Variable Annealing Length and Parallelism in Simulated Annealing

Vincent A. Cicirello
Computer Science and Information Systems
School of Business, Stockton University
101 Vera King Farris Drive, Galloway, NJ 08205
http://www.cicirello.org/

Abstract

In this paper, we propose: (a) a restart schedule for an adaptive simulated annealer, and (b) parallel simulated annealing, with an adaptive and parameter-free annealing schedule. The foundation of our approach is the Modified Lam annealing schedule, which adaptively controls the temperature parameter to track a theoretically ideal rate of acceptance of neighboring states. A sequential implementation of Modified Lam simulated annealing is almost parameter-free. However, it requires prior knowledge of the annealing length. We eliminate this parameter using restarts, with an exponentially increasing schedule of annealing lengths. We then extend this restart schedule to parallel implementation, executing several Modified Lam simulated annealers in parallel, with varying initial annealing lengths, and our proposed parallel annealing length schedule. To validate our approach, we conduct experiments on an NP-Hard scheduling problem with sequence-dependent setup constraints. We compare our approach to fixed length restarts, both sequentially and in parallel. Our results show that our approach can achieve substantial performance gains, throughout the course of the run, demonstrating our approach to be an effective anytime algorithm.

1 Introduction

For many scheduling and optimization problems, metaheuristics such as simulated annealing (SA), genetic algorithms (GA), ant colony optimization (ACO), tabu search, etc, offer a means of trading off guarantees of optimality in favor of efficiently finding high quality solutions. Such algorithms often exhibit anytime behavior, providing increasingly better solutions with increases in available time.

Metaheuristic behavior is usually controlled by several parameters. GAs have mutation and crossover rates, among other parameters. SA has parameters that control the annealing schedule. ACO has parameters that balance the relative influence of heuristic guidance and the learned pheromone trails. Control parameters can either be tuned beforehand by some process (automated or otherwise) or adapted dynamically using search feedback. For example, there exist parameter control approaches for GA and other forms of evolutionary computation (Eiben, Hinterding, and Michalewicz 1999; Wessing, Preuss, and Rudolph 2011; Aleti and Moser 2013), adaptive annealing schedules for SA (Lam and Delosme 1988; Swartz 1993; Boyan 1998), among others.

Parallel implementation of many metaheuristics is straightforward, such as population-based algorithms like GA and ACO, whose motivations come from naturally occurring distributed behavior. While for others, parallel implementation is less obvious. In this paper, we propose a new approach to parallel SA. We execute several independent runs, with restarts, in parallel of an adaptive SA using the Modified Lam (Swartz 1993; Boyan 1998) annealing schedule. The Modified Lam annealing schedule is nearly parameter-free, requiring only knowledge of the annealing length. In our proposed parallel SA, we eliminate this last parameter with restarts following a schedule of annealing lengths that better balance the risk associated with errors in estimating available computation time. The initial and restart annealing lengths increase exponentially. The result is a parallel parameter-free SA with improved anytime behavior.

We validate our approach using an NP-Hard scheduling problem with sequence-dependent setups. We begin our experiments with the sequential case. Although for any a priori known fixed time limit, a single run of that length outperforms our restart schedule at run end, our proposed restart schedule exhibits greatly improved anytime behavior during the run. We continue our experiments in parallel, demonstrating our parallel variable-length SA to significantly dominate parallel fixed-length restarts early in the run.

The paper is organized as follows. In Section 2, we discuss related work on parallel SA. We provide details of our parallel parameter-free SA in Section 3. Then, in Section 4, we validate our approach experimentally with an NP-Hard scheduling problem, consisting of sequence-dependent setup constraints, and offer conclusions in Section 5.

2 Background

SA is typically described in a very sequential manner, and not parallelized as obviously as other metaheuristics. There are two major categories of parallel SA (Rudolph 1993): namely, approaches that implement neighbor evaluations in parallel, and approaches that are essentially parallel implementations of multistart SA. An example of the first case is the work of Ludwin and Betz who use a parallel SA for FPGA placement to optimize critical path length (Ludwin and Betz 2011). They execute multiple move evaluations in

parallel using what they call speculative moves.

