International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 11(2019) pp. 1-28
https://doi.org/10.70917/ijcisim-2026-5991
Article

Certified lower and upper bounds on the pooled core capacity of an industrial compute farm under bounded job waiting times

Luis Eduardo Muñoz Guerrero*

Facultad de Ingenierías, Universidad Tecnológica de Pereira, Risaralda, Colombia
* Corresponding author e-mail: lemunozg@utp.edu.co ORCID: 0000-0002-9414-6187.

Abstract: How many cores does a batch compute farm need if every job must start within a bound of its submission, and how much do waiting and job suspension save? We answer for a mod-el that pools the cores of each pool and knows the jobs in advance, on one month of ac-counting logs of four pools of Intel's NetBatch farm, with 48,821,850 records. The non-preemptive problem is strongly NP-hard, so we bracket the optimum between energetic lower bounds, computed per interval length in time linear in the number of jobs and the horizon, and verified greedy list schedules found by binary search. In a pre-specified study, with three pools sealed until the plan was frozen, a bound of 1 h would lower the capacity needed without waiting by between 22.1% and 25.3% in pool A and between 17.9% and 21.8% in pool B, but only between 4.0% and 4.4% in pool D, so the hypothesis that it saves less than 10% was not supported; the second half of the month alone would have supported it in pool B. Suspension would save at most 0.4% more in pool D; in pools A and B the bounds could not decide whether it saves less than 2%, and by design they could never show that it saves more. On small instances solved exactly, the lower bound with the grid of the pools equalled the optimum in 27 of 31 cases, and the list schedule exceeded it by up to 28.9%. The frontiers price a waiting bound in pooled cores, a lower bound on what real machines would need, and the price differs strongly between pools and over time.
Keywords: Capacity planning, machine minimization, NP-hardness, energetic reasoning, list scheduling, preemptive scheduling, batch computing, workload traces

1. Introduction

Intel runs its chip-simulation workloads on a compute farm of tens of thousands of servers in several data centres. Its NetBatch system groups these servers into physical pools, each managed by a pool manager that matches queued jobs to free machines under fair-share rules [1]; a pool holds up to thousands of servers according to a study of these managers [1], and typically tens of thousands of nodes according to the archive that distributes its logs, which has not disclosed the exact configuration [2]. How many cores a pool needs is a capacity planning question with a direct cost: every core that a pool could do without is hardware, energy and floor space that the organization buys and maintains for nothing.

The answer depends on how long the users are willing to let their jobs wait. If no job may wait at all, a pool needs as many cores as the largest number of cores that jobs request at the same instant. If jobs may wait, the scheduler can shift work out of the peaks, and fewer cores suffice. A second lever is preemption: a system that can suspend a running job and resume it later can serve urgent work without holding cores in reserve for it, and NetBatch does suspend lower-priority jobs when higher-priority ones arrive [3]. Both levers trade waiting time for capacity, but how much capacity they actually buy on a production workload is an empirical question.

The usual tools answer a different question. Queueing models size a service system for a stationary arrival process, as the square-root safety staffing rule does for large call centres [4]. Dynamic right-sizing turns servers off

during periods of low load to save energy, trading operating cost against the cost of switching [5, 6]. Cell compaction measures how many machines a cluster scheduler needs for a workload by packing a snapshot of it into ever smaller cells, but it packs the workload from a point in time rather than replaying a long trace [7]. And a simulation study of the NetBatch pool managers scaled the offered load instead of the capacity, because the original load was too low to build a meaningful queue on the machine configuration available to its authors [1].

This paper asks the capacity question directly, on the month-long accounting logs of four NetBatch pools: what is the smallest number of cores with which every job of the month could have started within a waiting-time bound W of its submission, with and without job suspension, for a clairvoyant scheduler that knows the jobs in advance and pools the cores of all the machines? The answer is a yardstick, not a scheduler: on the same pooled cores, no online policy can meet the bound with fewer cores than the clairvoyant optimum of its kind, with or without suspension, so the optimum bounds what any investment in scheduling could save on them. Because the model leaves out machines, memory and priorities, its capacities are lower bounds on what the real pools would need for the same jobs, not a plan for them.

The difficulty is computational. Without suspension the question is a machine minimization problem with time windows, which is strongly NP-hard even when all jobs are released together. With suspension it is solved by network flows when every job uses one core, but the jobs of these pools use several cores at once, and even for one core per job the flow network grows with the number of jobs times the number of time intervals in their windows, beyond reach for a month with millions of jobs per pool. Hardness in the worst case does not make every instance hard, but the integer program of our exact baseline already exceeds its size limit at 12,800 jobs with a waiting bound of 60 min. We therefore bracket the answer instead of computing it exactly: a lower bound that holds for every schedule, and a verified schedule whose capacity exceeds it by a reported gap; when the two meet, the optimum is certified.

The contributions are the following.

First, we formulate the capacity question on a job trace as bounded-wait capacity minimization, the job-level counterpart of the trace-based required capacity of resource pools, and place it among known problems: without preemption it is strongly NP-hard even when all jobs are released together, by a short reduction from bin packing, and with preemption and unit demands it is solved by a known flow network. Second, we bracket the optimum of a month of a production pool with known techniques that scale to millions of jobs: energetic lower bounds evaluated for a fixed grid of interval lengths at every interval start, and greedy list schedules in submission order, found by binary search on the capacity and checked by an independent verifier. Third, we apply the procedure in a study of four NetBatch pools whose analysis plan was frozen before three of them were opened, the fourth being used for development, and we repeat the analysis on a second public log.

Three findings stand out. First, in the model, the saving that a waiting bound buys depends on the pool and on the period: in two of the three pools held out for confirmation, a bound of 1 h would save at least 17.9% of the cores needed without waiting, against at most 4.4% in the third, so the pre-specified hypothesis that such a bound saves less than 10% is not supported, although the second half of the month alone would support it in one of the two pools. Second, the value of suspension could be bounded below 2% only in the pool where waiting saves little; in the other two the bounds leave it undecided, and they could never have shown a large value, because they include no schedule with suspension. Third, on the small instances solved exactly, the lower bound with the set of interval lengths used for the pools equals the optimum in 27 of the 31 cases, which places the remaining gap, at least on those instances, in the list schedules. The rest of this section reviews related work.

Sizing computing systems. Queueing theory sizes a system under stationary, independent arrivals; for large call centres modelled as M/M/N queues, an asymptotic analysis gives the least number of agents that meets a constraint on waiting and revisits the square-root safety staffing rule [4]. Dynamic right-sizing turns servers off during periods of low load; its offline optimum has a simple structure in reverse time [5], and when the number of active servers must be an integer the offline problem is still solvable in polynomial time [6]. A greedy scheduler that fills the most energy-efficient servers first places tasks on a minimum number of servers while keeping the response time of the data centre

within a maximum, and its stability is proved for independent exponential arrivals [8]. Queueing models bound waiting in distribution under stochastic assumptions, and right-sizing trades energy against switching; neither says how many cores a given trace needs for a hard bound on the waiting time of every job. Trace-based capacity management comes closest: it defines the required capacity of a resource pool as the smallest capacity under which no unit of the measured demand is carried from one interval to the next for more than a given number of minutes [9]. That definition treats demand as divisible units per measurement interval; ours bounds the wait of every rigid job, with and without preemption. On production traces of a large enterprise cluster, a scheduler that enforces deadlines inferred from history lowered the footprint of the cluster by 14% to 28% [10], which measures what one scheduler achieves. Borg compares scheduling policies by the number of machines they need, through cell compaction of a production workload, and its authors note that they packed a workload from a point in time rather than replaying a long-term trace [7]. Our measure is also a number of cores, but it is computed over a month of replayed jobs, with a per-job waiting bound, and it is bracketed by proven bounds rather than estimated by a heuristic scheduler.

NetBatch. The NetBatch logs that we study were introduced in a simulation study of heuristics that match jobs to machines inside a physical pool; the study found that the original load was too low to build a meaningful queue on the machine configuration it used, and scaled the arrival times to raise the load [1]. An earlier analysis of one year of NetBatch traces from twenty pools found long completion times at peak hours, which it attributed to high-priority jobs suspending low-priority ones, even when the average utilisation was around 40%, and proposed rescheduling suspended jobs to other pools [3]. That study treats suspension as a source of waste; we ask how much capacity it saves.

Preemption in batch scheduling. Simulations that replay logs from supercomputer centres show that a tuneable strategy of selective suspension is effective in parallel job scheduling, and examine its impact on the worst-case response time [11]. Such studies fix the capacity and measure waiting; we fix the waiting bound and measure capacity.

Machine minimization. Finding the least number of identical machines on which jobs with release dates and deadlines can be scheduled is the machine minimization problem. Without preemption, randomised rounding of a linear relaxation was known to give an $O(\log n / \log \log n)$ approximation, and a constant factor is possible when the optimal number of machines is bounded by a constant [12]. With integer data and a slack of at most one between the release and the last allowed start, the minimum number of machines can be found efficiently, while slacks up to two already make the problem NP-complete; with equal release dates and deadlines it is bin packing [13]. Minimizing the peak demand of non-preemptible jobs with flexible start times, a problem of smart-grid energy scheduling, is NP-hard, and fixed-parameter and approximation algorithms are known for it [14]. The online preemptive version of machine minimization admits an $O(\log m)$-competitive algorithm [15], and an $O(\log \log m)$-competitive one [16]. When no waiting is allowed, the start time of every job is given, which is the setting of interval scheduling [17]. In project scheduling, minimizing the costs of the resources of a project subject to a time limit for its completion was studied early [18]; as the resource investment problem, in which the cost of each resource depends on the largest amount of it that the project requires, it is NP-hard and has lower bounds by Lagrangean relaxation and column generation that also yield feasible solutions [19]; our non-preemptive problem has the same objective for a single resource, with a time window for every job instead of precedence constraints and a common deadline.

Preemptive feasibility and lower bounds. Whether jobs that each use one machine at a time, with release times and deadlines, can be scheduled with preemption on identical or uniform machines is decided by a maximum flow computation in a suitable network [20, 21], or by a generalization of network flow techniques [22]. Preemptive relaxations solved by flows also give lower bounds for non-preemptive problems with release dates, and semi-preemptive variants strengthen them [23]. Energetic reasoning gives one of the necessary conditions for the existence of a feasible schedule of activities with release dates, deadlines and resource requirements on a resource of fixed capacity [24], and enhanced energetic tests substantially improve an exact branch-and-bound algorithm for identical parallel machines [25]. An exact check of energetic reasoning can be restricted to $O(n \log n)$ time intervals and runs

in $O(n \log^2 n)$ time [26]. Here energetic reasoning gives a lower bound over a whole production month rather than a test inside a search, evaluated on a fixed grid of interval lengths, which keeps the bound valid but may leave it weaker than the maximum over all intervals.

Workload logs. The Parallel Workloads Archive collects job-level accounting logs from production systems, and its curators document the data-quality problems that such logs contain, from missing and inconsistent records to configuration changes during the logging period [27].

Table 1. Notation: the symbols used in the paper, their meaning and their domain, with units.

Symbol	Meaning	Domain
J, n	jobs of a pool in the month (one per record of a run) and their number	set; $Z_{>0}$
r_j	release time: submission time of the iteration that the record describes	$Z_{\geq 0}$, s
p_j	processing time: finish time minus start time minus time suspended	$Z_{>0}$, s
c_j	demand: cores used at the same time while the job runs	$\frac{1}{2}Z_{>0}$, cores
W	waiting-time bound, the same for every job	$Z_{\geq 0}$, s
S_j	start time in a non-preemptive schedule, $r_j \leq S_j \leq r_j + W$	R, s
d_j	deadline in a preemptive schedule, $r_j + p_j + W$	$Z_{\geq 0}$, s
m	capacity of the pool in cores	$Z_{>0}$, cores
$m^*_{NP}(W)$, $m^*_P(W)$	least capacity without and with preemption	$Z_{>0}$, cores
$m^*(0)$	least capacity when no job may wait (Lemma 2)	$Z_{>0}$, cores
$S(W)$	share of $m^*(0)$ that the bound W saves	$[0,1)$
$V(W)$	share of $m^*_{NP}(W)$ that preemption saves	$[0,1)$
a, L	start and length of an interval $[a, a+L]$	Z; $Z_{>0}$, s
$l_j(a, L)$	overlap of the window $[r_j, d_j]$ of job j with the interval	$Z_{\geq 0}$, s
$F^P_j(a, L)$, $F^{NP}_j(a, L)$	forced work: core-seconds of job j that every preemptive or non-preemptive schedule runs in the interval	$R_{\geq 0}$, core-seconds
$G_L(a)$	total forced work in the interval, summed over the jobs	$R_{\geq 0}$, core-seconds
L_P, L_{NP}	energetic lower bounds (Theorem 3, Algorithm 1)	$Z_{>0}$, cores
U_P, U_{NP}	capacities of verified list schedules (Algorithm 2)	$Z_{>0}$, cores
V^-, γ, δ	upper end of $V(W)$; gap of the list; distance between the lower bounds	$R_{\geq 0}$
W_k	waiting bounds of the grid, increasing	$Z_{\geq 0}$, s
$\lambda(t)$	load without waiting: cores in use at t when every job starts at its release	$R_{\geq 0}$, cores

$w_j^{(m)}$, U_q	wait of job j with m cores; capacity for a share q of the jobs	$Z_{\geq 0}$, s; $Z_{>0}$, cores
L, T, K	interval lengths, horizon in seconds, number of distinct demands	$\subset Z_{>0}$, s; $Z_{>0}$, s; $Z_{>0}$

A replayed log is an open-loop model: it ignores the feedback from the system to its users, which workload models have to consider [28, 29].

