SPT is Optimally Competitive for Uniprocessor Flow

David P. Bunde*

Abstract

We show that the Shortest Processing Time (SPT) algorithm is $(\Delta + 1)/2$-competitive for nonpreemptive uniprocessor total flow time with release dates, where Δ is the ratio between the longest and shortest job lengths. This is best possible for a deterministic algorithm and improves on the $(\Delta + 1)$ competitive ratio shown by Epstein and van Stee using different methods.

Keywords: Algorithms; On-line algorithms; Scheduling

1 Introduction

We consider the problem of online nonpreemptive scheduling with release dates on a single machine to minimize total flow time $(1|r_i|\sum_i F_i)$. The input is a sequence of n jobs, where job J_i cannot be started before its release time r_i and must exclusively occupy the machine for its processing time p_i. In our model, the value of p_i is known at time r_i. Let S_i^A and C_i^A be the starting time and completion time of J_i when scheduled by algorithm A. The flow time of job J_i is $F_i^A = C_i^A - r_i$, the time between its release and completion. Our objective is to minimize $\sum_{i=1}^n F_i^A$, the total flow time. When preemption is allowed, i.e., jobs can be paused and resumed without penalty, the problem is solved optimally by the algorithm Shortest Remaining Processing Time (SRPT) [6, 7], which always runs the job with the least remaining processing time. When preemption is not allowed, every deterministic algorithm is $\Omega(n)$-competitive for total flow [4]. Even in the offline setting, flow cannot be approximated within a factor of $\Omega(n^{1/2-\epsilon})$, for any $\epsilon > 0$, unless P=NP [4].

These strong bounds make it natural to consider approximations in terms of other parameters. One choice is Δ, the ratio between the largest and smallest processing times. Epstein and van Stee [3] give bounds in terms of Δ for a resource-augmented version of the problem where an online algorithm running on l processors is compared to the optimal algorithm (SRPT) running on one processor. For the special case $l = 1$, they show that the algorithm Shortest Processing Time (SPT), which begins the shortest available job whenever the processor becomes idle, is $(\Delta+1)$-competitive. They also show an $\Omega(\Delta)$ lower bound for deterministic algorithms.

The main result of this paper is the following:

Theorem 1 *SPT is $(\Delta + 1)/2$-competitive for total flow.*

*Dept. of Computer Science, University of Illinois, Urbana-Champaign, (bunde@uiuc.edu). Partially supported by NSF grant CCR-0093348 and Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

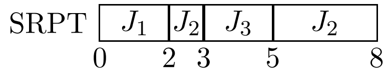


Figure 1: SRPT schedule of 3 jobs. The active intervals of jobs J_1, J_2, and J_3 are $[0, 2)$, $[2, 8)$, and $[3, 5)$ respectively. Intervals $[0, 2)$ and $[2, 8)$ are maximal, so the blocks are $\{J_1\}$ and $\{J_2, J_3\}$.

In Section 2, we define blocks of jobs based on the SRPT schedule and show that the jobs of a block are also executed together in the SPT schedule. In Section 3 we define a block-based schedule ECHO and show it has competitive ratio $(\Delta + 1)/2$. In Section 4 we use ECHO as an intermediate between SRPT and SPT to prove Theorem 1. Then, in Section 5, we show a lower bound of $(\Delta + 1)/2$ on the competitive ratio of any deterministic algorithm. Section 6 gives concluding remarks and open problems.

2 Block Structure

Although typical definitions of SRPT and SPT do not specify how the algorithms choose among jobs of the same processing time, it is necessary to do so to prove our results. If SRPT has already worked on one of the jobs with equal remaining processing time, we require that it resume this job before starting the others. It may choose between jobs with equal initial processing time arbitrarily, provided that SPT uses the same order.

Now we can define blocks. The *active interval* of job J_i is the half-open interval $[S_i^{SRPT}, C_i^{SRPT})$. When two active intervals intersect, one contains the other [4]. We focus on *maximal* active intervals, those not contained in any other. A *block* is the set of jobs run during a maximal active interval. Since maximal active intervals are disjoint, the blocks partition the set of jobs. Figure 1 illustrates these definitions.