The second type can be referred to as parallel multistart. Although restarting some algorithms, such as hill climbers, offers a means of countering large numbers of local optima; many have shown sequential multistart SA to be ineffective. One long run of SA the length of the available time is typically more effective than taking the best solution from a set of shorter runs. Therefore, it is not surprising that approaches to parallel SA that execute multiple runs of SA in parallel rarely involve independent runs. More commonly, parallel multistart SA involves sharing information among the parallel runs. For example, Ram et al's approach (for job shop scheduling) periodically exchanges the best solution among the parallel runs, each continuing its search from there (Ram, Sreenivas, and Subramaniam 1996). More recently, in Jha and Menon's approach, at intervals called "beats" the best solution is shared among threads (Jha and Menon 2014). Jha and Menon developed their approach for general purpose computation on graphics processing units (GPGPU) for a sports league scheduling problem.

Others apply SA in parallel to optimize a set of sub-problems, each SA instance solving a different sub-problem. For example, Rahimian et al's approach to graph partitioning, specifically for large social network graphs, distributes the problem, and individual distributed instances of SA optimize portions of the problem (Rahimian et al. 2015). They apply this to both edge-cut and vertex-cut partitioning.

Other forms of search often rely on restarts, quite effectively. For example, in constraint satisfaction, satisfiability, and other similar problems, backtracking search using randomized variable-ordering and value-ordering heuristics often exhibit heavy-tailed runtime distributions (Gomes et al. 2000). Using an effective restart strategy, one can try to abandon the runs that are likely in the heavy-tail, restarting in an attempt to more directly solve the problem. The Luby restart schedule is the most widely known (Luby, Sinclair, and Zuckerman 1993), and has been parallelized (Cire, Kadioglu, and Sellmann 2014). The first several restart lengths of the Luby schedule (in number of backtracks) are as follows: $[1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots]$. You begin with a restart sequence, $[1, 1, 2]$, then repeat the entire prior sequence from the beginning followed by a restart double the length of the last, etc.

3 Technical Approach

3.1 Modified Lam Simulated Annealing

The foundation of our approach is an existing sequential simulated annealer with an adaptive annealing schedule. SA operates via a mechanism modeled after the process of heating a metal and allowing it to cool slowly. Heating enables the material to be shaped as desired, while cooling at a slow rate minimizes internal stress thus enabling greater stability in the final state.

In SA, search is controlled by a temperature parameter. The most basic form of SA begins with a high temperature and then "cools" at some rate, with both initial temperature and cooling rate as system parameters. The Modified Lam annealing schedule (Swartz 1993; Boyan 1998) eliminates these parameters, by dynamically adjusting temperature using search feedback. It's based on results of Lam and Delosme (Lam and Delosme 1988), where they showed that the ideal run of SA accepts neighboring states at a rate of 0.44, which formed the basis for an annealing schedule that tracks this acceptance rate. Lam and Delosme's version originally used a monotonically decreasing temperature schedule, and adjusted the size of the neighborhood to maintain the acceptance rate as near 0.44 as possible—e.g., they increased the size of the local neighborhood to decrease the acceptance rate, and decreased the size of the local neighborhood to increase the acceptance rate. They relied on the common assumption that nearby search states are of similar quality; and thus, a smaller local neighborhood implies smaller difference between current fitness and neighbor fitness, which leads to higher probability of neighbor acceptance.

Modified Lam Annealing

$S \leftarrow$ GenerateRandomInitialState
$T \leftarrow 0.5$
AcceptRate $\leftarrow 0.5$
for $i = 1$ **to** MaxEvals **do**
  $S' \leftarrow$ random selection from $\eta(S)$
  if Cost(S') $\leq$ Cost(S) **or** Rand $\in [0, 1) <$ $e^{(\text{Cost}(S)-\text{Cost}(S'))/T}$ **then**
    $S \leftarrow S'$
    AcceptRate $\leftarrow \frac{1}{500}(499 \cdot \text{AcceptRate} + 1)$
  else AcceptRate $\leftarrow \frac{1}{500}(499 \cdot \text{AcceptRate})$
  if $i/\text{MaxEvals} < 0.15$ **then**
    LamRate $\leftarrow 0.44 + 0.56 \cdot 560^{-i/\text{MaxEvals}/0.15}$
  else if $0.15 \leq i/\text{MaxEvals} < 0.65$ **then**
    LamRate $\leftarrow 0.44$
  else if $0.65 \leq i/\text{MaxEvals}$ **then**
    LamRate $\leftarrow 0.44 \cdot 440^{-(i/\text{MaxEvals}-0.65)/0.35}$
  if AcceptRate $>$ LamRate **then** $T \leftarrow 0.999 \cdot T$
  else $T \leftarrow T/0.999$
return Best solution found during run

Figure 1: SA with the Modified Lam annealing schedule.