Materials and Methods presents the problem, its complexity, the bounds and algorithms, the data and the pre-specified design; Results, Discussion and Conclusions follow.

2. Materials and Methods

Problem. A pool receives a set J of n jobs during a month. Job j is submitted at its release time r_j, runs for p_j seconds of processing and uses c_j cores at the same time while it runs. A pool with capacity m has m interchangeable cores; we treat the cores of a pool as a single cumulative resource of interchangeable cores and ignore the boundaries between machines and the memory of each machine, so the optimal capacities of this paper are lower bounds on what a pool of real machines would need for the same release and processing times. The waiting-time bound $W \geq 0$ is the same for all jobs. Table 1 lists the notation.

A non-preemptive schedule gives each job a start time S_j and runs it without interruption in $[S_j, S_j + p_j)$. It is feasible with capacity m if no instant uses more than m cores, Equation (1), and no job waits more than W, Equation (2):

$$\sum_{j \in J:\, S_j \leq t < S_j + p_j} c_j \leq m \qquad for\ all\ t \tag{1}$$

$$r_j \leq S_j \leq r_j + W \qquad for\ all\ j \in J \tag{2}$$

In a preemptive schedule, the processing of job j may be suspended and resumed, on any cores, as long as no instant uses more than m cores and the job receives p_j seconds of processing on c_j cores inside the window that ends at its deadline, given by Equation (3), so that its total wait, including the time it spends suspended, is at most W:

$$d_j = r_j + p_j + W \tag{3}$$

The quantities we study are the least capacities with which such schedules exist, Equation (4), and two ratios: the share of the capacity without waiting that a bound W saves, and the share of the non-preemptive capacity that suspension saves on top, Equation (5).

$$\begin{aligned} & m_{NP}^*(W) \\ & = min\{\, m :\ a\ feasible \quad non \\ & - preemptive\ schedule\ exists\,\}, m_P^*(W) \\ & = min\{\, m :\ a\ feasible \quad preemptive\ schedule\ exists\,\} \end{aligned} \tag{4}$$

$$S(W) = 1 - \frac{m_{NP}^*(W)}{m^*(0)}, V(W) = \frac{m_{NP}^*(W) - m_P^*(W)}{m_{NP}^*(W)} \tag{5}$$

Lemma 1. For every $W \geq 0$, $m_P^*(W) \leq m_{NP}^*(W)$, and both are non-increasing in W. A non-preemptive schedule is a preemptive one, and a schedule that meets a bound also meets every larger bound.

Lemma 2. With $W = 0$ every job runs exactly in $[r_j, r_j + p_j)$, preemption cannot help, and the least capacity is the largest number of cores in use at any instant, the load without waiting $\lambda(t)$, rounded up to an integer, Equation (6). It is computed in $O(n \log n)$ time by sorting the $2n$ start and end events and scanning them, with ends before starts at equal times.

$$\lambda(t) = \sum_{j:\, r_j \le t < r_j + p_j} c_j, \quad m^*(0) = \lceil \max_t \lambda(t) \rceil \tag{6}$$

Theorem 1. Deciding whether a non-preemptive schedule with capacity m exists is strongly NP-complete, even when $c_j = 1$ for every job, all jobs are released at the same time and W is the same for all.

Strong NP-hardness rules out even algorithms whose time is polynomial in the input length and the magnitude of its numbers, unless P = NP [30]. The proof reduces bin packing, which remains NP-hard even when all its numbers are bounded by a polynomial in the length of the input and is therefore strongly NP-hard [31], to the decision problem.

Given items $a_1, \ldots, a_k$ and B bins of size C, create one job of length a_i per item and B blocking jobs of length $C + 1 + max_i\, a_i$, all released at time zero, with $W = C$ and $m = B$. With unit demands, a schedule can be split among individual cores, as the Supplementary Material shows, and without loss of generality the jobs of each core run back to back from time zero, because starting a job earlier never lengthens a wait; the last job of a core then starts at the total length of the others, and that start must not exceed C. A blocking job is longer than C, so it can only be the last job of its core, and with B cores and B blocking jobs every core ends with one of them. The item jobs of each core therefore total at most C, which is a packing; conversely, any packing gives such a schedule. The transformation takes polynomial time, and the numbers it creates are at most twice the largest number of the bin-packing instance plus one, so strong NP-hardness carries over. The Supplementary Material gives the full proof and draws the construction (Figure S1). With a common release time and unit demands, a schedule with capacity m exists exactly when the jobs other than the m longest fit on m cores within W, because exchanging a job with a longer last job of another core never lengthens a wait; the theorem is therefore a short observation on a classical problem rather than a new hardness result.

Theorem 2. With unit demands, $m_P^*(W)$ can be computed in polynomial time: for a given m, the existence of a preemptive schedule with release times and deadlines is decided by a maximum flow computation [20, 21], and, because $m_P^*(W) \le n$, a binary search on m between one and n finds it with $O(log\, n)$ such computations. With demands of several cores, as in the pools, we claim no polynomial algorithm. Figure S2 draws the network on three jobs.

The two results set what can be computed. Unless P = NP, no algorithm computes the non-preemptive capacity of every instance in polynomial time, and exact methods such as the integer program of the baseline below are out of reach for a month of a production pool; even the preemptive capacity with unit demands requires a flow network whose size grows with the number of jobs times the number of time intervals in their windows, beyond reach for the millions of jobs of a pool. We therefore bracket both capacities between proven lower bounds and capacities of explicit schedules, which certify the optimum of an instance whenever they meet.

Lower bounds. Consider any interval $[a, a + L]$ of length L. In a preemptive schedule, job j is processed for p_j seconds inside its window $[r_j, d_j]$, whose length is $p_j + W$. Outside the interval the window offers at most $p_j + W - |[r_j, d_j] \cap [a, a + L]|$ seconds, so at least the amount of its core-seconds given by Equation (7) must be processed inside the interval. In a non-preemptive schedule the job occupies $[S_j, S_j + p_j)$ with $r_j \le S_j \le r_j + W$, and its overlap with the interval is smallest at one of the two extreme starts, which gives the larger forced work of Equation (8) (Figure 1(a)).

$$F_j^P(a,L) = c_j \max\big(0,\ l_j(a,L) - W\big), l_j(a,L) \\ = \big|[r_j, d_j] \cap [a, a+L]\big| \tag{7}$$

$$F_j^{NP}(a,L) = c_j \max(0,\ \min(L,\ p_j,\quad r_j + p_j \\ - a,\quad a + L - r_j - W)) \tag{8}$$

Theorem 3. For every set L of interval lengths, every feasible schedule of the corresponding kind satisfies Equation (9), with $F_j = F_j^P$ for preemptive and $F_j = F_j^{NP}$ for non-preemptive schedules. The interval holds at most mL core-seconds and must contain the forced work of every job; this is the energetic reasoning of cumulative scheduling [24], applied here to every interval start and to a set of lengths.

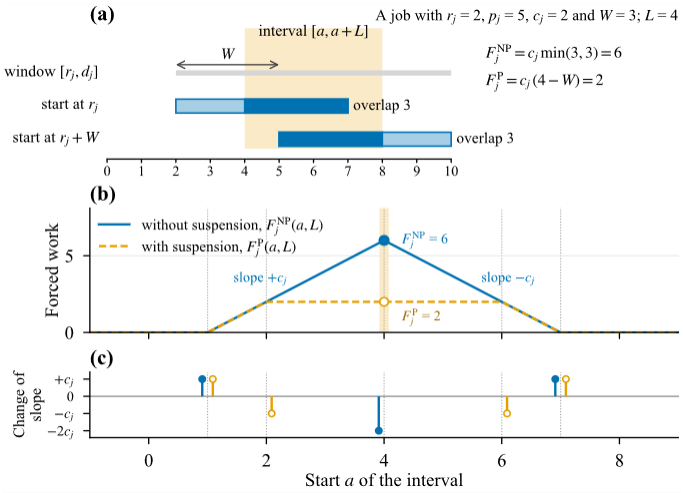


Figure 1. Forced work of one job in one interval, an illustration of Equation (7) and Equation (8). (a) The window $[r_j, d_j]$ of the job, with $d_j = r_j + p_j + W$, and its run when it starts at its release or at its last allowed start $r_j + W$; the darker part of each run lies inside the interval $[a, a+L]$ (shaded). Without suspension, every allowed run lies between these two, so the smaller of their overlaps with the interval, times c_j, is forced into it. With suspension, the job can process outside the interval everything that its window leaves outside, so the overlap of the window with the interval minus W, times c_j, is forced into it. (b) The forced work as a function of the start a of the interval, with L fixed, without suspension (solid) and with it (dashed): a trapezoid with slopes $\pm c_j$; the dots mark the interval of (a). (c) The four changes of slope of each trapezoid, filled without suspension and open with it (side by side where both change), which Algorithm 1 adds to a vector; two prefix sums of the vector give (b).

$$m \geq \lceil \max_{L \in L} \max_{a \in Z} \frac{1}{L} \sum_{j \in J} F_j(a,L) \rceil \tag{9}$$

The bound holds for any choice of lengths; a richer set can only make it stronger. We write L_P and L_{NP} for its values with W fixed. The non-preemptive forced work is never smaller than the preemptive one, so $L_P \leq L_{NP}$.

Algorithm 1 evaluates the bound for every integer start a at once. For a fixed L, the forced work of a job, as a function of a, rises with slope c_j, stays flat at height $c_j h$ and falls with slope $-c_j$ (Figure 1(b)). The algorithm records the four changes of slope of every job in one array, and two prefix sums turn them into the total forced work at every integer start, which is the invariant of the listing. In the preemptive case every change of slope falls on an integer, so the maximum over integer starts is the maximum over all starts; in the non-preemptive case the peak of a job can fall

halfway between two integers, and rounding its height down keeps the values exact at the integers. Leaving out a start can only weaken the bound, never invalidate it, because Theorem 3 holds for every interval. Each length costs $O(n + T)$ time, with T the length of the horizon in seconds, and the whole bound $O(|L|(n + T))$; the cost is pseudo-polynomial, because T counts seconds, and a sweep over the changes of slope alone would remove the dependence on T. The demands of pool A include halves, so every sum is a multiple of one half and is exact in double-precision arithmetic, far below 2^{53}.

Upper bounds. A capacity is an upper bound on the optimum as soon as a schedule with that capacity meets the waiting bound. Algorithm 2 builds such schedules by greedy list scheduling in submission order. With a bound on the wait, the last allowed start of job j is $r_j + W$, so ordering the jobs by release time is ordering them by deadline on the start; among jobs released in the same second, the shortest starts first, because it frees its cores sooner.

Algorithm 1 Energetic lower bound on the capacity	
	Input: jobs (r_j, p_j, c_j), waiting bound W, interval lengths L, variant (P or NP)
	Output: an integer l with $l \le m_P^*(W)$ for P and $l \le m_{NP}^*(W)$ for NP
1:	$\beta \leftarrow 0$
2:	**for each** $L \in L$ **do**
3:	$D \leftarrow$ zero array indexed by the integers from $t_0 = min_j(r_j + W - L)$ to $max_j(r_j + p_j)$
4:	**for each** job j **do**
5:	$h \leftarrow min(L - W, p_j)$ for P; $h \leftarrow min(L, p_j, \lfloor (p_j + L - W)/2 \rfloor)$ for NP
6:	$x \leftarrow r_j + W - L$ in both variants
7:	**if** $h > 0$ **then** add $+c_j$ at x, $-c_j$ at $x + h$, $-c_j$ at $r_j + p_j - h$, $+c_j$ at $r_j + p_j$ to D
8:	$G \leftarrow$ prefix sum of the prefix sum of D ▷ invariant: $G[k] = \sum_j \quad F_j(t_0 + k + 1, L)$
9:	$\beta \leftarrow max(\beta, max_k\, G[k]/L)$
10:	**return** $l = \lceil \beta \rceil$

One queue per distinct demand lets a small job start when a larger one at the head of the line does not fit, and a heap holds the finish times of the running jobs. Unlike backfilling schedulers, which let small jobs move ahead only if they do not delay the first job in the queue or any queued job [32], the list holds no cores in reserve for a larger job. The schedule keeps three invariants: (i) at every event the running demand is at most m; (ii) no job starts before its release; and (iii) no released job that fits waits while cores are idle. A run costs $O(n \log n + nK)$ time, with K the number of distinct demands: 25, 5, 6 and 4 in pools A to D.