The main result of this section is that SPT obeys the block structure of SRPT; the only difference is the order in which it runs the jobs of each block. To show this, we label the blocks $B_1, B_2, \ldots, B_m$ in the order SRPT runs them and use I_i to denote the interval when SRPT runs the jobs of B_i. Let the *SRPT-rank* of job J be the index of the block containing it, i.e. J has SRPT-rank i if $J \in B_i$.

Theorem 2 *SPT runs jobs in order of non-decreasing SRPT-rank.*

We say that an algorithm is *busy* if it is idle only when it has completed all jobs that have been released. Because SRPT and SPT are both busy algorithms, they are idle at exactly the same times and Theorem 2 is equivalent to the following:

Corollary 3 *For each i, SPT runs exactly the jobs of B_i during interval I_i.*

Proof of Theorem 2: Consider a counterexample with fewest jobs. In such a counterexample, SPT must begin a job of SRPT-rank 2 before finishing all jobs of SRPT-rank 1, because otherwise both SPT and SRPT would finish the first block of jobs at the same time and these jobs could be removed to create a smaller counterexample.

Because we specified that they use the same tie-breaking rule to select a job, SPT and SRPT both begin with the same job J_a. SRPT must preempt J_a, because otherwise this is the only job of SRPT-rank 1. Suppose SPT first starts a job of SRPT-rank 2 at time t and let J_b be the job it starts. Because we have a smallest counterexample, all jobs are started by at least one of the algorithms by time t. In particular, this implies that no jobs arrive after time t.

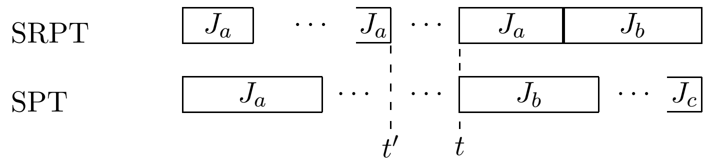


Figure 2: Illustration for the proof of Theorem 2

Let $p_a(t)$ be the remaining processing time of J_a at time t, and let time $t' \le t$ be the latest time before time t that SRPT works on J_a. Since SRPT works on J_a immediately before time t', it had finished all jobs shorter than $p_a(t)$ available before time t'. Because the jobs SRPT works on between times t' and t must be shorter than $p_a(t)$, they arrive no earlier than time t'. They are also shorter than p_b since $J_b \notin B_1$ implies $p_b \ge p_a(t)$. Thus, SPT schedules them before J_b. Since SPT starts J_b at time t, these jobs finish in time $t - t'$, so SRPT also finishes them at time t and resumes J_a. Because no jobs arrive after time t, J_a is not interrupted. The block ends when SRPT finishes J_a so SRPT runs J_b next. Figure 2 depicts the situation.

SPT and SRPT are both busy so they finish the instance at the same time. SPT must run at least one job after J_b since $p_a(t) > 0$. Let J_c be the last job it runs, so J_b and J_c finish simultaneously. Because $p_b \ge p_a(t)$, SRPT finishes J_a no later than SPT finishes J_b and thus starts J_b no later than SPT starts J_c. Hence, $p_b \ge p_c$.

Recall, however, that SPT runs J_b before J_c. J_c was available when SPT starts J_b at time t because no jobs arrive after time t, implying $p_b \le p_c$. Since SRPT runs J_c before J_b, consistent tie breaking strengthens this to $p_b < p_c$, a contradiction. $\Box$

3 Schedule ECHO

Now we define a new schedule ECHO in terms of the SRPT schedule. ECHO is idle at exactly the same times as SRPT. During I_j, ECHO runs the jobs of B_j, starting with the same job as SRPT and then running the others in order of increasing SRPT completion time. The jobs are run without delay so they complete during I_j. Figures 3 and 4 give examples of ECHO schedules.