Swartz later made additional observations on Lam and Delosme's annealing schedule (Swartz 1993), which were then refined by Boyan into the Modified Lam schedule (Boyan 1998). Specifically, Swartz observed that at the beginning of the search, the acceptance rate is near 1.0 (i.e., random search) and decreases at an exponential rate during the first 15% of the run when it reaches the target acceptance rate of 0.44, continues at that rate for 50% of the run, and then declines exponentially to the end of the run (i.e., end of run converges to a stochastic hill climb). Rather than adjusting the size of the local neighborhood, Swartz's and Boyan's Modified Lam schedule varies the temperature—increasing temperature to increase acceptance rate, and decreasing temperature to decrease acceptance rate. Figure 1 shows SA with the Modified Lam schedule. In the pseudocode, $\eta(S)$ refers to the set of neighboring states of S (i.e., our neighborhood function).

3.2 Parallel Variable Length Runs

The Modified Lam annealing schedule is nearly parameter-free. However, it requires the annealing length, referred to as MaxEvals in the pseudocode of Figure 1. It is not always practical to accurately predict the time available for search. If the run is shorter than you anticipate, the search would have spent too much time randomly exploring, and insufficient time exploiting observed high quality portions of the search space. If the run is much longer than you expected, it will get stuck too early in a local optimum, failing to effectively utilize the unexpected extra time.

For sequential SA, many have shown that one longer run of SA is typically much better than restarting shorter runs. Extending this to parallel SA with independent instances, one would expect better performance if all parallel instances were single runs of the available time. However, the available time may not be known, and may be difficult to accurately predict. Our approach attempts to balance the risk associated with incorrectly estimating time available, using restarts with a schedule of increasing annealing lengths.

Additionally, we define our restart schedule to support both sequential and parallel implementations. Specifically, we propose a parallel SA, that executes several SA instances with the Modified Lam annealing schedule. Each SA instance has a different initial value of MaxEvals. As each SA instance completes its initial run, it restarts with a new longer run. The SA instances operate independently, sharing no data, and each restart begins anew with randomly generated initial solutions. The best solution found among all parallel runs is returned. The approach is essentially a parallel implementation of a multistart SA, but where the length of the restarts varies and increases.

Variable Annealing Length (VAL): In the sequential case, our annealing length schedule, VAL, is as follows. Restart r ($r = 0$ is the initial run) is of length:

$$\text{MaxEvals}(r) = 1000 * 2^r. \tag{1}$$

Thus, the multistart SA follows a sequence of annealing lengths $\{1000, 2000, 4000, 8000, 16000, 32000, \ldots\}$.

Parallel Variable Annealing Length Version 0 (P-VAL-0): Assume that we execute N instances, $\{\text{SA}_0, \text{SA}_1, \ldots, \text{SA}_{N-1}\}$, of multistart Modified Lam SA in parallel. The length, $\text{MaxEvals}_i(r)$, for restart r of instance SA_i is:

$$\text{MaxEvals}_i(r) = 1000 * 2^{i+r*N}. \tag{2}$$

In the case of $N = 1$, there is a single instance SA_0 and thus P-VAL reduces to VAL.

Consider $N = 3$ as an example. SA_0 has a sequence of annealing lengths $\{1000, 8000, 64000, \ldots\}$, SA_1 has annealing lengths $\{2000, 16000, 128000, \ldots\}$, and SA_2 has annealing lengths $\{4000, 32000, 256000, \ldots\}$.

Parallel Variable Annealing Length (P-VAL): There are deficiencies in P-VAL-0 related to parallel speedup, for $N > 4$, which we discuss later in Section 4.5. We resolve those deficiencies with the following schedule of annealing lengths $\text{MaxEvals}_i(r)$, for restart r of instance SA_i:

$$\text{MaxEvals}_i(r) = 1000 * 2^{(i \bmod 4)+r*\min\{N,4\}}. \tag{3}$$

When $N \leq 4$, P-VAL is identical to P-VAL-0. When $N > 4$, $\{\text{SA}_0, \text{SA}_4, \text{SA}_8, \ldots\}$ all have a sequence of annealing lengths $\{1000, 16000, 256000, \ldots\}$, $\{\text{SA}_1, \text{SA}_5, \text{SA}_9, \ldots\}$ have annealing lengths $\{2000, 32000, 512000, \ldots\}$, $\{\text{SA}_2, \text{SA}_6, \text{SA}_{10}, \ldots\}$ have annealing lengths $\{4000, 64000, 1024000, \ldots\}$, and $\{\text{SA}_3, \text{SA}_7, \text{SA}_{11}, \ldots\}$ have annealing lengths $\{8000, 128000, 2048000, \ldots\}$.