The capacity U_{NP} comes from a binary search on the integers between $max(L_{NP}, max_j\, c_j)$ and $m^*(0)$, with which the list starts every job at its release: it is a capacity whose list schedule meets the waiting bound while that of one core less does not, unless it is the lower end of the search. At W^* the search ran 13, 13, 10 and 11 simulations in pools A to D. The waits of a list schedule need not fall as cores are added, so the search may miss a smaller capacity that meets the bound; it needs no monotonicity to be valid, because it returns only capacities whose schedule was built and checked, and an independent verifier, which shares no code with the list scheduler, replays every returned schedule and checks the start times, the waits and the peak demand in $O(n \log n)$ time. A non-preemptive schedule is also preemptive, so $U_P = U_{NP}$, and the lower end of the interval of the value of suspension below is therefore zero.

Theorem 4. $L_P \le m_P^*(W) \le U_P$ and $L_{NP} \le m_{NP}^*(W) \le U_{NP}$. With $m^*(0)$ exact by Lemma 2, the two ratios of Equation (5) lie in the certified intervals of Equation (10). When $L_{NP} = U_{NP}$ the non-preemptive optimum is certified; otherwise the relative gap $(U_{NP} - L_{NP})/L_{NP}$ bounds the error of reporting U_{NP} as the capacity.

$$S(W) \in \left[1 - \frac{U_{NP}}{m^*(0)},\ 1 - \frac{L_{NP}}{m^*(0)}\right], V(W) \in \left[\frac{max(0,\ L_{NP} - U_P)}{U_{NP}},\ \frac{U_{NP} - L_P}{L_{NP}}\right] \tag{10}$$

Figure 2 works through both algorithms on a small instance, the first of a fixed sequence of random instances of 8 jobs, with a waiting bound of 3 units of time, in which the five capacities differ. The jobs with forced work in the interval I^* that attains L_{NP} must process 14 core-units in its 3 units of time, so every schedule needs at least 5 cores. The list schedule meets the waiting bound with 7 cores, against 8 without waiting; the optimum, found by enumeration, needs 6, and with suspension the lower bound falls to 4.

Algorithm 2 Greedy list scheduling in submission order with demand classes	
	Input: jobs sorted by (r_j, p_j), capacity m
	Output: start times S_j and the largest wait $max_j(S_j - r_j)$
1:	$f \leftarrow m$; $H \leftarrow$ empty min-heap of (finish time, demand); one queue Q_k per distinct demand
2:	append every job to the queue of its demand, in (r_j, p_j) order; $t \leftarrow min_j\, r_j$
3:	**while** some job has not started **do**
4:	pop from H every job that finishes at or before t and add its demand to f
5:	**while** some queue head j has $r_j \le t$ and $c_j \le f$ **do**
6:	take the head with the smallest (r_j, p_j); $S_j \leftarrow t$; $f \leftarrow f - c_j$; push $(t + p_j, c_j)$ to H
7:	$t \leftarrow$ the earlier of the next release and the earliest finish in H; if neither exists, **return** ∞ ▷ invariants (i)–(iii)
8:	**return** S and $max_j(S_j - r_j)$

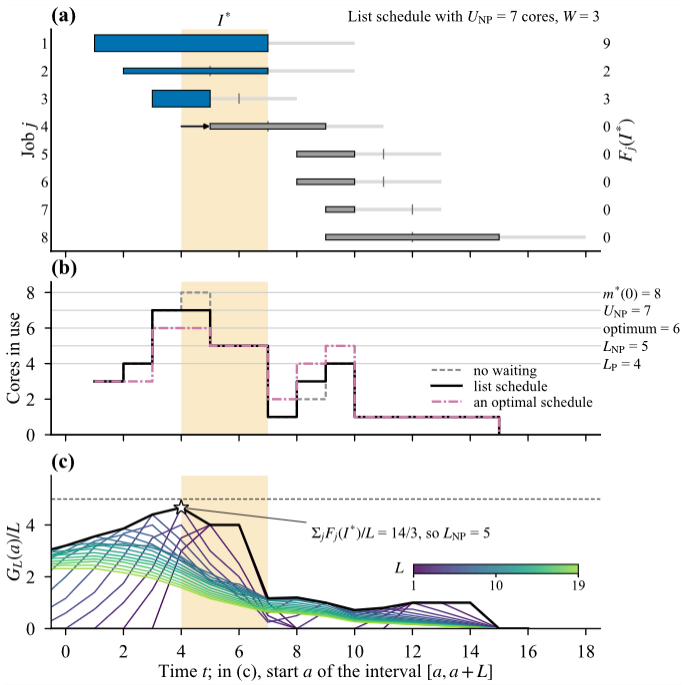


Figure 2. The two algorithms on a small instance, an illustration: the first of a fixed sequence of random instances of 8 jobs with integer times and a waiting bound of 3 in which the five capacities differ. (a) List schedule of Algorithm 2 with U_{NP} cores: for each job, its window $[r_j, r_j + p_j + W]$ (light gray, with a tick at its last start $r_j + W$), its run (bar of height proportional to c_j; in colour, the jobs with forced work in I^*) and its wait (arrow). The shaded interval $I^* = [a, a + L]$ attains L_{NP}, and the numbers at the right are the forced work $F_j^{NP}(I^*)$ of each job, in core-units. (b) Cores in use if every job starts at its release, in the list schedule and in an optimal schedule found by enumeration, against the five capacities, from $m^*(0)$ down to L_P. (c) Forced work per unit of length $G_L(a)/L$ of Algorithm 1 at every integer start a, one line per length L (colour), with their upper envelope (black); its maximum, at the start of I^* (star), rounded up, is L_{NP} (dashed).

Data. The Intel NetBatch logs of the Parallel Workloads Archive cover one month, November 2012, of four physical pools, A in Israel and B, C and D on the west coast of the United States [2]; they were collected at the managers of the pools [1]. They contain 48,821,850 accounting records, of which 13,368,191, 13,085,800, 13,313,793 and 9,054,066 belong to pools A to D. Each record has the submission, start and finish times in seconds, the time the run was suspended, the number of cores, an iteration number with its own submission time, and anonymized identifiers of the user, group, command and machine [2]; Table S17 lists every field and how this study uses it.

We used these original files rather than the archive's conversion to its standard workload format, which has no field for suspension and therefore loses it [2, 27]. The number of records of every pool matches the archive's conversion, and their total, the negative run times of all four pools and the fractional core counts of pool A match the figures on its page [2]; our counts of the other anomalies that it lists are close to its figures except for the records of pool C whose average processor time exceeds their duration, and the archive does not say how it counted them (Table S18). The archive does not give the number of cores of any pool, whose exact details have not been disclosed [2], so the installed capacity is unknown; the logs name from 1,453 distinct machines in pool A to 4,100 in pool B (Table 2), but not their cores. The times are local times of each pool, as the archive describes them, and all submission, start and finish times fall within November 2012 [2]. In pools B, C and D, the end of daylight saving time repeats one hour of local time early in the month: their 1,002, 257 and 730 records with a negative duration all start and end within that hour, and the records that run across it seem one hour shorter than they were. We kept the times as recorded, as the

plan fixed, and Results reports what a conversion to a monotone time scale changes. The archive notes that the log of pool D appears to have a couple of flurries [2]; they are kept, like every other burst, because a pool has to serve them.

A record describes one iteration of a job, and the iterations of a job run one after another: in every pair of consecutive iterations of pools A, C and D, and in all but 10 pairs of pool B, the next iteration is submitted after the previous one finishes, a median of 3 to 16 s later depending on the pool (Figure S18). In 98.8% of the pairs of consecutive iterations of pool B and 54.9% of those of pool A, both iterations carry the same job-level submission time. We therefore take as the release time r_j of a record the submission time of its iteration; the archive's own conversion likewise treats the iterations of a job as a chain in which each depends on the previous one [2]. With the job-level time, the iterations of a series would be released at once and would seem to need cores side by side. This choice was made before the plan was frozen, from this structure of the four pools and the convention of the archive, before any capacity of the sealed pools had been computed. The share of records whose previous iteration is in the log ranges from 0.3% in pool C to 45% in pool B, but most of these records have zero processing time: among the jobs of the model, only 2.6%, 1.5%, 0.03% and 2.1% in pools A to D follow a previous iteration in the log.

The processing time p_j of a record is its finish time minus its start time minus the time it was suspended, and its demand c_j is its number of cores. In pool A, 158,440 records request a number of cores that ends in one half, which the archive attributes to time slicing on the cores [2]; we keep these demands, because the cores of a pool are pooled. We excluded the records with a negative processing time, a start before the submission of their iteration or a non-positive number of cores; records with zero processing time occupy no core at any instant, and from 84.9% of them in pool C to 99.7% in pool B ended with a nonzero exit status, which the archive reads as a failure [2]. Table 2 gives the counts for each pool, and Figure 3 shows, as an exploratory description, where the work of each pool lies: jobs that run for at most W^* make up 88.3%, 81.5%, 91.3% and 76.2% of the jobs of pools A to D but only 17.3%, 9.7%, 9.4% and 7.1% of their core-seconds, and jobs that ask for at most one core hold 88.8% of the core-seconds of pool A and more in the others, from 96.3% in pool C to 99.8% in pool B. Because the logs contain only runs that were both submitted and completed within the month, the first days miss the load of runs submitted in the previous month and the last days miss long runs that ended in the next one. Release times are fixed in the model, as in any replay of a log, which ignores the feedback from the system to its users that workload models have to consider [28, 29]. Here, it also ignores that the iterations of a series would shift with the waits of their predecessors, and the release time of a job that follows a previous iteration carries the wait that the real scheduler gave to that iteration.

Table 2. Characteristics of the four NetBatch pools: records, exclusions and jobs analysed, the users and machines that appear in each log, and the processing times, demands and requested memory of the jobs.

	Pool A	**Pool B**	**Pool C**	**Pool D**
Records	13,368,191	13,085,800	13,313,793	9,054,066
Inconsistent[a]	1,974	1,306	309	1,042
Zero processing time	2,387,531	6,508,280	412,462	604,383
Jobs analysed	10,978,686	6,576,214	12,901,022	8,448,641
Previous iteration[b]	12.7%	44.9%	0.3%	3.3%
Fractional demand	158,440	0	0	0
Users[c]	1,104	1,614	1,146	862
Machines[c]	1,453	4,100	1,521	2,426

	Pool A	Pool B	Pool C	Pool D
Median p_j (s)[d]	243	342	64	487
p_j (s), 90th percentile[d]	4,404	8,021	2,813	14,778
p_j (s), 99th percentile[d]	38,659	79,767	47,131	93,479
Longest p_j (s)[d]	2,300,187	2,159,509	2,207,947	2,399,482
Jobs with $c_j \leq 1$ core[d]	97.3%	99.7%	99.2%	99.9%
Widest job (cores)[d]	89	8	16	8
Median memory (GB)[c]	2	2	1	2
Memory (GB), 99th percentile[c]	24	20	20	14

a Negative processing time, start before the submission of the iteration, or no cores.

b Share of the records whose previous iteration of the same job is in the log.

c Over all the records of the log: distinct identifiers of users and of machines, and the memory that each record requests for all its threads.

d Of the jobs analysed: processing time p_j and demand c_j, with percentiles by linear interpolation.

Pre-specified design. Before any capacity of pools A, B and D was computed, we froze an analysis plan with a tag in the version control of the project; the plan was not filed with a public registry. One pool, drawn at random with a fixed seed recorded with the partition, served for development: pool C, on which the algorithms were built and tested. The other three were sealed: until the plan was frozen, only their structure was audited, and afterwards every computation of their capacities opened them through a gate that checked the frozen plan and logged the access; the description of the iterations above read their structure directly. The plan fixed two one-sided hypotheses at the primary waiting bound W^* of 1 h: H1 states that the capacity saving $S(W^*)$ is below 10% in every sealed pool, and H2 that the value of suspension $V(W^*)$ is below 2% in every sealed pool. The thresholds are conventions fixed in the plan, not estimates: a saving below 10% was taken as too small to justify by itself a service agreement with waits, and a value of suspension below 2% as too small to pay for the delay of suspended jobs and the memory that they hold. Both were set by the intended use, before the sealed pools were opened and not from the values of the development pool. For each pool, a hypothesis is supported when the upper end of its certified interval in Equation (10) is below the threshold, rejected when the lower end reaches it, and undecided otherwise; because the lower end of the interval of V is zero, H2 can be supported or undecided but never rejected, so the design could show that suspension saves little but never that it saves much.