For the SRPT schedule at time t, let $\mathit{first}(t)$ be the work remaining on the block's first job, $\mathit{part}(t)$ be the work done on partially-completed jobs other than the first, and $\mathit{curr}(t)$ be the work remaining on the currently-running job. These quantities obey the following relationship:

Lemma 4 *$\mathit{curr}(t) \le \mathit{first}(t) - \mathit{part}(t)$, with strict inequality unless SRPT is idle or working on the block's first job at time t.*

Proof: We prove this for each block. At the start of a block, the claim holds because $\mathit{curr}(t) = \mathit{first}(t)$ and $\mathit{part}(t) = 0$. When SRPT does not switch jobs, the inequality remains true because $\mathit{curr}(t)$ decreases to compensate for changes in $\mathit{first}(t)$ or $\mathit{part}(t)$. Now suppose that SRPT switches jobs at time t. Let J_i be the job SRPT was running immediately before time t.

> *Case 1: J_i is preempted to run a job J_j.* Let $\mathit{curr}'(t)$ be the value of $\mathit{curr}(t)$ if J_i had not been preempted. Because J_j preempts J_i, $p_j < \mathit{curr}'(t)$. If J_i is the first job, $\mathit{part}(t) = 0$ and $\mathit{curr}'(t) = \mathit{first}(t)$ imply the inequality. Otherwise, $\mathit{curr}(t) = p_j < \mathit{curr}'(t)$ preserves it.

Case 2: J_i is finished and the block ends. If a new block is begun, $\mathit{curr}(t) = \mathit{first}(t)$ and $\mathit{part}(t) = 0$. Otherwise, the processor is idle at time t, with $\mathit{first}(t) = \mathit{part}(t) = \mathit{curr}(t) = 0$.

Case 3: J_i is finished, but the block does not end. Let J_k be the unfinished job that was most recently preempted and time t^* be when its most recent preemption occurred. By definition, SRPT finishes jobs it runs between times t^* and t, so $\mathit{part}(t) = \mathit{part}(t^*)$. Also, $\mathit{first}(t) = \mathit{first}(t^*)$ since SRPT does not run the block's first job until J_k is finished. Because the inequality would hold at time t^* if J_k had not been preempted, $p_k(t^*) \leq \mathit{first}(t^*) - \mathit{part}(t^*)$. If J_k is resumed at time t, $\mathit{curr}(t) = p_k(t^*)$. Otherwise, a job shorter than J_j runs so $\mathit{curr}(t) < p_k(t^*)$. In either case, the inequality holds.

When SRPT is working on a job other than the block's first, the first job has been preempted and not resumed. Applying case 1 when the first job is preempted causes the inequality to become strict until the first job is resumed. □

Now we are ready to prove the soundness of ECHO.

Theorem 5 *ECHO does not run a job before its release time.*

Proof: This is clear for the first job in each block. For other jobs, we show that ECHO only starts jobs SRPT has already finished. To see this, view the ECHO schedule as being constructed incrementally, starting with the first job in the block and adding other jobs as SRPT completes them. At any time t after ECHO finishes the first job in the block, SRPT has spent $\mathit{first}(t) - \mathit{part}(t)$ time on jobs that it has completed, but ECHO has not. Thus, ECHO will take time $\mathit{first}(t) - \mathit{part}(t)$ to finish the already-scheduled jobs and Lemma 4 implies that SRPT finishes a job before ECHO runs out of already-scheduled jobs. □

Now we consider the competitiveness of ECHO.

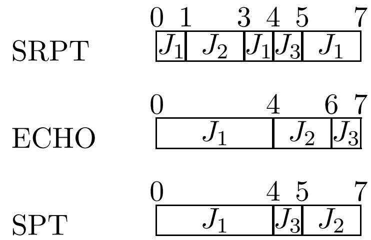
Lemma 6 *ECHO is $(\Delta + 1)/2$-competitive for total flow.*