4 Experiments

4.1 Scheduling with Sequence-Dependent Setups

To validate our approach, we consider an NP-Hard single machine scheduling problem, characterized by sequence-dependent setups, with an objective of minimizing weighted tardiness. The problem is NP-Hard even if setups are independent of job ordering (Morton and Pentico 1993), and the sequence-dependent setups magnify computational difficulty by inducing a non-order-preserving property of the evaluation function (Sen and Bagchi 1996).

The problem is defined as follows, and consists of N jobs, $J = \{j_1, j_2, \ldots, j_N\}$. Each job j_k has weight w_k, duedate d_k, and processing time p_k. Setup times $s_{i,k}$, required prior to processing job j_k if it immediately follows job j_i, depend on the job ordering, and are asymmetric (i.e., $s_{i,k} \neq s_{k,i}$); and $s_{0,k}$ is the initial setup time required if job j_k is processed first. The jobs J must be sequenced to minimize:

$$T = \sum_{k=1}^{N} w_k T_k = \sum_{k=1}^{N} w_k \max(c_k - d_k, 0), \tag{4}$$

where T_k is the tardiness of job j_k. The completion time c_k of j_k is the sum of the processing and setup times of j_k and of all jobs that preceed j_k. If $\pi(k)$ is the position of job j_k in the sequence, then define c_k as:

$$c_k = \sum_{\pi(x) \leq \pi(k), \pi(x) = \pi(y)+1} (p_x + s_{y,x}). \tag{5}$$

In our experiments, we use the standard benchmark set for the problem (Cicirello 2003a; 2003b), which consists of 120 instances, 40 each of loose, medium, and tight due dates. Of these, 22 loose duedate instances have an optimal weighted tardiness of 0.

The best available exact solver, Tanaka and Araki's Successive Sublimation Dynamic Programming, can solve all of the available benchmark instances, but requires over two weeks of memory-intensive CPU time to solve the hardest instances (Tanaka and Araki 2013). Therefore, metaheuristics are a more practical approach. A variety of algorithms have been proposed for the problem, such as dynamic programming (Tanaka and Araki 2013), neighborhood search (Liao, Tsou, and Huang 2012), iterated local search (Xu, Lü, and Cheng 2014), value-biased stochastic sampling (Cicirello and Smith 2005), ACO (Liao and Juan 2007), GA (Cicirello 2015; 2006), SA (Cicirello 2007), etc.

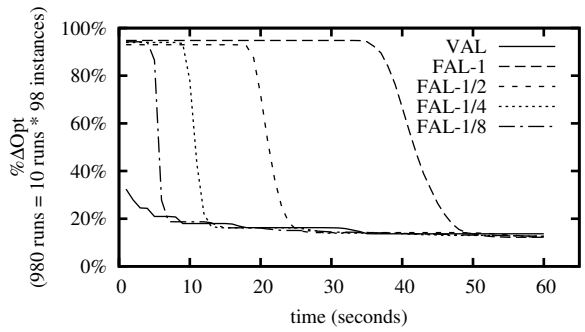

Figure 2: Sequential case: %ΔOpt.

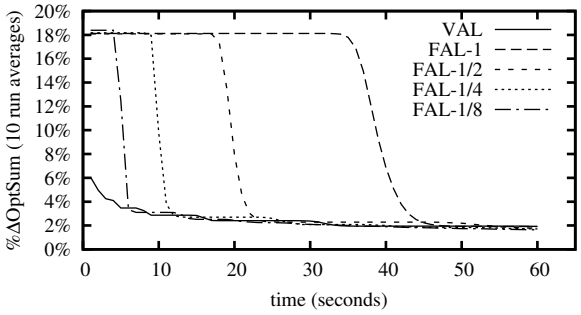

Figure 3: Sequential case: %ΔOptSum.

We preprocess the instances transforming them as suggested by others (Tanaka and Araki 2013; Cicirello 2015). To minimize the impact of setup times, we increase the processing time of each job, j_k, by its minimum setup time, and reduce all setup times accordingly (Cicirello 2015):

$$s_k^{\min} = \min_{0 \leq i \leq N, i \neq k} s_{i,k}, \tag{6}$$

$$p_k = p_k + s_k^{\min}, \tag{7}$$

$$s_{i,k} = s_{i,k} - s_k^{\min}, \forall i, i \neq k, 0 \leq i \leq N. \tag{8}$$

We also eliminate jobs j_k, such that $w_k = 0$, if $\forall x \forall y, x \neq y, s_{x,k} + p_k + s_{k,y} \geq s_{x,y}$ (Tanaka and Araki 2013).