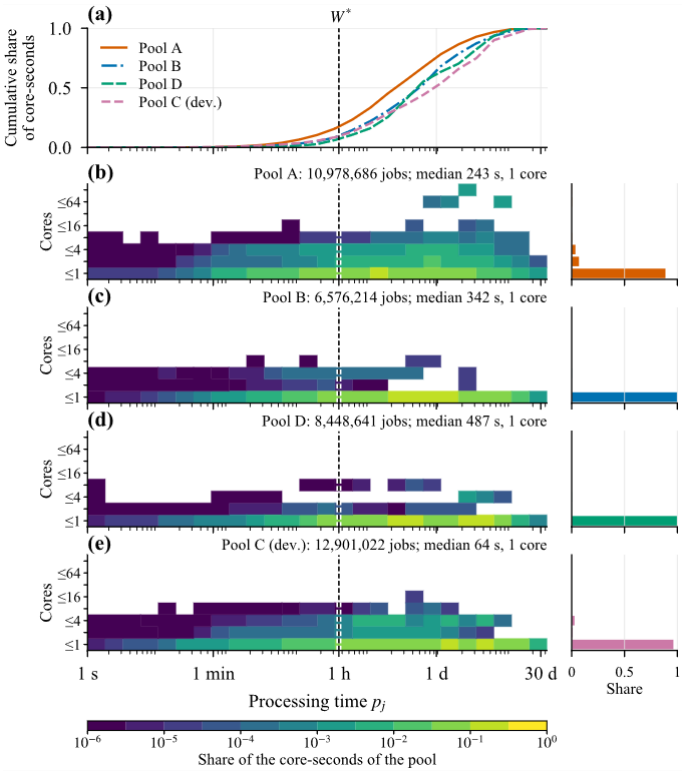


Figure 3. Where the work of each pool lies, an exploratory description of the jobs of the model. (a) Cumulative share of the core-seconds $\sum_j \quad c_j p_j$ of each pool held by its jobs with processing time up to p_j; the dashed line marks W^*. (b)–(e) Share of the core-seconds of each pool in each cell of processing time (four intervals per decade) and demand c_j, on a logarithmic colour scale (cividis; white cells hold no job), with the share of each demand class at the right; the headings give the number of jobs and the median processing time and demand. Pool C was used for development.

A hypothesis holds overall only if it is supported in all three pools; otherwise the plan reports a null result, which it calls informative when the intervals lie above the threshold. The decisions are also reported for other thresholds. The ends of every certified interval are printed rounded outwards, so that the printed interval contains the computed one. The quantities studied are the two ratios of the log of each pool for that month, as preprocessed, in the model. The logs are the complete record of the runs that were submitted and completed within the month, so these ratios carry no sampling error: there are no p-values, and the gap between the certified bounds is the only uncertainty of the computation. A month is, however, one realization of a workload that changes over time; how much the ratios depend on the period, on the truncation of the log at the ends of the month and on the preprocessing is assessed by the halves of the month, the robustness checks, the multiverse and the second system below.

A supported hypothesis counts as valid only if it passes six checks fixed in the plan: the independent verifier accepts every schedule that gives an upper bound; the bounds are ordered as Theorem 4 requires; on small instances, the exact optima lie inside the certified intervals; the method recovers known answers on three synthetic controls, namely a constant load with no saving, a burst of identical jobs with a large saving that must not be reported as small, and unit jobs for which suspension cannot help; the decision does not change when only the jobs submitted between 3 and 28 November are kept or when the jobs that waited more than a day are removed; and the capacity is computed from release times, processing times and demands alone, so the decisions of the real scheduler enter only through the processing times and, for the jobs that follow a previous iteration, through their release times. The plan applies these checks to supported hypotheses; the rejections are examined with the same variants, the halves of the month and the multiverse in Results. Days are counted in the local time of each pool.

The lower bounds of the pools use the set of interval lengths fixed in the plan: 120 lengths spaced geometrically from 1 s to 30 days, each about 13% longer than the previous one, rounded to whole seconds, together with $W + 1$; at W^* these are 111 distinct lengths, of which only the 55 longer than W^* can contribute to the preemptive bound. By Theorem 3 any such set gives a valid bound, and a denser one can only raise it.

The plan also fixed the secondary analyses, which run after the primary one in this order: certified frontiers for W from 0 to 24 h in all four pools, with pool C marked as the development pool; the value of suspension on the same grid; a comparison with the schedules that NetBatch actually produced; the two halves of the month as separate instances; a secondary analysis on the MetaCentrum2 log of the Czech national grid [33, 34], with one instance per month of submission from 2013 to 2014, built from the submission time, the run time and the allocated processors of each job in the archive's conversion, without the records that have a negative run time or wait or no processors; a resampling of whole days, with a fixed seed, which recombines the days of each month; a multiverse of the 48 combinations of defensible preprocessing choices; the exact baseline; and the benchmark. Some measurements were added after the plan was frozen and are reported as additions: the size limit of the integer programs, the stages of the benchmark beyond those of the plan, twenty repetitions instead of five for stages that take less than a second, the speedups over reference implementations and the memory curve. Three further descriptors, which the plan did not fix, are reported as exploratory: the length of the stretch of time that sets each lower bound, the jobs that keep the list schedule from meeting the bound with one core less than its capacity, and how much the lower bounds rise with denser sets of interval lengths. Added later, and also exploratory, are a check of every lower bound of the frontiers against the interval that attains it, the overlap of consecutive iterations in the list schedules, the sensitivity to the repeated hour of local time, and the load without waiting over the month.

Exact baseline. To measure how far the bounds and the list schedules are from the optimum, we solved a time-indexed integer program for the non-preemptive problem, with a binary variable for each job and admissible start minute, a capacity constraint for each minute and the capacity as the objective, using the Coin-or branch and cut (CBC) solver with a limit of 600 s per instance. The instances are the n consecutive jobs of pool C from the start of the interval that attains its lower bound at W^*, with times in minutes (releases rounded down, durations up), for W of 1, 5 and 60 min, and n doubles from 25 until the solver no longer closes within the limit, the model exceeds 12,000,000 nonzero coefficients, or n reaches 25,600. The instances of each waiting bound are therefore nested and start where the bound of the development pool is set, without the jobs already running there. The limit on the size of the model was added after the plan, so that it fits with a margin in the memory of the computer described below. The lower bound of the baseline uses every interval length from 1 to 3,999 min, a denser set than that of the pools. For unit demands and $n \leq 100$, a maximum-flow computation with binary search gives the preemptive optimum as well.

Implementation and measurement. The pipeline is written in Python 3.7 with NumPy 1.15 and pandas 0.23; the two algorithms are compiled with numba 0.41, and a slower reference implementation of Algorithm 1 in NumPy is kept for testing. The integer programs use PuLP 1.6 with CBC 2.9, and the flows use networkx 2.2; every dependency is pinned to a fixed version.

Property-based tests with Hypothesis check each lemma and theorem on random instances of up to eight jobs with small integer times, against optima found by enumerating every start time or by flows, and the verifier shares no code with the list scheduler. A separate reimplementation of the primary analysis was written from the text of the plan, the documentation of the data and the raw files, without the code of the pipeline, and its results were compared with those of the pipeline within tolerances fixed before they were seen; the limits of its independence are stated at the end of this section. All random draws use fixed seeds, which the result files record. Timings were taken on a desktop computer with 8 cores and 16 hardware threads at a 4.7 GHz base clock, 16 GB of memory and the Windows operating system in its high-performance power plan, with one thread per process. The benchmark measures the six stages that certify one pool at one waiting bound: loading the jobs, the capacity without waiting, the two lower bounds, the binary search with list schedules and the verifier. In each of at least five repetitions, twenty for stages that take less than a second, a fresh process runs a stage once to warm up, which includes the compilation of the algorithms, and once to

measure its wall-clock time, processor time and peak resident memory, and we report the median and the interquartile range; speedups are ratios of medians with percentile bootstrap intervals, which with so few repetitions are only indicative. The scaling curves use the jobs of pool C released in its first hour, its first four hours and its first one, four, sixteen and thirty days, and for memory each process loads only the jobs of its window. Every number in this paper is read from a result file that the pipeline of Figure S17 writes together with the commit, the hash of the data and the versions used, and automatic tests and checks must pass before the manuscript is packaged for submission. A second complete run of the pipeline, from the raw files and with the code of a single commit, gave the same value in all 8,287 compared fields of the result files, apart from time stamps, commits and measured times. We report the study following the pitfalls of parallel job scheduling evaluation [29] and the twelve rules for reporting performance results [35]; the completed checklists are Table S20 and Table S21.

3. Results

Primary analysis. All capacities and savings in this section are those of the pooled model on the logs of the month. Table 3 gives the certified bounds at the primary waiting bound W^*.In pool D, both hypotheses are supported: a waiting bound of 1 h would save between 4.0% and 4.4% of the capacity needed without waiting, and suspension at most 0.4% more. In pools A and B, H1 is rejected, because even the lower end of the saving exceeds the threshold: the same waiting bound would save at least 22.1% of the capacity in pool A and at least 17.9% in pool B. H2 is undecided in both pools: suspension would save at most 4.4% of the non-preemptive capacity in pool A and at most 7.1% in pool B, but the certified intervals reach down to zero and contain the threshold of 2%. Neither hypothesis is therefore supported overall: in the terms of the plan, the analysis gives a null result for both, an informative one for H1, whose savings exceed the threshold in two pools, and an undecided one for H2.

The two nulls differ in kind. The rejection of H1 is informative: in two of the three sealed pools, even the lower end of the saving at 1 h exceeds the threshold of 10%, although in pool B the second half of the month falls below it, as shown below. The undecided H2 is a limit of the certificate rather than a measured effect. The lower end of $V(W^*)$ is zero in every pool, because a non-preemptive schedule is also a preemptive one and therefore $U_P = U_{NP}$. Its upper end V^- is the sum of two parts, Equation (11): the gap γ between the list schedule and the non-preemptive lower bound, which is 4.2% in pool A and 4.8% in pool B, against 0.3% in pool D, and the distance δ between the two lower bounds, which is 0.18%, 2.21% and 0.04%:

$$V^- = \frac{U_{NP} - L_P}{L_{NP}} = \gamma + \delta \geq \delta, \gamma \qquad (11)$$
$$= \frac{U_{NP} - L_{NP}}{L_{NP}}, \qquad \delta = \frac{L_{NP} - L_P}{L_{NP}}$$

The smaller upper end $(U_{NP} - L_P)/U_{NP}$, which the Supplementary Material proves valid as well, is 4.2% in pool A and 6.7% in pool B, and changes no decision. In pool A, a better schedule could decide H2. In pool B, δ alone exceeds the threshold, and since $V^- \geq \delta$ the upper end cannot fall below it, whatever the schedule or the non-preemptive bound, unless the preemptive lower bound rises. The plan expected gaps of at most 1% and fixed no remedy for larger ones; those of pools A and B are larger. In pools A and B, the bounds rule out that suspension saves more than the upper ends above, but they cannot tell whether it saves less than 2%. An exploratory check shows that the set of interval lengths is not what keeps H2 undecided: a set four times denser and a local refinement around the best length of the plan raise the lower bounds at W^* by 2, 9 and 2 cores without preemption and by 13, 14 and 1 with it in pools A, B and D, which changes no decision; the upper end of $V(W^*)$ in pool B would fall only from 7.1% to 7.0%.

Simpler sizing rules. Figure 4 sets the certified capacity at W^* against the capacities that simpler rules would give, and Table S15 gives them in cores, as Equation (S12) defines them. In every pool, the mean load without waiting

lies below L_{NP}, so no schedule could meet a bound of 1 h with it, and the load without waiting exceeded during a hundredth of the month lies above U_{NP}: 13,592 against 13,009 cores in pool A and 19,441 against 18,707 in pool B.

Table 3. Certified capacities and ratios at the primary waiting bound W^*, with the decisions of the pre-specified rule.