Proof: We consider a single block starting with J_i. Let x be the sum of sizes of jobs other than J_i. Let Σ^{SRPT} and Σ^{ECHO} denote the total flow of SRPT and ECHO, respectively. In the SRPT schedule, J_i has flow $p_i + x$ and the other jobs have at least x, for at least $p_i + 2x$ total. ECHO delays J_i by x less than SRPT and delays each of the other jobs by at most p_i more. Since there are at most x/p_{min} other jobs, where p_{min} is the instance's minimum processing time, $\Sigma^{ECHO} \leq \Sigma^{SRPT} - x + p_i x/p_{min} \leq \Sigma^{SRPT} + (\Delta - 1)x$. Thus, the competitive ratio is $\Sigma^{ECHO}/\Sigma^{SRPT} \leq 1 + (\Delta - 1)x/\Sigma^{SRPT} \leq 1 + (\Delta - 1)x/(p_i + 2x) \leq (\Delta + 1)/2$ □

4 Proof of Theorem 1

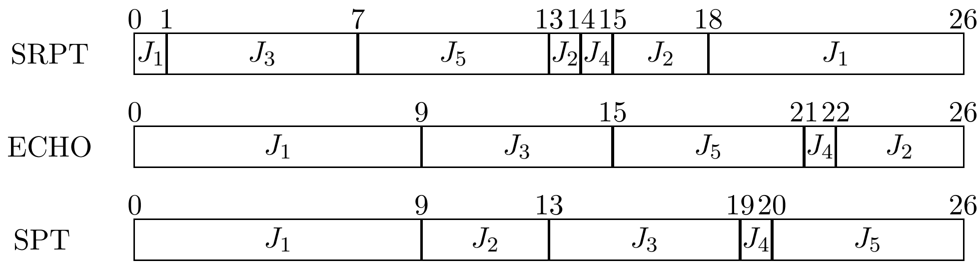
Now we prove Theorem 1 by showing that SPT generates a schedule no worse than ECHO for every problem instance. (Figure 3 gives an instance where SPT is strictly better.) Lemma 6 then implies that SPT is $(\Delta + 1)/2$-competitive.

To compare SPT and ECHO, consider changing a SPT schedule into an ECHO schedule by repeatedly removing the first difference between them. By Corollary 3, it suffices to consider the schedule of a single block. Without loss of generality, assume SPT runs jobs in numerical order: J_1 followed by J_2 and so on. Let the schedules first differ at time τ, when ECHO starts J_i and SPT starts $J_{i'}$ with $i > i'$. We remove this difference with a *slide*; start J_i at time τ and delay $J_{i'}, \ldots, J_{i-1}$



i	r_i	p_i
1	0	4
2	1	2
3	4	1

Figure 3: Instance with SPT better than ECHO ($\sum F_i^{SPT} = 11$ and $\sum F_i^{ECHO} = 12$)



i	r_i	p_i
1	0	9
2	9	4
3	1	6
4	14	1
5	7	6

Figure 4: SRPT, ECHO, and SPT schedules

by p_i. This increases flow by $(i - i')p_i - \sum_{k=i'}^{i-1} p_k = \sum_{k=i'}^{i-1}(p_i - p_k)$. The latter expression gives the increase as the sum of increases due to each *inversion*, a change in relative order of J_i and J_k. We denote an inversion moving J_i before J_j with (J_i, J_j). For our proof, we call inversion (J_i, J_j) *bad* if $p_i < p_j$ and *good* otherwise; bad inversions are those that decrease flow.

For the instance in Figure 4, changing the SPT schedule into the ECHO schedule requires 3 slide operations, advancing J_3, J_5, and J_4. These slides cause the inversions (J_3, J_2), (J_5, J_4), (J_5, J_2), and (J_4, J_2). The only bad inversion is (J_4, J_2), which changes flow by $p_4 - p_2 = -3$.

We now show that the slide operations increase flow by proving that procedure PAIR given in Figure 5 pairs each bad inversion in which job J_{slide} moves earlier with a good inversion in which J_{slide} moves later so that each pair has a net increase. To demonstrate this procedure, we use it to find a partner for (J_4, J_2) in the instance shown in Figure 4. $J_{slide} = J_4$ and only J_2 is colored blue on line 2. In the first iteration of the loop (lines 4–14), $J_{blue} = J_2$, $t = 9$, $X = \{J_1\}$, and $Y = \{J_3\}$. Job J_3 is alone in $Y \setminus X$ so $J_{pick} = J_3$. SPT runs J_4 after J_3 so we color J_3 blue and set its note to J_2. In the second iteration, $J_{blue} = J_3$, $t = 13$, $X = \{J_1, J_2\}$, and $Y = \{J_3, J_5\}$. The only red job in $Y \setminus X$ is J_5. SPT runs J_4 before J_5 so inversion (J_5, J_4) is paired with $(J_4, note(J_3) = J_2)$. This pair has net change in flow $(p_5 - p_4) + (p_4 - p_2) = 2$.