4.2 Experimental Design

We conduct our experiments on an Ubuntu 14.04 Server, with 32GB memory and two Intel Xeon L5520 Quad-Core CPUs (2.27GHz). The L5520 supports hyper-threading with two threads per core, so our server has a total of 16 logical cores. We implement our experiments with Java 8 and the Java HotSpot 64-bit Server VM.

We conduct experiments in both the sequential case ($N = 1$) as well as in parallel. For the parallel runs, we consider both $N = 4$ and $N = 8$ parallel instances. We record the best solution found at 1 second intervals over 60 seconds.

We compare our VAL and P-VAL to multistart SA with fixed annealing length (FAL and P-FAL in parallel). We consider several fixed annealing lengths. FAL-1 and P-FAL-1 use an annealing length tuned to the total available time (60 seconds), specifically runs of 108 million SA evaluations. This threshold was determined based on the total number of evaluations that VAL was able to do in 60 seconds. Likewise, FAL-1/2, FAL-1/4, FAL-1/8, use annealing lengths that are $1/2$, $1/4$, and $1/8$ of the available 60 second limit (54 million, 27 million, and 13.5 million SA evaluations, respectively), restarting at that same length as long as time remains. P-FAL-1, P-FAL-1/2, P-FAL-1/4, P-FAL-1/8 are equivalent to a best of N, $2N$, $4N$, and $8N$ independent runs, respectively, with approximately the same total cost.

We represent solutions as permutations, and use Insertion Mutation as our neighborhood function. This operator removes a random element, and reinserts it at a different random position. Several existing metaheuristics for this scheduling problem use this operator, and it has been shown effective for permutation optimization problems characterized by asymmetric edges and general position within the permutation (Cicirello 2016). This problem has both: sequence-dependent setups (asymmetric edges), and due dates influence general position within permutation.

We use the following commonly employed metrics in the analysis of our experiments for this scheduling problem. Most commonly reported is the average percentage deviation from the optimal solutions, averaged only across the 98 instances with non-zero optimal values:

$$\%\Delta\text{Opt} = \frac{100}{N} \sum_{i=1}^{N} \frac{(S_i - O_i)}{O_i}, \tag{9}$$

where S_i and O_i are the value of the solution found for problem instance i and its optimal solution, respectively. One issue with this metric is that it ignores the 22 instances whose optimal solutions have weighted tardiness of 0. Thus, we also report the percentage deviation of the sum across all 120 instances relative to the sum of the optimal solutions:

$$\%\Delta\text{OptSum} = 100 \frac{\sum_{i=1}^{N} S_i - \sum_{i=1}^{N} O_i}{\sum_{i=1}^{N} O_i}. \tag{10}$$

Using each algorithm, we optimize each instance 10 times. We use t-tests to test the significance of the %ΔOptSum results. We use the Wilcoxon signed rank test to test the significance of the %ΔOpt. Since %ΔOpt is an average across multiple problem instances with values of varying scale, the t-test's normality requirement is not met.

4.3 Sequential Results

The results for sequential SA are summarized in Figures 2 and 3, which show average %ΔOpt and %ΔOptSum, respectively, throughout the duration of the 60 second runs. Early in the run, VAL dominates, and then performance approximately tracks that of each progressively longer fixed annealing length.

On the %ΔOpt metric, VAL dominates early in the run, but the FAL variations overtake it at approximately the time corresponding to the annealing length for which they were tuned. However, as VAL's longer runs complete, VAL's performance then matches that of the fixed length restarts. For

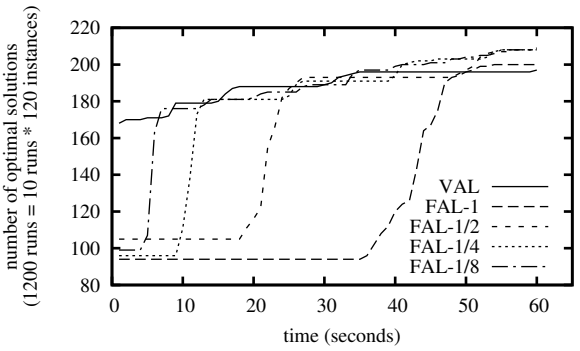

Figure 4: Sequential case: Number of optimals.