Pool	$m^*(0)$	L_P	L_{NP}	U_{NP}	Gap[a]	$S(W^*)$	$V(W^*)$	H1b	H2b
A	16,716	12,465	12,488	13,009	4.17%	[22.1%, 25.3%]	[0.0%, 4.4%]	rejected	undecided
B	22,809	17,457	17,852	18,707	4.79%	[17.9%, 21.8%]	[0.0%, 7.1%]	rejected	undecided
D	25,998	24,867	24,877	24,947	0.28%	[4.0%, 4.4%]	[0.0%, 0.4%]	supported	supported
Cc	16,991	16,250	16,259	16,319	0.37%	[3.9%, 4.4%]	[0.0%, 0.5%]	–	–

a $(U_{NP} - L_{NP})/L_{NP}$.

b Decision of the pre-specified rule in each sealed pool.

c Development pool, not part of the confirmatory analysis.

In pools C and D, every rule from the load exceeded during a tenth of the month to the logged peak lies within a few percent of $m^*(0)$, whereas in pools A and B the choice of the rule moves the capacity by thousands of cores; even the peak of the load averaged over W^* lies above U_{NP}.

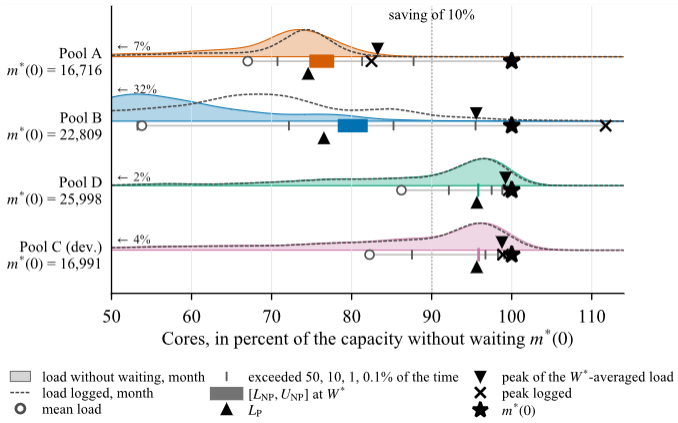


Figure 4. The certified capacity against the load of the month and simpler sizing rules, in percent of the capacity without waiting $m^*(0)$ of each pool, which the labels give in cores. Above each row, the distribution over the month of the maximum in each ten-minute interval of the load without waiting (filled) and of the load logged by NetBatch (dashed), as Gaussian kernel densities (Scott's rule, common height), with the share of the intervals left of the axis. On the row: the mean load without waiting (open circle); the loads without waiting exceeded during half, a tenth, a hundredth and a thousandth of the month (ticks, from left to right); the certified interval $[L_{NP}, U_{NP}]$ at W^* (bar) and L_P (triangle); the peak of the load averaged over W^*, Equation (S12) (inverted triangle); the peak of the cores in use logged by NetBatch (cross); and $m^*(0)$ (star). The dotted line marks the saving of 10% of the threshold of H1. Pool C was used for development.

Other thresholds. The overall decisions stay null at every threshold of the sensitivity analysis fixed in the plan (Table S3). For H1, pool A is rejected at every threshold, pool D is supported from 5% upwards and pool B becomes

undecided at 20%; for H2, pool D is supported at every threshold from 0.5% upwards, pool A only at 5%, and pool B at none.

Validity checks. Since no hypothesis is supported overall, the checks change no conclusion, but they qualify the decisions in each pool. The independent verifier accepted the schedule behind every upper bound, including those of the robustness variants; the bounds were ordered as Theorem 4 requires in every pool; and the capacities were computed from release times, processing times and demands alone. The separate reimplementation computed the 32 quantities that it shares with the pipeline (capacities without waiting, lower bounds, list capacities, ratios and excluded records in the four pools) and agreed exactly on 32 of them; the counts of jobs, which the reimplementation takes to include the jobs of zero length, were replaced in the comparison by the excluded records after the first comparison, on pool C, without changing any tolerance. Of the two robustness variants, the one that keeps only the jobs submitted between 3 and 28 November narrows the interval of $V(W^*)$ in pool A to [0, 1.9%], which would support H2 in that pool; every other decision is the same under both variants (Table S4). Had H2 been supported in pool A, this check would therefore have marked it as not valid.

The method recovered the known answers of the three synthetic controls (Figure S3). With a waiting bound of one hour, the constant load gave a saving of at most 0.5%; the burst of identical jobs gave its exact saving of 50% with a bound of ten minutes, and with one hour an interval of [85.7%, 86.7%] that contains its true saving of 85.7%; and for unit jobs, the interval of the value of suspension collapsed to zero at every waiting bound.

Checks added after the plan. Recomputing job by job the forced work in the interval that attains each lower bound, at every waiting bound of the grid, with Equations (7) and (8) and without the changes of slope of Algorithm 1, gives back every lower bound of the four pools (Table S13 and Table S14), so each of them can be checked from its interval alone in time linear in the number of jobs; at W^*, from 100,131 jobs in pool B to 2,128,975 in pool C contribute to these intervals (Figure S10). In the schedules behind U_{NP} at W^*, 2,070 of the 165,846 pairs of consecutive iterations with positive processing times overlap in pool A, 535 of 90,736 in pool B and 14 of 3,881 in pool D: the upper bounds hold for the model with fixed release times, in which an iteration may start before the previous one of its series ends, whereas the lower bounds hold as well when the iterations of a series must run in order, because that only adds constraints. Converting the times of pools B, C and D to a monotone scale, with the repeated hour of local time assigned to either side, leaves every capacity of pools C and D at W^* unchanged and lowers that of pool B without waiting from 22,809 to 22,807 cores and its list capacity from 18,707 to 18,706, which changes no decision.

Exact baseline. Figure 5 compares the certified bounds with the optima of the integer program on consecutive jobs of the development pool, and Table S1 gives every instance. With waiting bounds of 1 and 5 min, the solver proved the optimum at every size of the plan, up to 25,600 jobs; with 60 min it did so up to 6,400 jobs, and the model for 12,800 jobs exceeded the size limit. All 31 optima lie inside their certified intervals, and so do the preemptive optima that the flow computation gives for the 9 instances with unit demands. With every interval length, the lower bound equals the optimum in 30 of the 31 instances and is never more than 0.26% below it, whereas the list schedule exceeds the optimum by up to 0.9%, 22.2% and 28.9% with waiting bounds of 1, 5 and 60 min. With the coarser set of lengths used for the pools, carried over to minutes, the lower bound equals the optimum in 27 of the 31 instances and is never more than 0.39% below it, and the preemptive bound equals the preemptive optimum in 9 of the 9 instances with unit demands. On these instances, the gap of the certificate thus lies mostly on the side of the list schedule; the smallest of them, bursts of short jobs released together, are easy for the lower bound, and the instances are nested, so they are not independent cases. On one short stretch of the development pool, in minutes, suspension matters: with 100 jobs and a waiting bound of 5 min, the preemptive optimum is 32 cores against 46 without preemption, a value of suspension of 30%.

Capacity–waiting frontiers. Figure 6(a)–(d) shows the certified frontier of every pool over the pre-specified grid of waiting bounds, and the pools differ in how fast waiting pays. Between two consecutive bounds $W_k < W_{k+1}$

of the grid the frontier is certified as well, because by Lemma 1 the least capacity cannot grow with the bound and Theorem 4 brackets it at both ends, Equation (12), and likewise with preemption:

$$L_{NP}(W_{k+1}) \le m^*_{NP}(W) \le U_{NP}(W_k)\ for\ W_k \le W \le W_{k+1} \tag{12}$$

In pool A, a bound of 15 min would already save between 20.6% and 22.2% of the capacity; longer bounds add little to the list schedule, whose capacity a single wide job sets (see the Discussion), and the interval widens to [25.5%, 32.5%] at 24 h. In pool B, the saving grows steadily with the bound, from [8.4%, 10.8%] at 15 min to [44.1%, 45.5%] at 24 h. In pools C and D it grows slowly with the bound: at 24 h it is at most 15.2% and 10.7%. The upper end of the value of suspension (Figure 6(e)) is below the threshold of H2 at 9 of the 9 positive bounds in pool C and at 8 in pool D, with maxima of 1.8% and 2.4%; in pools A and B it is above the threshold at every bound, with maxima of 11.5% at 5 min and 7.1% at 60 min.

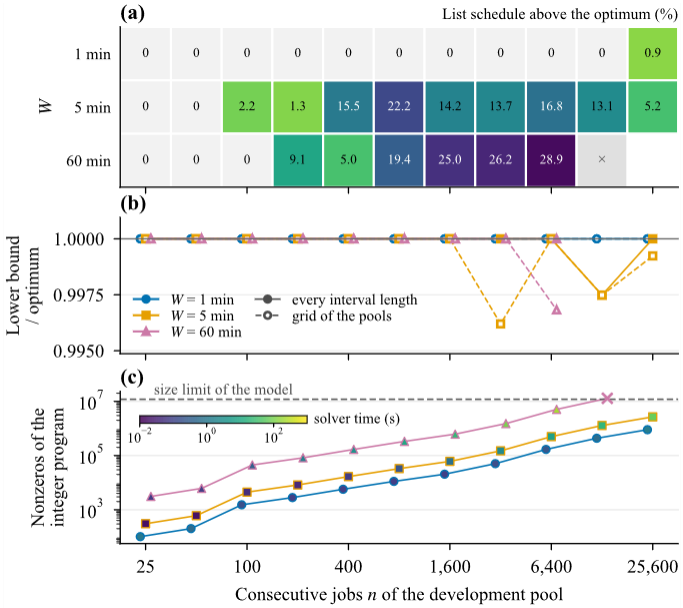


Figure 5. Exact baseline on consecutive jobs of the development pool, with times in minutes, for the instances whose optimum the solver proved. (a) How much the capacity of the list schedule exceeds the optimum of the integer program, in percent, in each cell of waiting bound and number of jobs (darker is larger); the cross marks the model that exceeded the size limit, after which no larger model was built for that waiting bound. (b) Lower bound divided by the optimum, with every interval length (filled markers) and with the grid of lengths used for the pools (open markers), on an enlarged axis: the bounds stay within 0.26% of the optimum. (c) Nonzero coefficients of each integer program built, on a logarithmic axis, against the size limit (dashed line), with each marker coloured by the wall-clock time that the solver took to prove the optimum (cividis, logarithmic scale); the cross marks the model that exceeded the limit. The series of each waiting bound are slightly offset in n so that they do not hide each other; the exact values are in the supplementary material.

Where the bounds are set. Figure 7 draws, as an exploratory description, what Algorithm 1 searches at W^*: for every start and length of an interval of the month, the forced work per unit of length, relative to L_{NP}. In pools C and D the bound rests on a plateau: the intervals that reach 98% of it start on 10 and 18 different days of the month, and the one that sets it lasts 99 h. In pools A and B it rests on a single stretch, and such intervals start on 2 and 2 days only. With suspension, the bound of pools A and B is set by longer intervals, of 60 and 15 h against 41 and 9 h without it (Figure 7(e)), whereas in pools C and D both bounds are set by intervals of the same length. The supplementary

material draws the load of every pool over the month and the burst, far above a level that the load keeps for days, that sets the capacity without waiting in pool A (Figure S12).

What NetBatch did. Counting every job as busy from its logged start to its logged finish (Figure S13), the logged peak of the month had 13,788.5 cores in use in pool A, against a capacity without waiting of 16,716; with that many cores, the list schedule keeps every wait within 644 s. Measured from the submission of each iteration, half of the logged waits were at most 15, 13, 4 and 8 s in pools A to D, and their 99th percentiles were 9,826, 12,789, 231 and 1,546 s. In pools C and D, the logged peaks (16,806 and 25,902 cores) come close to the capacities without waiting (16,991 and 25,998), and with those capacities the list schedule keeps every wait within 57 and 61 s. In pool B, the logged peak (25,487 cores) exceeds the capacity without waiting (22,809); delays can stack more work at one instant than a schedule without them, and the count includes the time that jobs spent suspended, which the logs do not place within a run and which the capacity without waiting leaves out. The logs record a suspension for 0.22%, 1.59%, 0.12% and 0.14% of the records of pools A to D. The verified schedules that give U_{NP} at W^* use the waiting bound very differently (Figure 8 and Table S16): only 12.4% and 2.7% of the jobs of pools A and B wait at all, in short bursts, whereas 28.7% and 52.4% of those of pools C and D wait, for days on end. NetBatch, for its part, logged a positive wait for 99.5% to 99.9% of the jobs of each pool, most of them of a few seconds, as the medians above show.