We begin showing that PAIR works by proving a pair of technical lemmas.

Lemma 7 *At each step,* $p_{blue} \leq p_{pick}$.

Proof: SPT runs J_{blue} rather than J_{pick} at time t; $J_{pick} \notin X$ so SPT has not run J_{pick}, but $J_{pick} \in Y$ is available because SRPT has finished it. □

Lemma 8 J_{slide} *is not released until SPT has started all the blue jobs.*

1 Procedure PAIR(job J_{slide}, schedule SPT, schedule SRPT)
2 color job J blue if bad inversion (J_{slide}, J) occurs and red otherwise
3 attach a note "J" to each job J
4 foreach blue job J_{blue} in SPT (in order of increasing SPT start time)
5 $t =$ start time of J_{blue} in SPT
6 $X =$ jobs SPT finished by time t
7 $Y =$ jobs SRPT finished by time t
8 $J_{pick} =$ any red job of $Y \setminus X$
9 if(SPT runs J_{slide} before J_{pick})
10 pair (J_{pick}, J_{slide}) with $(J_{slide}, note(J_{blue}))$
11 color job J_{pick} green
12 else
13 color J_{pick} blue
14 copy the note of J_{blue} to J_{pick}

Figure 5: Procedure PAIR, which finds partners for bad inversions involving J_{slide}

Proof: The lemma follows from the following invariants: (1) J_{slide} is shorter than each of the blue jobs, and (2) SPT runs J_{slide} after all the blue jobs. Both hold initially because each blue job occurs in a bad inversion with J_{slide}. If J_{pick} is colored blue on line 13, the first invariant holds by Lemma 7 because the newly blue J_{pick} is longer than one of the jobs that was already blue. The second invariant holds because we only color J_{pick} blue if SPT runs J_{slide} after it. □

Now we show J_{pick} can always be selected on line 8.

Lemma 9 PAIR *can always find a red job in* $Y \setminus X$.

Proof: First we show there is a blue job in X for each blue or green job in Y. Consider a job $J \in Y$ that is not red. It cannot have been colored blue on line 2 because this implies that SRPT finishes it after J_{slide} and J_{slide} has not been released yet by Lemma 8. Since J is not red and was not colored blue on line 2, it was J_{pick} in a previous loop iteration. The J_{blue} from that iteration is in X since line 4 considers jobs in order of SPT start time.

Now observe that $J_1 \in X$. Since the block ends when SRPT finishes J_1, $J_1 \notin Y$. Also, J_1 is never colored blue so it did not cause a J_{pick} to be selected. The lemma follows since SRPT has always finished at least as many jobs as any other algorithm [6]. □

Next we show that the resulting pairs are valid and have non-negative flow.

Lemma 10 *Each pair consists of 2 valid inversions whose combined change to flow is non-negative.*

Proof: First we show that inversion (J_{pick}, J_{slide}) exists. (The other inversion exists since $note(J_{blue})$ was colored blue on line 2.) A pair is only made if SPT runs J_{slide} before J_{pick}. SRPT finishes $J_{pick} \in Y$ by time t. Since J_{slide} is released after time t by Lemma 8, SRPT and ECHO run J_{slide} after J_{pick}.

Now we show that the pair gives non-negative change in flow. Denote the processing time of the job named in $note(J_i)$ with $p_{note(i)}$. Then the net change of $\{(J_{pick}, J_{slide}), (J_{slide}, note(J_{blue}))\}$ is $p_{pick} - p_{slide} + p_{slide} - p_{note(blue)} = p_{pick} - p_{note(blue)}$. By Lemma 7, this is at least $p_{blue} - p_{note(blue)}$.