example, consider VAL versus FAL-1/8. For the first 6 seconds, VAL strongly dominates at extremely significant levels (p-values: $< 10^{-18}$). FAL-1/8 then outperforms VAL for approximately 2 seconds at significant levels (p-value at second 8 is < 0.0001). From that point onward, there are periods where VAL's increasingly longer runs enable it to gain a performance advantage over FAL-1/8 (at significant levels), and stretches with no significant performance difference. Next, consider VAL versus FAL-1, fixed annealing length as long as the experiment. FAL-1 catches up to VAL in performance (on $\%\Delta$Opt) at second 50, and its end of run performance is best among all variations considered. However, the end of run differences are not significant. At one second intervals, from second 48 to the end of the run, p-values (from Wilcoxon signed rank test) between VAL and FAL-1 are no lower than 0.06 and are as high as 0.86.

The results on $\%\Delta$OptSum are similar. For example, for the first 48 seconds, VAL outperforms FAL-1 at significant levels (t-test p-values from near zero early on to $p = 0.018$ at second 48). However, from second 49 to the end of the run, the difference in performance between VAL and FAL-1 is not significant (p-values > 0.35).

Figure 4 shows the number of optimal solutions found out of 1200 runs as a function of time. VAL dominates early in the run, finding more optimal solutions than the others. At the end of the run, the FAL variations all find optimal solutions slightly more often than VAL.

In the sequential case, the restart schedule of annealing lengths enables SA to approximate the performance of long runs near the end of the run, while simultaneously obtaining huge performance gains earlier in the run.

4.4 Parallel Results: 4 Parallel Instances

Figures 5 and 6 show average $\%\Delta$Opt and $\%\Delta$OptSum, respectively, for the duration of the 60 second runs for $N = 4$ parallel instances. Among the P-FAL variations, P-FAL-1 has the best performance on both metrics at the end of the run. However, the end of run differences among the P-FAL variations are not statistically significant (p-values > 0.13 for $\%\Delta$Opt and p-values > 0.08 for $\%\Delta$OptSum). Early in the run, P-VAL dominates relative to any fixed annealing length. For 90% of the run, P-VAL strongly dominates P-FAL-1, and dominates P-FAL-1/2 for nearly half the run, on

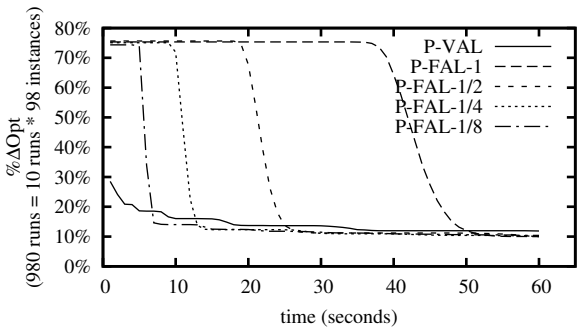

Figure 5: Parallel case ($N = 4$): $\%\Delta$Opt.

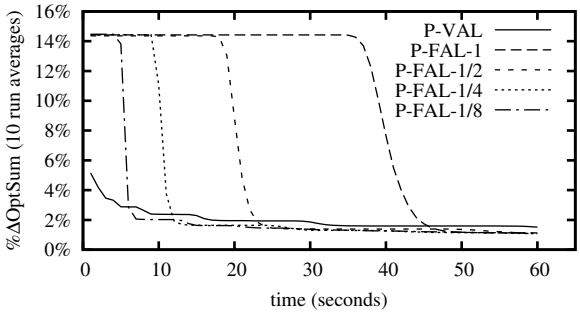

Figure 6: Parallel case ($N = 4$): $\%\Delta$OptSum.

both metrics, and similarly for P-FAL-1/4 and P-FAL-1/8. But late in the run, P-VAL does not match the performance of fixed annealing length as well as it did in the sequential case. Once the fixed annealing length is reached, P-FAL outperforms to end of run, while P-VAL exhibits superior performance during the run. All differences in performance between P-VAL and the P-FAL variations at each 1 second interval, except where the P-FAL curves cross the P-VAL curve, are statistically significant (p-values < 0.0001).

Although in the sequential case, VAL approximates the end of run performance of long fixed length runs, in parallel P-VAL is outperformed at the end of the run by the long fixed length runs. P-VAL's advantage, however, is in improved anytime performance early in the run.

