did not observe such a bottleneck in practice. We intend to include our bottleneck-free algorithm (Section 3.2) in a future version of our system.

4.2 Engineering

For good practical performance of our system, careful engineering was necessary. For instance, our system exclusively features asynchronous communication, i.e., a PE will never block for an I/O event when sending or receiving messages. As a result, our protocols are designed without explicit synchronization (barriers or similar). We only let the main thread of a PE issue MPI calls, which is the most widely supported mode of operation for multithreaded MPI programs.

As we aim for scheduling latencies in the range of milliseconds, each PE must frequently check its message queue and react to messages. For instance, if the main thread of a PE allocates space for a job description, this can cause a prohibitively long period where no messages are processed. For this reason, we use a separate thread pool for tasks which involve a risk of taking a long time. Furthermore, we use the scalable memory allocator `jemalloc` [9] instead of the default `malloc` implementation, and we split large messages into batches of smaller messages, e.g., when transferring large job descriptions to new workers.

5 Evaluation

We now present our experimental evaluation. All experiments have been conducted on the supercomputer SuperMUC-ng. If not specified otherwise, we used 128 compute nodes, each with an Intel Skylake Xeon Platinum 8174 processor clocked at 2.7 GHz with 48 physical cores (96 hardware threads) and 96 GB of main memory. SuperMUC-ng is running Linux (SLES) with kernel version 4.12 at the time of running our experiments. We compiled Mallob with GCC 9 and with Intel MPI 2019. We launch twelve PEs per machine, assign eight hardware threads to each PE, and let a worker on a PE use four parallel worker threads. Our system can use the four remaining hardware threads on each PE in order to keep disturbance of the actual computation to a minimum. Our software and experimental data is available at https://github.com/domschrei/mallob.

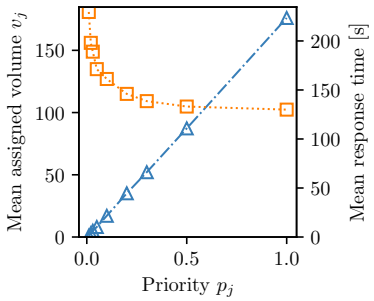
5.1 Uniform Jobs

In a first set of experiments, we analyze the base performance of our system by introducing a stream of jobs in such a way that exactly n_{par} jobs are in the system at any time. We limit each job j to a CPU time budget B inversely proportional to n_{par}. Each job corresponds to a difficult SAT formula which cannot be solved within the given budget. As such, we emulate jobs of fixed size.

We chose m and the values of n_{par} in such a way that $m/n_{\text{par}} \in \mathbb{N}$ for all runs. We compare our runs against a hypothetical rigid scheduler which functions as

n_{par}	θ	θ_{opt}	$\frac{\theta}{\theta_{opt}}$	η	u
3	0.159	0.16	0.991	0.990	0.981
6	0.318	0.32	0.994	0.990	0.983
12	0.636	0.64	0.993	0.991	0.984
24	1.271	1.28	0.993	0.992	0.985
48	2.543	2.56	0.993	0.993	0.985
96	5.071	5.12	0.990	0.993	0.986
192	10.141	10.24	0.990	0.995	0.985
384	20.114	20.48	0.982	0.995	0.983
768	39.972	40.96	0.976	0.992	0.980

Table 1. Scheduling statistics for uniform jobs on 1536 PEs (6144 cores) compared to a hypothetical optimal rigid scheduler. From left to right: Max. number n_{par} of parallel jobs; max. measured throughput θ, optimal throughput θ_{opt} (in jobs per second), throughput efficiency θ/θ_{opt}; work efficiency η; mean measured CPU utilization u of worker threads.



p_j	$\tilde{v}_j$	RT [s]
0.01	1.0	229.6
0.02	3.0	198.0
0.03	5.0	189.1
0.05	8.1	171.3
0.10	17.1	161.2
0.20	35.2	146.0
0.30	52.4	138.6
0.50	87.5	133.1
1.00	176.6	130.0

Fig. 4. Impact of job priority on mean assigned volume (left axis, blue triangles) and response time (right axis, orange squares). The table shows the used priorities p_j with the corresponding mean assigned volume $\tilde{v}_j$ and mean response times in seconds.

follows: Exactly m/n_{par} PEs are allotted for each job, starting with the first n_{par} jobs at $t = 0$. At periodic points in time, all jobs finish and each set of PEs instantly receives the next job. This leads to perfect utilization and maximizes throughput. We neglect any kind of overhead for this scheduler.