Now it suffices to show $p_i \geq p_{note(i)}$ for all i. This is initially true because $note(J_i) = J_i$. Notes are only changed on line 14, where a note is copied from J_{blue} to J_{pick}. Since $p_{blue} \leq p_{pick}$ by Lemma 7, the claim is maintained. □

Finally, we address termination.

Lemma 11 PAIR *terminates and pairs all bad inversions caused by sliding* J_{slide}.

Proof: PAIR terminates because each job is J_{blue} at most once. Furthermore, since $J_{pick} \notin X$ by construction, if J_{pick} is colored blue on line 13, SPT runs it after J_{blue}. Thus, each blue job creates either a pair or a later blue job. Since all blue jobs are visited, all inversions are paired. □

5 Lower Bound

Now we show that ECHO and SPT have the best possible competitive ratio.

Theorem 12 *No deterministic algorithm for nonpreemptive uniprocessor total flow is c-competitive for any fixed* $c < (\Delta + 1)/2$.

Proof: For any algorithm, we construct an adversarial instance on which the algorithm's competitive ratio is arbitrarily close to $(\Delta + 1)/2$. The instance's first job has processing time Δ and release time 0. Any deterministic algorithm delays for some constant time C, dependent on the algorithm, and then starts this job. The instance's remaining jobs all have processing time 1 and release time $C + \epsilon + i$ for small ϵ and each $i = 0, \ldots, n-2$. The algorithm has flow at least $C + \Delta + (\Delta + 1 - \epsilon)(n-1)$. The optimal algorithm runs the small jobs as they arrive and runs the first job either immediately if $C \geq \Delta$ or after the small jobs if $C < \Delta$. The latter case yields greater flow, $C + \Delta + 2(n-1)$. The ratio between the algorithm's flow and the optimal flow approaches $(\Delta + 1)/2$ as $n \to \infty$ and $\epsilon \to 0$. □

6 Concluding Remarks

We have shown that SPT has the best possible competitive ratio among deterministic algorithms, but it is open whether randomized algorithms can do better. The best lower bound known for randomized uniprocessor flow is $\Omega(\sqrt{\Delta})$ [2].

On a multiprocessor, SRPT does not have the same block structure because preempted jobs can be restarted on different processors. However, Awerbuch et al. [1] give an $O(\log \min\{n, \Delta\})$-competitive multiprocessor algorithm that uses preemption, but not migration. In their algorithm, each processor runs SRPT on a subset of the jobs. Replacing the SRPT schedule on each processor with ECHO removes the preemptions while increasing the competitive ratio by an $O(\Delta)$ factor. No algorithm was known to be competitive for online nonpreemptive multiprocessor flow; the best offline approximation known is $O\left(\sqrt{n/m}\log(n/m)\right)$ on m processors [5].

Acknowledgements The author would like to acknowledge Cynthia Phillips, Vitus Leung, Jeff Erickson, and the anonymous referees for suggestions that greatly improved this paper.

References

[1] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing flow time without migration. In *Proc. 31st Symp. on Theory of Computation*, pages 198–205, 1999.

[2] L. Epstein and R. van Stee. Lower bounds for on-line single-machine scheduling. In *Proc. 26th Symp. Math. Found. Comput. Sci.*, number 2136 in LNCS, pages 338–350. Springer-Verlag, 2001. http://www.informatik.uni-freiburg.de/~vanstee/papers/weight.ps.

[3] L. Epstein and R. van Stee. Optimal on-line flow time with resource augmentation. In *Proc. 13th Symp. Fund. Comp. Theory*, number 2138 in LNCS, pages 472–482. Springer-Verlag, 2001. http://www.math.tau.ac.il/~lea/flow.ps.gz.

[4] H. Kellerer, T. Tautenhahn, and G. J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. *SIAM J. Computing*, 28(4):1155–1166, 1999.

[5] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proc. 29th Symp. on Theory of Computation*, pages 110–119, 1997.

[6] L. Schrage. A proof of the optimality of the shortest processing remaining time discipline. *Operations Research*, 16:687–690, 1968.

[7] D. Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26(1):197–199, 1976.