4.5 P-VAL and P-VAL-0: 8 Parallel Instances

Earlier, we indicated that deficiencies exist in P-VAL-0 when $N > 4$. We consider this further here. First, examine the performance of P-VAL-0 relative to P-VAL in the case of 8 parallel instances. Figures 7 and 8 show average $\%\Delta$Opt and $\%\Delta$OptSum, respectively, for the duration of the 60 second runs. P-VAL strongly dominates P-VAL-0 throughout the run, on both metrics, at extremely statistically significant levels (p-values < 0.00001 at every one second interval).

Why is this the case? For P-VAL-0, restart r of SA_i is of length $1000 * 2^{i+r*N}$, and completes at time proportional to:

$$C_i(r) = 1000 \sum_{j=0}^{r} 2^{i+j*N} = 1000 * 2^i * \frac{2^{N(r+1)} - 1}{2^N - 1}, \quad (11)$$

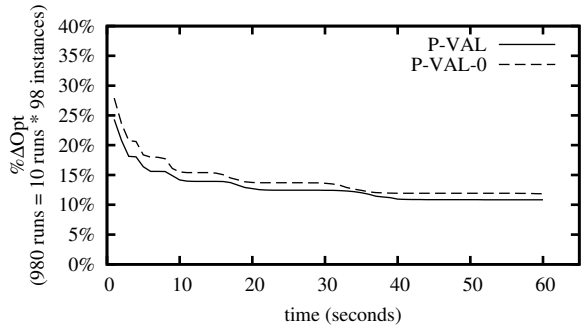

Figure 7: Parallel case ($N = 8$): %ΔOpt.

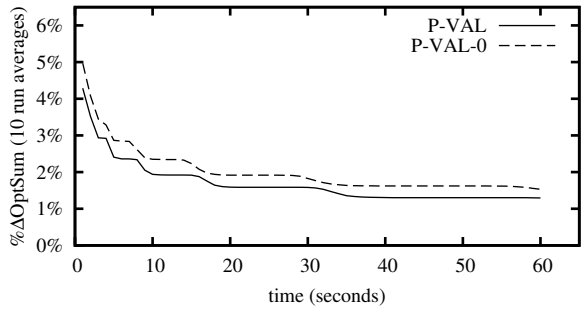

Figure 8: Parallel case ($N = 8$): %ΔOptSum.

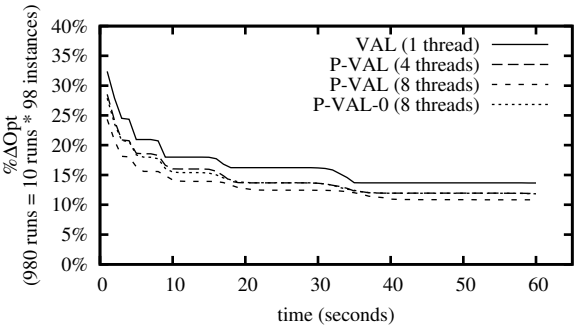

Figure 9: VAL vs P-VAL vs P-VAL-0: %ΔOpt.

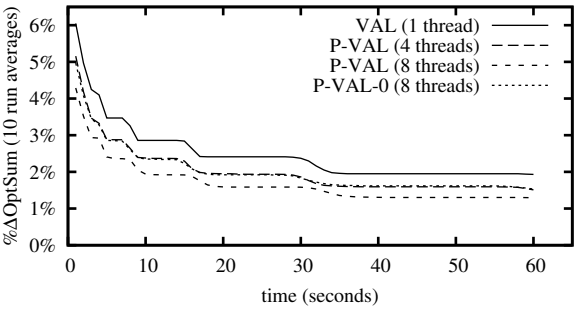

Figure 10: VAL vs P-VAL vs P-VAL-0: %ΔOptSum.

which is the sum of the annealing lengths up to and including restart r. The restart, r_0, of the sequential VAL that is of length $1000 * 2^{i+r*N}$ is $r_0 = i + r * N$, and completes at time proportional to:

$$C_0(r_0) = 1000 \sum_{j=0}^{i+r*N} 2^j = 1000 * (2^{i+r*N+1} - 1), \quad (12)$$

the sum of the run lengths up to and including restart r_0. If parallel speedup was impacted only by completing longer runs sooner, then the expected speedup factor is:

$$\frac{C_0(r_0)}{C_i(r)} = \frac{(2^{i+r*N+1} - 1)(2^N - 1)}{2^i(2^{N(r+1)} - 1)}. \quad (13)$$