Service levels. Figure 9 shows the waits of the list schedule as cores are removed below the capacity without waiting. With U_{NP} cores, the median wait is zero in pools A, B and C and 151 s in pool D. The capacity U_q with which a share q of the jobs starts within W is the one that a binary search over list schedules returns, as for U_{NP}, Equation (13), where $w_j^{(m)} = S_j - r_j$ is the wait of job j in the list schedule with m cores and Q_q the q-quantile of the waits over the jobs, with linear interpolation:

$$Q_q(w^{(U_q)}) \le W \quad and\ Q_q(w^{(U_q-1)}) > W \ \ if\ U_q > \lceil \max_j c_j \rceil \tag{13}$$

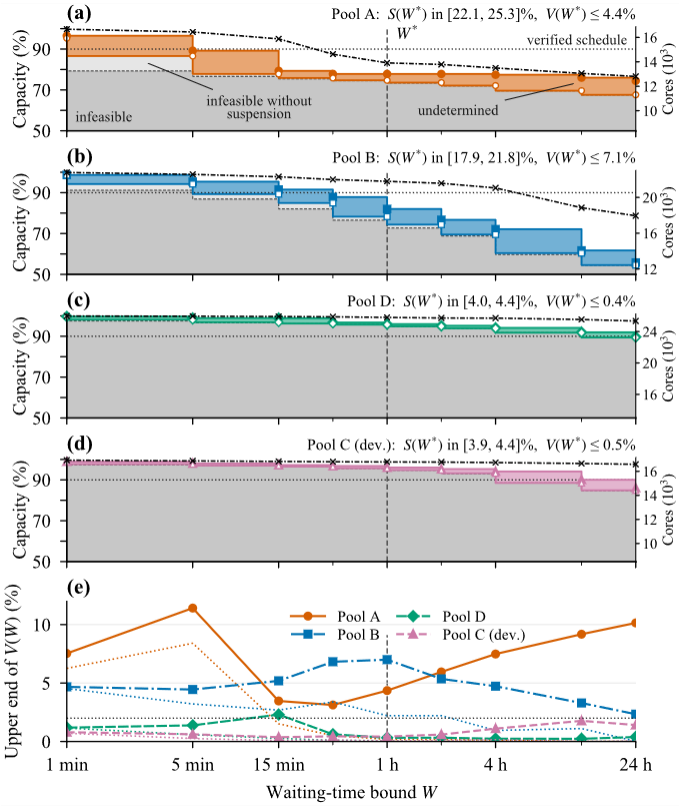


Figure 6. Certified Pareto front between capacity and waiting-time bound. (a)–(d) For each pool, capacity as a percentage of the capacity without waiting $m^*(0)$ (left axis) and in thousands of cores (right axis) against the waiting bound W, with what the bounds of the pre-specified grid certify: above U_{NP} (filled markers) a verified schedule exists; between U_{NP} and L_{NP} (open markers) the minimum capacity is undetermined (colour of the pool); between L_{NP} and L_P (dashed) no schedule without suspension meets W (light gray), and below L_P no schedule does (gray). Between two points of the grid the regions follow from Equation (12). The dash-dotted line is the peak of the load averaged over W (Equation (S12)), above U_{NP} at every bound. The saving $S(W)$ is the distance from the top of the panel, and the dotted line marks the threshold of H1, a saving of 10%; the headings give the certified intervals at W^* (dashed vertical line). (e) Upper end of the value of suspension $V(W)$ (markers), whose lower end is zero; the thin dotted line of each pool is the distance between the two lower bounds, $(L_{NP} - L_P)/L_{NP}$, and the rest is the gap of the list schedule; the gray dotted line is the threshold of H2. Pool C was used for development.

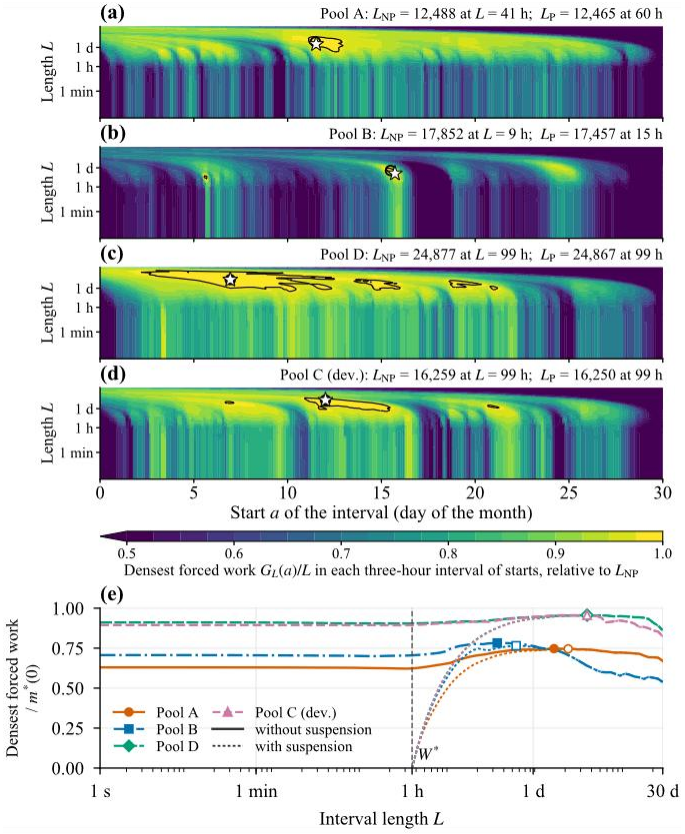


Figure 7. Energetic landscape of each pool at W^*, an exploratory description of what Algorithm 1 maximizes. (a)–(d) For every three-hour interval of starts a in the month and every length L of the pre-specified grid, the largest forced work per unit of length without suspension, $G_L(a)/L$, divided by L_{NP} (cividis; values below one half take the darkest colour). The black contour encloses the intervals that reach 98% of the bound, the star marks the interval $[a, a + L]$ that attains L_{NP} and the circle the one that attains L_P; intervals that run past the end of the logs hold less forced work, which darkens the upper right. (e) Largest forced work per unit of length over all starts, $max_a\, G_L(a)/L$, without suspension (solid) and with it (dotted), as a fraction of $m^*(0)$; their maxima, marked, are L_{NP} and L_P, and with suspension no work is forced in intervals no longer than W^*. Pool C was used for development.

If one job in a hundred may wait longer than W^*, the list schedule needs 12,526 cores in pool A and 17,923 in pool B, against 13,009 and 18,707 when every job must start in time, but 16,300 and 24,906 in pools C and D, against 16,319 and 24,947: where the load stays near its peak for days, a percentile of the waits buys almost nothing. These

capacities come from list schedules alone, without a lower bound for a share of the jobs; Table S2 gives them at every waiting bound of the grid.

Halves of the month. Split by submission date (Table S4), the first half of the month reproduces the decisions of the whole month in every pool, and in pools A and D exactly its capacities, because it contains the moment of the capacity without waiting and the stretch that sets the lower bound. The second half gives pools A and B much smaller savings at W^*, [15.3%, 16.1%] and [8.6%, 9.3%], and much narrower intervals for the value of suspension, whose upper ends of 1.0% and 1.2% would support H2 in both pools; in pool B, H1 would be supported as well. In pools C and D, both ratios stay below the thresholds in both halves. Like the month, each half starts without the jobs submitted before it, and over a shorter window its ends weigh more; the stretch that sets the lower bound of pool B for the whole month lies in the second half but also holds jobs submitted in the first, because the second half alone gives a lower bound below that of the month.

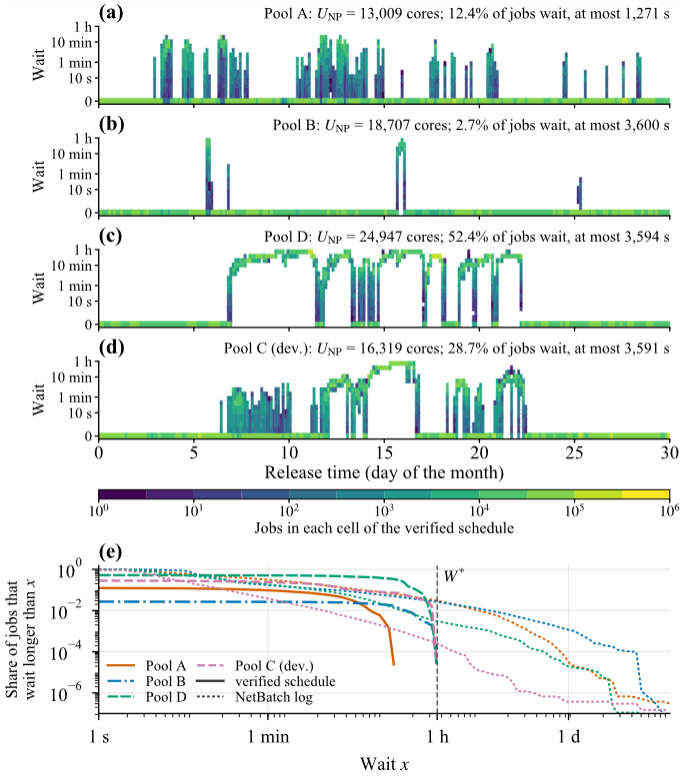


Figure 8. Waits in the verified schedules behind U_{NP} at W^*, an exploratory description. (a)–(d) Number of jobs of each pool in each cell of release time (three-hour intervals) and wait in the list schedule with U_{NP} cores (logarithmic intervals up to W^*; the bottom row holds the jobs that start at their release), on a logarithmic colour scale (cividis; white cells hold no job); the headings give the share of jobs that wait and the longest wait. (e) Share of the jobs that wait longer than each value, on log–log axes, in the verified schedule (solid) and in the logs of NetBatch, measured from the submission of each iteration (dotted). Pool C was used for development.

Weekly windows. Certifying every window of 7 days of releases, advancing one day at a time (Figure 10; every window in Table S5 and Table S6), shows how the answer moves through the month. H1 is rejected in all 24 windows of pool A and in 15 of pool B, supported in 4 windows of pool B, all of which start in the second half of the month, and supported in every window of pools C and D, where the upper end of the saving stays below 9.2% and 8.3%. H2 would be supported in 12, 6, 19 and 22 of the windows of pools A to D. Like the halves, each window starts without the jobs released before it.

Second system. On the MetaCentrum2 log, which the model treats as a single pool of all the cores of a national grid of clusters, a waiting bound of 1 h would save less than 10% of the capacity in 6 of the 24 months, at least 10% in 16 and an undecided amount in 2. H2 is supported in 0 of the 24 months: the upper end of the value of suspension ranges from 11.3% to 44.2%, and in 24 months the distance between the two lower bounds alone, from 10.8% to 34.9%, exceeds the threshold, and is larger than the gap of the list schedule in 22 of them. As in pools A and B, a bound of one hour buys a large part of the capacity in most months of the second system, and the certificate cannot bound the value of suspension below the threshold; Table S10 gives every month, with both lower bounds and the gap. The archive describes a configuration of 19 clusters with 8,412 cores, and some jobs ran on further clusters [33]; the capacities without waiting of 22 of the 24 months exceed it, and so do the lower bounds at W^* of 19 (Figure S6), which shows that the real system served bursts by making jobs wait. The archive also warns that the log contains large flurries of jobs that have not been cleaned [33]; we used it as converted, so this analysis says more about the method than about a second industrial farm.

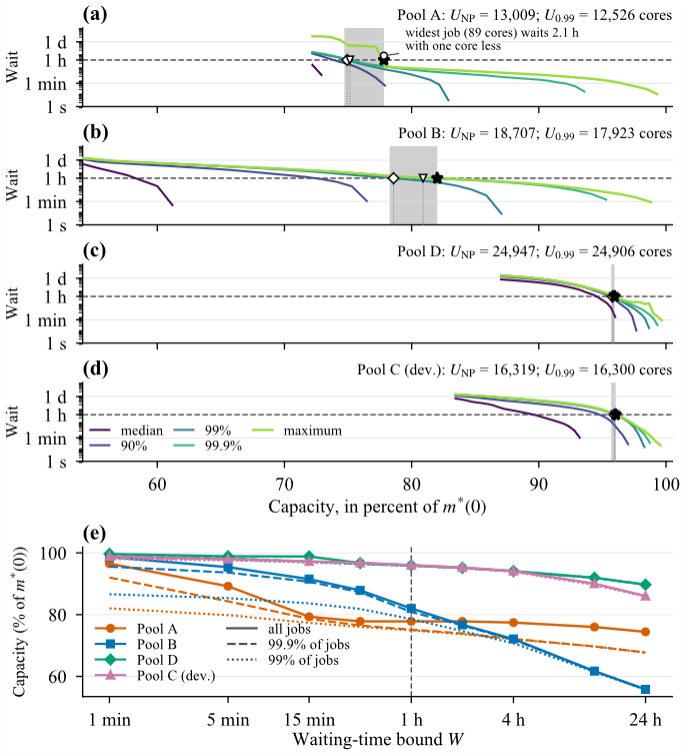


Figure 9. Waits of the list schedule of Algorithm 2 against its capacity, an exploratory description. (a)–(d) For each pool and capacities from just below the certified one to $m^*(0)$, in percent of $m^*(0)$: the median wait, the waits exceeded by one job in ten, in a hundred and in a thousand, and the longest wait, on a logarithmic axis (a wait of zero falls off the axis); the gray band is the certified capacity at W^* and the dashed line W^*, on which the markers give the capacities U_q of Equation (13) for every job (star, U_{NP}), all but one in a thousand (triangle) and all but one in a hundred (diamond), the first and the last in cores in the headings; in (a), the circle is the wait of the widest job with one core less (Figure S16). (e) Capacity that the list schedule needs, against the waiting bound W, so that every job starts within W (U_{NP}, solid), all but one job in a thousand (dashed) or all but one in a hundred (dotted); the last two come from binary searches over list schedules and are upper bounds only, without a matching lower bound. Pool C was used for development.