For a modest number of parallel jobs n_{par} in the system ($n_{\text{par}} \leq 192$), our scheduler reaches 99% of the optimal rigid scheduler's throughput (Table 1). This efficiency decreases to 97.6% for the largest n_{par} where $v_j = 2$ for each job. As the CPU time of each job is calculated in terms of its assigned volume and as the allocation of workers takes some time, each job uses slightly less CPU time than advertised: Dividing the time for which each job's workers have been active by its advertised CPU time, we obtained a work efficiency of $\eta \geq 99\%$. Lastly, we measured the CPU utilization of all worker threads as reported by the operating system, which averages at 98% or more. In terms of overall work efficiency $\eta \times u$, we observed an optimum of 98% at $n_{\text{par}} = 192$, a point where neither n_{par} nor the size of individual job trees is close to m.

5.2 Impact of Priorities

In the following we evaluate the impact of job priorities. We use 32 nodes (1536 cores, 384 PEs) and introduce nine streams of jobs, each stream with a different

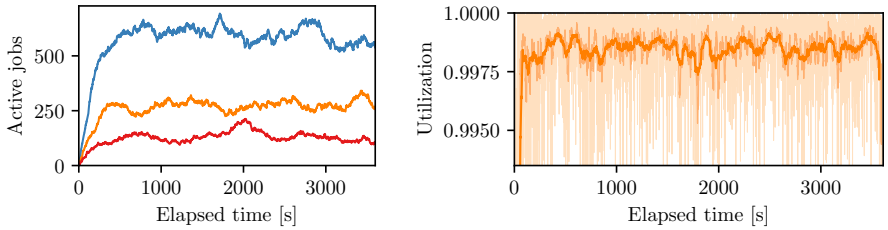


Fig. 5. Left: Number of active jobs for interarrival times $1/\lambda$ of 2.5 s (top), 5 s (middle), and 10 s (bottom). Right: System utilization (i.e., ratio of busy PEs) for $1/\lambda = 5$ s at a sliding average of window size 1 s, 15 s, and 60 s respectively.

job priority $p \in [0.01, 1]$ and with a wallclock limit of 300 s per job. As such, the system processes nine jobs with nine different priorities at a time. Each stream is a permutation of 80 diverse SAT instances [26].

As expected, we observed a proportional relationship between priority and assigned volume, with small variations due to rounding (Fig. 4). By contrast, response times appear to decrease exponentially towards a certain lower bound, which is in line with the NP-hardness of SAT and the diminishing returns of parallel SAT solving [26]. The modest margin by which response times decrease is due to the difficulty of the chosen SAT benchmarks, many of which cannot be solved within the imposed time limit at either scale.

5.3 Realistic Job Arrivals

In the next set of experiments, we analyze the properties of our system in a more realistic scenario. Four PEs introduce batches of jobs with poisson-distributed arrivals (inter-arrival time of $1/\lambda \in \{2.5\,\text{s}, 5\,s, 10\,\text{s}\}$) and between one and eight jobs per batch. As such, we simulate users which arrive at independent times and submit a number of jobs at once. We also sample a priority $p_j \in [0.01, 1]$, a maximum demand $d_j \in 1, \ldots, 1536$, and a wallclock limit $b_j \in [1, 600]$ s for each job. We ran this experiment with our current request matching (Section 4.1) and with each request message performing up to h random hops (as in [26]) until our request matching is employed, for varying values of h. In addition, we ran the experiment with three different suspended worker reuse strategies: No deliberate reuse at all, the basic approach from [26], and our approach.