Resampling of days. Rebuilding each month from whole days drawn at random with replacement, 30 times, moves the upper end of $S(W^*)$ within a 95% percentile interval of [17.4%, 25.0%] in pool A and [12.4%, 23.9%] in pool B, which with so few resampled months lies close to their range, and it reaches the threshold of H1 in 30 of the

30 resampled months of pool A and 30 of those of pool B. In pools C and D, the intervals are [5.7%, 16.4%] and [5.8%, 13.9%], and the upper end reaches the threshold in 13 and 9 resampled months. The resampled months do not surround the observed one: in pools C and D, 30 and 30 of the 30 have a larger upper end of $S(W^*)$ than the observed month, and in pool A 29 have a smaller one. The resampling therefore describes months whose days follow one another at random, which break up the busy stretches of the observed month, rather than an interval around the observed value; in pool D, 9 of the resampled months would not support H1.

Multiverse. Figure S5 shows the 48 combinations of the preprocessing choices of the plan in every pool. H1 is rejected in all 48 specifications of pool A and all 48 of pool B, and supported in all 48 of pool D; H2 is undecided in all 48 specifications of pool B and supported in all 48 of pool D. In pool A, H2 would be supported in 8 specifications, all of which shorten the window of the month while keeping the fractional demands and the processing times without suspension, as in the robustness check above.

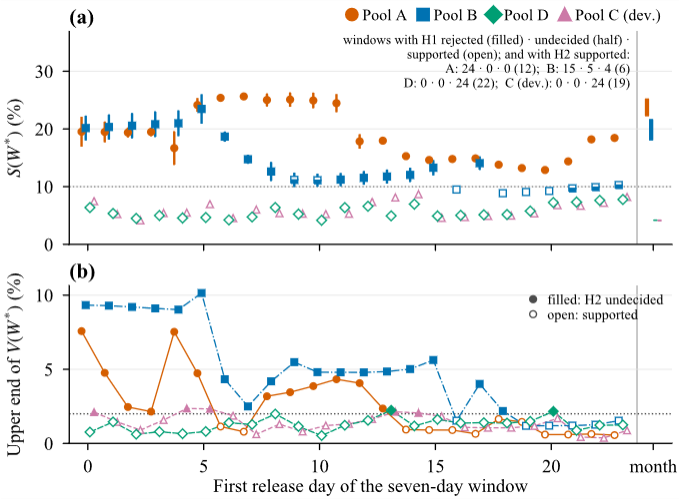


Figure 10. The certificate on windows of seven days of releases that advance one day at a time, an exploratory description. (a) Certified interval of $S(W^*)$ for each window and pool, with its centre filled, half filled or open as H1 is rejected, undecided or supported, and at the right the interval of the whole month; the dotted line is the threshold of H1, and the box gives the number of windows with each decision in each pool and, in parentheses, the number in which H2 is supported. (b) Upper end of $V(W^*)$ for the same windows, filled when H2 is undecided and open when it is supported, with the threshold of H2. Like the halves of the month, each window starts without the jobs released before it. Pool C was used for development.

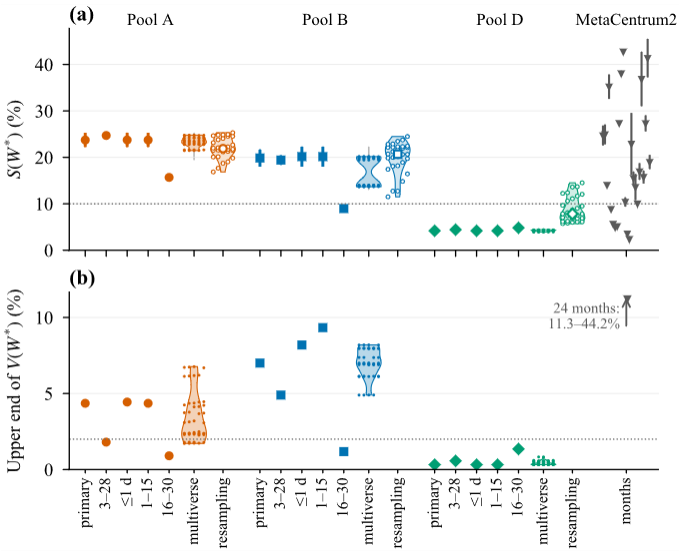


Figure 11. All analyses at the primary waiting bound W^*. (a) Certified interval of the saving $S(W^*)$, with the marker of the pool at its centre, for the primary analysis, the robustness variants that keep only the jobs released on days 3–28 (3–28) or drop the jobs that waited more than a day (≤1 d), and the two halves of the month (1–15, 16–30); for the multiverse, the centre of the interval of each specification (dots), with a violin of their density (Gaussian kernel, Scott's rule, cut at the extreme values), and the range of all its intervals (gray line); for the resampling of whole days, the upper end of $S(W^*)$ of each resampled month (open dots) over its violin, with its percentile interval (dotted) and its median (open marker); and for MetaCentrum2, the certified interval of each month in chronological order. (b) Upper end of the value of suspension $V(W^*)$ for the same analyses; the axis is cut above the values of NetBatch, and the months of MetaCentrum2 continue beyond the arrow. Resampled months have no value of suspension. The dotted lines are the thresholds of H1 and H2.

In pool B, the specifications that count the time spent suspended as processing give savings between 12.9% and 14.8%, against 17.9% to 22.4% without it, still above the threshold of H1. In the development pool C, both ratios stay below the thresholds in all 48 specifications. The overall decision is therefore null for both hypotheses in every specification. Some options change no job of a pool, so several specifications give the same result: the 48 specifications give 48, 24, 6 and 17 different intervals for pools A to D. Figure 11 sets all the analyses at W^* side by side.

Computational cost. On the whole month of pool C, with 12,901,022 jobs, the six stages that certify one waiting bound took 31 s with one thread, of which 11.2 and 5.6 s went to the two lower bounds and 10.7 s to the binary search with list schedules (Table S11). On windows of the same pool, one simulation of Algorithm 2 grows with a slope of 1.03 (1.01–1.05) on log–log axes, as expected of its $O(n \log n + nK)$ cost, and one interval length of Algorithm 1 with a slope of 0.96 (0.94–0.97) against $n + T$; memory grows linearly, by 72 bytes per job for a process that runs a simulation (Figure S8). The integer program of the exact baseline, by contrast, exceeded its size limit at 12,800 jobs with a waiting bound of 60 min.

4. Discussion

What a waiting bound buys. How much capacity a waiting bound saves depends on the pool as well as on the bound. In pools A and B, a bound of 1 h would save at least 17.9% of the capacity sized for no waiting, and in pool A a bound of 5 min already at least 10.8%, which suggests that its peak without waiting is a short burst that a wait of minutes spreads out. In pools C and D, an hour would save at most 4.4% in pool C and 4.4% in pool D, and a day at most 15.2% and 10.7%. An exploratory reading of the certificate is consistent with this (Figure 7): the stretch of time that sets the lower bound at W^* has the length of 99 h in pools C and D, against 41 h in pool A and 9 h in pool B,

which suggests a load that stays close to its peak for days in pools C and D; these lengths are those of the grid, whose consecutive lengths differ by about 13%. The second system behaves like pools A and B in most months, although its model pools a whole grid. For capacity planning, the frontier gives the price of a waiting bound in pooled cores before any scheduler is chosen: in the model, a pool sized for its peak without waiting holds at least 28.4% more cores than an hour of waiting requires in pool A and at least 21.9% more in pool B, but at most 4.6% more in pool D, and only the log of the pool tells which case applies. These are ratios of two capacities of the model, not bounds on the saving of a real pool: machines and memory would add cores to both capacities, so the saving of a real pool with the same jobs would probably be smaller. The capacity without waiting is neither the installed capacity, which the archive does not give, nor a configuration that any operator runs; turning a frontier into a purchase needs the installed capacity, the cost of a core against that of a delay, and the load of more than one month, since in these logs even the half of the month changes the answer.

Why H2 stays undecided. The upper end of the certified value of suspension adds the gap of the list schedule to the distance between the two lower bounds, and in pools A and B the sum is wider than the threshold of H2. In pool A the gap of the list schedule dominates; in pool B the distance between the lower bounds alone exceeds the threshold, so deciding H2 there needs a stronger preemptive lower bound and not only a better schedule. In every pool, the certificate contains no preemptive schedule, so it could never show that suspension saves much. On the instances of the exact baseline, the gap lay mostly on the side of the list schedule, and an exploratory look at the list schedules of the sealed pools shows what sets their capacities. In pool A, a single job keeps the list schedule from meeting the bound with one core less than its 13,009 cores: the widest job of the month, which ran on 89 cores for 79 h while the median job used 1 core, and which then waits 2.1 h (Figure S16); with 13,009 cores, no job waits more than 1,271 s. The same job keeps the list capacity of pool A constant over several waiting bounds of the grid (Table S2), so the flat part of its frontier is partly a property of the list schedule rather than of the load. A greedy list lets narrower jobs take the cores that free up, so a wide job waits until enough of them are free at once; a schedule that held cores for it, as backfilling schedulers do for the first job in the queue [32] and as the pool manager of NetBatch does when it reserves a machine for a job that it cannot yet match [1], might meet the bound with fewer cores. In pools B and D, by contrast, the list schedules sit at the bound: with one core less, 16 and 23 jobs of at most one core each exceed the bound, by at most 2 and 11 s. In pool A, a schedule that treats wide jobs better could therefore narrow the interval of $V(W^{*})$; in pool B, the certificate cannot tell whether the gap of the list schedule lies in the schedule or in the non-preemptive bound, and the preemptive bound would have to rise as well.

The real scheduler. The logs also show how far NetBatch was from these yardsticks, with the caveat that the model leaves out the machines, the memory and the priorities that NetBatch had to respect. At its logged peak, pool A had 13,788.5 cores in use, more than the 13,009 cores with which the list schedule meets a bound of one hour, and with that many cores the list schedule keeps every wait within 644 s, whereas the 99th percentile of the logged waits was 9,826 s. Unless the logged peak overstates the cores of the pool, which cannot be ruled out because our count treats suspended jobs as busy, this is consistent with long logged waits that come from what the pooled model leaves out, such as machines, memory, priorities and the requirements of the jobs, or from scheduling online, rather than from the total number of cores; the logs alone cannot tell these explanations apart. In pools C and D, NetBatch ran close to the capacity without waiting, and half of the jobs started within seconds.

Limitations and further work. Five limits of this study point to further work. First, the upper side of the certificate is a greedy list schedule without reservations, which the exact baseline shows to be its weak side on small instances and which a single wide job sets in pool A; backfilling with reservations, other priority rules or local search could narrow the intervals, and an exact maximum of the energetic bound over all intervals could tighten their lower side. Second, the certificate contains no preemptive schedule, so it can show that suspension saves little but never that it saves much; measuring a large value would need a preemptive upper bound, such as a schedule that rounds a relaxation, and in pool B and on the second system a stronger preemptive lower bound as well. Third, pooling the cores of a pool ignores the machines and the memory; bounds that respect them would combine the packing of jobs into machines with the timing studied here. Fourth, release times are fixed, so the feedback from waits to the next

submissions of a series is not modelled, and the list schedules let a few consecutive iterations of a series overlap; schedules that keep each series in order, or releases that follow the simulated end of the previous iteration, would remove both. Fifth, a single waiting bound for all jobs over one month of each pool ignores priority classes, percentiles of the waits and the change of the load over time; frontiers per class or for a share of the jobs, over several months, and online policies measured against them are natural next questions.

Threats to validity. *Construct validity.* The certified quantity is the capacity of a clairvoyant scheduler that knows every job in advance and pools the cores of all the machines of a pool. The logs record the machine that ran each job and its memory [2], but the model uses neither: it ignores how free cores are fragmented among machines and how much memory jobs hold.

Any schedule of a real pool that meets the same waiting bound with the same release and processing times is also feasible in the model, so the capacities are lower bounds on what a real pool would need for those jobs; the savings compare two such bounds and do not bound the saving of a real pool. The processing times are those observed under the load of the real pool, and the release times are fixed, as discussed in Materials and Methods. *Internal validity.* The confirmatory pools were sealed until the analysis plan was frozen, and their capacities were then computed only through a gate that logged every access, although the description of the iterations read their structure directly; errors of implementation are checked by the independent verifier, the property-based tests, the exact baseline, the recomputation of every lower bound from its interval and the separate reimplementation, whose independence is limited as stated in Materials and Methods; and the preprocessing choices are varied in the multiverse. *External validity.* The study covers one month of four pools of one company and one other system, whose model pools a whole grid. Workloads with other mixes of jobs, or users who change their behaviour when waits are bounded, may respond differently, and a single waiting bound for all jobs ignores priority classes. *Conclusion validity.* The logs contain one record per completed job [2], so the ratios of each log carry no sampling error for that month. Besides the gap of the certificate, which the intervals report, they depend on the period, which the halves of the month change, on the truncation of the log at the ends of the month, which changes the decision on H2 in pool A, and on the preprocessing, which the multiverse varies, whereas the conversion of the local times changes no decision. The rule that a hypothesis must hold in all three sealed pools makes a supported hypothesis demanding, but it protects no rejection; the decisions are also reported for other thresholds; and the resampling of whole days shows how the bound changes when the days of a month are recombined.

5. Conclusions

We bracketed how many cores four production pools of an industrial compute farm would need under a bound on job waiting times, for a clairvoyant scheduler that pools the cores of each pool, between energetic lower bounds and verified list schedules on logs of millions of jobs, in 31 s for a month of the development pool at one waiting bound with one thread. Neither pre-specified hypothesis was supported. In the model, a bound of one hour would save at least 17.9% of the capacity in two of the three sealed pools and at most 4.4% in the third, so the value of tolerating waits differs between pools and has to be measured pool by pool and over time; in these logs, even the half of the month changes it. In the primary analysis, the value of suspension could be bounded below the threshold only in the pool where waiting saves little, and the bounds could never have shown a large value. The frontiers turn a waiting bound into a number of pooled cores, a lower bound on what real machines would need for the same jobs, before any scheduler is chosen. On small instances the gap of the certificate lay mostly in the list schedules, and in one sealed pool a single wide job sets the list capacity, so schedules that hold cores for wide jobs are the next step towards precision, although deciding the value of suspension also needs a preemptive schedule and, in one of the pools, a stronger preemptive bound.

6. Funding

This research received no external funding.

7. Conflict of interest

The author declares that there is no conflict of interest.

8. Data availability

The Intel NetBatch and MetaCentrum2 accounting logs are publicly available from the Parallel Workloads Archive [2, 33]. The Intel NetBatch workload logs were graciously provided by Ohad Shai, Edi Shmueli and Nir Antebi from Intel, and the MetaCentrum workload log by the Czech National Grid Infrastructure MetaCentrum; the archive asks users to acknowledge the donors in this way and to cite the papers that describe the logs, as this paper does. The files used and their SHA-256 digests are listed in Table S19. The logs carry no formal licence and are not redistributed with this paper.

9. Code availability

The code that produces every number, table and figure, the frozen analysis plan with the log of the accesses to the sealed pools, and the result files are available from the author on request.

10. Ethics statement

The study analyses publicly available accounting logs of computing jobs, whose identifiers of users, groups, commands and machines were anonymized by their providers; the analysis uses no information about individual users. It involves no human participants and no animals, so no ethics approval was required.

REFERENCES

1. O. Shai, E. Shmueli, D. G. Feitelson. “Heuristics for Resource Matching in Intel’s Compute Farm”, in *Job Scheduling Strategies for Parallel Processing*, N. Desai, W. Cirne (eds.), Springer, Berlin, Heidelberg, pp. 116–135, 2014. https://doi.org/10.1007/978-3-662-43779-7_7
2. Parallel Workloads Archive, “The Intel Netbatch logs”, 2013. Available at: http://www.cs.huji.ac.il/labs/parallel/workload/l_intel_netbatch/index.html
3. Z. Zhang, L. T. X. Phan, G. Tan, S. Jain, H. Duong, B. T. Loo, et al. “On the feasibility of dynamic rescheduling on the Intel Distributed Computing Platform”. In *Proceedings of the 11th International Middleware Conference Industrial track*, pp. 4–10, 2010. https://doi.org/10.1145/1891719.1891720
4. S. Borst, A. Mandelbaum, M. I. Reiman. “Dimensioning Large Call Centers”, *Operations Research*, 52 (1), pp. 17–34, 2004. https://doi.org/10.1287/opre.1030.0081
5. M. Lin, A. Wierman, L. L. H. Andrew, E. Thereska. “Dynamic Right-Sizing for Power-Proportional Data Centers”, *IEEE/ACM Transactions on Networking*, 21 (5), pp. 1378–1391, 2013. https://doi.org/10.1109/TNET.2012.2226216
6. S. Albers, J. Quedenfeld. “Optimal Algorithms for Right-Sizing Data Centers”. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pp. 363–372, 2018. https://doi.org/10.1145/3210377.3210385
7. A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes. “Large-scale cluster management at Google with Borg”. In *Proceedings of the Tenth European Conference on Computer Systems*, pp. 1–17, 2015. https://doi.org/10.1145/2741948.2741964
8. Z. Dong, N. Liu, R. Rojas-Cessa. “Greedy Scheduling of Tasks with Time Constraints for Energy-Efficient Cloud-Computing Data Centers”, *Journal of Cloud Computing: Advances, Systems and Applications*, 4 (1), article 5, 2015. https://doi.org/10.1186/s13677-015-0031-y
9. D. Gmach, J. Rolia, L. Cherkasova, A. Kemper. “Capacity Management and Demand Prediction for Next Generation Data Centers”. In *IEEE International Conference on Web Services (ICWS 2007)*, pp. 43–50, 2007. https://doi.org/10.1109/ICWS.2007.62
10. S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, et al. “Morpheus: Towards Automated SLOs for Enterprise Clusters”. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 117–134, 2016. Available at: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/jyothi
11. R. Kettimuthu, V. Subramani, S. Srinivasan, T. Gopalsamy, D. K. Panda, P. Sadayappan. “Selective preemption strategies for parallel job scheduling”, *International Journal of High Performance Computing and Networking*, 3 (2/3), pp. 122–152, 2005. https://doi.org/10.1504/ijhpcn.2005.008032
12. J. Chuzhoy, S. Guha, S. Khanna, J. Naor. “Machine Minimization for Scheduling Jobs with Interval Constraints”. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, pp. 81–90, 2004. https://doi.org/10.1109/FOCS.2004.38
13. M. Cieliebak, T. Erlebach, F. Hennecke, B. Weber, P. Widmayer. “Scheduling With Release Times and Deadlines on A Minimum Number of Machines”, in *Exploring New Frontiers of Theoretical Informatics*, J. J. Levy, E. W. Mayr, J. C. Mitchell (eds.), Kluwer Academic Publishers, Boston, pp. 209–222, 2004. https://doi.org/10.1007/1-4020-8141-3_18

14. S. Yaw, B. Mumey. “Scheduling Non-Preemptible Jobs to Minimize Peak Demand”, *Algorithms*, 10 (4), article 122, 2017. https://doi.org/10.3390/a10040122
15. L. Chen, N. Megow, K. Schewior. “An O(log m)-Competitive Algorithm for Online Machine Minimization”, *SIAM Journal on Computing*, 47 (6), pp. 2057–2077, 2018. https://doi.org/10.1137/17M116032X
16. S. Im, B. Moseley, K. Pruhs, C. Stein. “An O(Log Log m)-Competitive Algorithm for Online Machine Minimization”. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pp. 343–350, 2017. https://doi.org/10.1109/RTSS.2017.00039
17. A. W. J. Kolen, J. K. Lenstra, C. H. Papadimitriou, F. C. R. Spieksma. “Interval scheduling: A survey”, *Naval Research Logistics*, 54 (5), pp. 530–543, 2007. https://doi.org/10.1002/nav.20231
18. R. H. Möhring. “Minimizing Costs of Resource Requirements in Project Networks Subject to a Fixed Completion Time”, *Operations Research*, 32 (1), pp. 89–120, 1984. https://doi.org/10.1287/opre.32.1.89
19. A. Drexl, A. Kimms. “Optimization guided lower and upper bounds for the resource investment problem”, *Journal of the Operational Research Society*, 52 (3), pp. 340–351, 2001. https://doi.org/10.1057/palgrave.jors.2601099
20. W. A. Horn. “Some simple scheduling algorithms”, *Naval Research Logistics Quarterly*, 21 (1), pp. 177–185, 1974. https://doi.org/10.1002/nav.3800210113
21. A. Federgruen, H. Groenevelt. “Preemptive Scheduling of Uniform Machines by Ordinary Network Flow Techniques”, *Management Science*, 32 (3), pp. 341–349, 1986. https://doi.org/10.1287/mnsc.32.3.341
22. C. Martel. “Preemptive Scheduling with Release Times, Deadlines, and Due Times”, *Journal of the ACM*, 29 (3), pp. 812–829, 1982. https://doi.org/10.1145/322326.322337
23. M. Haouari, A. Gharbi. “An improved max-flow-based lower bound for minimizing maximum lateness on identical parallel machines”, *Operations Research Letters*, 31 (1), pp. 49–52, 2003. https://doi.org/10.1016/s0167-6377(02)00177-3
24. P. Baptiste, C. Le Pape, W. Nuijten. “Satisfiability tests and time-bound adjustments for cumulative scheduling problems”, *Annals of Operations Research*, 92, pp. 305–333, 1999. https://doi.org/10.1023/A:1018995000688
25. L. Hidri, A. Gharbi, M. Haouari. “Energetic reasoning revisited: application to parallel machine scheduling”, *Journal of Scheduling*, 11 (4), pp. 239–252, 2008. https://doi.org/10.1007/s10951-008-0070-z
26. Y. Ouellet, C. G. Quimper. “A O(n log² n) Checker and O(n² log n) Filtering Algorithm for the Energetic Reasoning”, in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, W. J. van Hoeve (eds.), Springer, Cham, pp. 477–494, 2018. https://doi.org/10.1007/978-3-319-93031-2_34
27. D. G. Feitelson, D. Tsafrir, D. Krakov. “Experience with using the Parallel Workloads Archive”, *Journal of Parallel and Distributed Computing*, 74 (10), pp. 2967–2982, 2014. https://doi.org/10.1016/j.jpdc.2014.06.013
28. D. G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*, Cambridge University Press, 2015. https://doi.org/10.1017/CBO9781139939690
29. E. Frachtenberg, D. G. Feitelson. “Pitfalls in Parallel Job Scheduling Evaluation”, in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, E. Frachtenberg, L. Rudolph, U. Schwiegelshohn (eds.), Springer, Berlin, Heidelberg, pp. 257–282, 2005. https://doi.org/10.1007/11605300_13
30. M. R. Garey, D. S. Johnson. “‘Strong’ NP-Completeness Results: Motivation, Examples, and Implications”, *Journal of the ACM*, 25 (3), pp. 499–508, 1978. https://doi.org/10.1145/322077.322090
31. F. Brandão, J. P. Pedroso. “Bin Packing and Related Problems: General Arc-flow Formulation with Graph Compression”, *Computers & Operations Research*, 69, pp. 56–67, 2016. https://doi.org/10.1016/j.cor.2015.11.009
32. A. W. Mu’alem, D. G. Feitelson. “Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling”, *IEEE Transactions on Parallel and Distributed Systems*, 12 (6), pp. 529–543, 2001. https://doi.org/10.1109/71.932708
33. Parallel Workloads Archive, “The MetaCentrum 2 log”, 2015. Available at: http://www.cs.huji.ac.il/labs/parallel/workload/l_metacentrum2/index.html
34. D. Klusáček, Š. Tóth, G. Podolníková. “Real-Life Experience with Major Reconfiguration of Job Scheduling System”, in *Job Scheduling Strategies for Parallel Processing*, N. Desai, W. Cirne (eds.), Springer, Cham, pp. 83–101, 2017. https://doi.org/10.1007/978-3-319-61756-5_5
35. T. Hoefler, R. Belli. “Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results”. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2015. https://doi.org/10.1145/2807591.2807644