Fig. 5 (left) shows the number of active jobs in the system over time for our default configuration (our reuse strategy and immediate matching of requests). For all tested interarrival times, considerable changes in the system load can be observed during a job's average life time which justify the employment of a malleable scheduling strategy. Fig. 5 (right) illustrates for $1/\lambda = 5\,s$ that system utilization is at around 99.8% on average and almost always above 99.5%. We also measured the ratio of time for which each PE has been idle: The median PE was busy 99.08% of all time for the least frequent job arrivals ($1/\lambda = 10$ s),

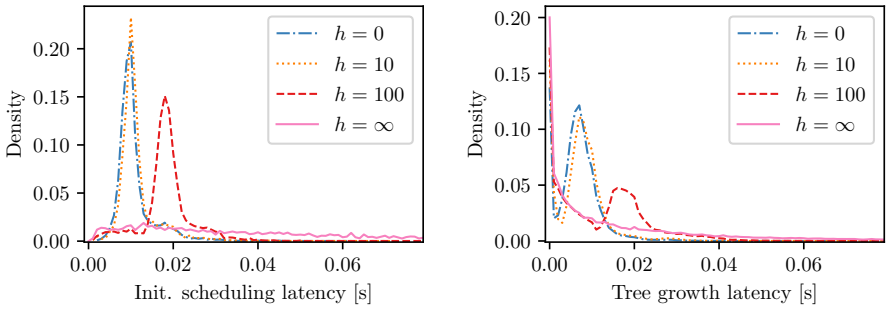


Fig. 6. Distribution over measured latency for the initial scheduling of a job (left) and the expansion of a job tree by another worker (right), for inter arrival rate $1/\lambda = 5\,s$, for a varying number h of random hops until a request message is routed along R.

99.77% for $1/\lambda = 5\,\text{s}$, and 99.85% for $1/\lambda = 2.5\,\text{s}$. Also note that $\sum_j d_j < m$ for the first seconds of each run, hence not all PEs can be employed immediately.

In the following, we focus on the experiment with $1/\lambda = 5\,\text{s}$. The latency of our volume calculation, i.e., the latency until a PE received an updated volume for an updated job, reached a median of 1 ms and a maximum of 34 ms for our default configuration. For the scheduling of an arriving job, Fig. 6 (left) shows that the lowest latencies were achieved by our request matching ($h = 0$). For increasing values of h, the variance of latencies increases and high latencies ($\geq 50\,\text{ms}$) become more and more likely. Note that jobs normally enter a fully utilized system, and have $d_j = 1$. Therefore, the triggered balancing calculation may render only a single PE idle, which heavily disfavors performing a random walk. Fig. 6 (right) indicates that random walks sometimes result in very low latencies for expanding a job tree by another layer, but still result in a shift towards high latencies ($> 10\,\text{ms}$) for a considerable share of requests.

To compare suspended worker reuse strategies, we divided the number of created workers for a job j by its maximum assigned volume v_j. This Context Creation Ratio (CCR) is ideally 1 and becomes larger the more often a worker is suspended and then re-created at a different PE. We computed the CCR for each job and in total: As Tab. 2 shows, our approach reduces a CCR of 2.16 down to 1.83 (-15.3%). Context switches (i.e., how many times a PE changed its affiliation) and average response times are improved marginally compared to the naive approach. Last but not least, we counted on how many distinct PEs each w_j^i has been created: Our strategy initializes 86% of all workers only once, and 94% of workers have been created at most five times. We conclude that most jobs only feature a small number of workers, at the bottom of their job tree, which are rescheduled frequently.

	CCR			CS				Pr [WC ≤ ·]				
	med.	max.	total	med.	mean	Pr.	RT	1	2	5	10	25
None	1.43	33.5	2.16	136	138.2	5920	153.48	0.78	0.87	0.93	0.96	0.993
Basic	1.40	31.5	2.08	134	135.3	5917	153.99	0.78	0.87	0.94	0.97	0.993
Ours	1.25	24.5	1.83	130	131.8	5931	152.54	0.86	0.90	0.94	0.97	0.997

Table 2. Comparison of worker reuse strategies in terms of context creation ratio (CCR, per job – median, maximum – and in total), context switches (CS, median per PE and mean), the number of processed jobs within 1 h (Pr.), their mean response time (RT), and the fraction of workers created on at most $\{1, 2, 5, 10, 25\}$ distinct PEs.

6 Conclusion

We have presented a decentralized and highly scalable approach to online job scheduling of malleable NP-hard jobs with unknown processing times. We split our problem into two subproblems, namely the computation of fair job volumes and the assignment of jobs to PEs, and proposed scalable distributed algorithms with good asymptotic guarantees for both of them. We presented a practical implementation and experimentally showed that it schedules incoming jobs within a few dozens of milliseconds, distributes resources proportional to each job's priority, and leads to near-optimal utilization of resources.

For future work, we intend to add engines for applications beyond SAT into our system. Furthermore, we want to generalize our approach to heterogeneous computing environments and add fault tolerance to our distributed algorithms.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500). The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de). The authors wish to thank Tim Niklas Uhl as well as the anonymous reviewers for their helpful feedback.

References

1. Ajtai, M., Komlós, J., Szemerédi, E.: Sorting in $\log n$ parallel steps. Combinatorica **3**(1), 1–19 (1983)
2. Alquraan, A., Takruri, H., Alfatafta, M., Al-Kiswany, S.: An analysis of Network-Partitioning failures in cloud systems. In: Symposium on Operating Systems Design and Implementation. pp. 51–68 (2018)
3. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI. pp. 399–404 (2009)

4. Axtmann, M., Sanders, P.: Robust massively parallel sorting. In: ALENEX. pp. 83–97 (2017)
5. Blazewicz, J., Kovalyov, M.Y., Machowiak, M., Trystram, D., Weglarz, J.: Preemptable malleable task scheduling problem. IEEE Transactions on Computers **55**(4), 486–490 (2006)
6. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: Symposium on Operating Systems Design and Implementation. pp. 173–186 (1999)
7. Cook, S.A.: The complexity of theorem-proving procedures. In: ACM symposium on Theory of computing. pp. 151–158 (1971)
8. Desell, T., El Maghraoui, K., Varela, C.A.: Malleable applications for scalable high performance computing. Cluster Computing **10**(3), 323–337 (2007)
9. Evans, J.: Scalable memory allocation using jemalloc. Notes Facebook Eng (2011)
10. Feitelson, D.G.: Job scheduling in multiprogrammed parallel systems (1997)
11. Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M.: SAT competition 2020. Artificial Intelligence **301**, 103572 (2021)
12. Gropp, W., Gropp, W.D., Lusk, E., Skjellum, A., Lusk, E.: Using MPI: portable parallel programming with the message-passing interface, vol. 1. MIT press (1999)
13. Gupta, A., Acun, B., Sarood, O., Kalé, L.V.: Towards realizing the potential of malleable jobs. In: HiPC. pp. 1–10. IEEE (2014)
14. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation **6**(4), 245–262 (2010)
15. Heisinger, M., Fleury, M., Biere, A.: Distributed cube and conquer with paracooba. In: SAT. pp. 114–122. Springer (2020)
16. Huang, C., Lawlor, O., Kale, L.V.: Adaptive MPI. In: Int. workshop on languages and compilers for parallel computing. pp. 306–322. Springer (2003)
17. Hungershofer, J.: On the combined scheduling of malleable and rigid jobs. In: Symposium on Computer Architecture and HPC. pp. 206–213. IEEE (2004)
18. Kleine Büning, M., Balyo, T., Sinz, C.: Using DimSpec for bounded and unbounded software model checking. In: ICFEM. pp. 19–35. Springer (2019)
19. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of satisfiability, pp. 131–153. ios Press (2009)
20. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. Journal of Automated Reasoning **24**(1), 165–203 (2000)
21. Ozdemir, A., Wu, H., Barrett, C.: SAT solving in the serverless cloud. In: FMCAD. pp. 241–245. IEEE (2021)
22. Sanders, P., Mehlhorn, K., Dietzfelbinger, M., Dementiev, R.: Sequential and Parallel Algorithms and Data Structures. Springer (2019)
23. Sanders, P., Speck, J.: Efficient parallel scheduling of malleable tasks. In: IPDPS. pp. 1156–1166. IEEE (2011)
24. Sanders, P., Speck, J.: Energy efficient frequency scaling and scheduling for malleable tasks. In: Euro-Par. pp. 167–178. Springer (2012)
25. Schreiber, D.: Lilotane: A lifted SAT-based approach to hierarchical planning. JAIR **70**, 1117–1181 (2021)
26. Schreiber, D., Sanders, P.: Scalable SAT solving in the cloud. In: SAT. pp. 518–534. Springer (2021)
27. Sonmez, O., Mohamed, H., Lammers, W., et al.: Scheduling malleable applications in multicluster systems. In: IEEE Cluster. pp. 372–381. IEEE (2007)