In the limit as the number of restarts r grows large, the speedup due to completing longer runs earlier is:

$$\lim_{r\to\infty} \frac{C_0(r_0)}{C_i(r)} = \frac{2^N - 1}{2^{N-1}}. \quad (14)$$

For $N = 4$, the anticipated speedup from completing longer runs earlier is 1.875, and for $N = 8$ is approximately 1.992. In fact, in the limit as the number of parallel instances $N \to \infty$, the speedup factor approaches 2.0. Specifically, the longest completed restart finishes in half the time relative to the sequential VAL. With $N = 4$, P-VAL-0 is already approaching this limiting behavior, maximizing the benefit from shortening the time for the longest runs to complete.

We experimentally examine this in Figures 9 and 10, which show average %ΔOpt and %ΔOptSum, respectively, for a single sequential instance of VAL, P-VAL for both $N = 4$ and $N = 8$ parallel instances, as well as P-VAL-0 for $N = 8$ (recall that P-VAL and P-VAL-0 are identical for $N \leq 4$). Visually, it is impossible to distinguish the P-VAL-0 results with $N = 8$ from the $N = 4$ case for both metrics. The performance differences between these two cases are not significant. Using more than 4 parallel instances with P-VAL-0 does not improve performance.

The time for P-VAL with $N = 4$ to reach a given %ΔOpt (and likewise %ΔOptSum) is approximately one-half to one-third the time taken by the sequential VAL (speedup factor between 2 and 3), rather than the one-fourth we would expect from linear speedup. This is consistent with the analysis above of completion time of the longest runs. The speedup factor for P-VAL with $N = 8$ is approximately 4 (the sequential VAL takes approximately 4 times as long to reach equivalent levels of %ΔOpt and %ΔOptSum).

As you can see, as we increase N from 1 to 4 and then to 8, the difference in performance for P-VAL relative to VAL is consistent throughout the run. The speedup, through parallelization, is sublinear, but not limited by number of parallel instances.

4.6 Parallel Results: 8 Parallel Instances

Now consider $N = 8$ parallel instances. Figures 11 and 12 show average %ΔOpt and %ΔOptSum, respectively, for the duration of the 60 second runs. For most of the run, P-VAL very strongly dominates. Each of the P-FAL variations eventually overtake P-VAL as they near their tuned annealing

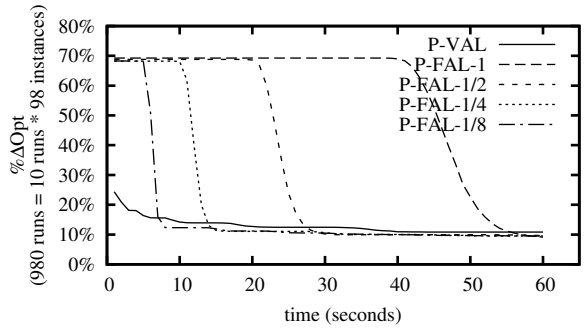

Figure 11: Parallel case ($N = 8$): $\%\Delta$Opt.

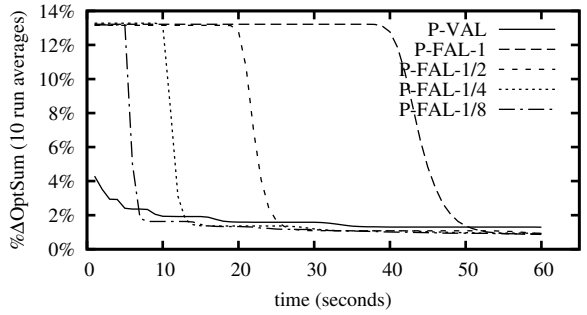

Figure 12: Parallel case ($N = 8$): $\%\Delta$OptSum.

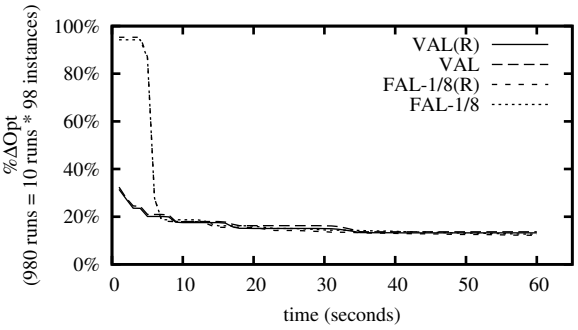

Figure 13: Reannealing Sequential: $\%\Delta$Opt.

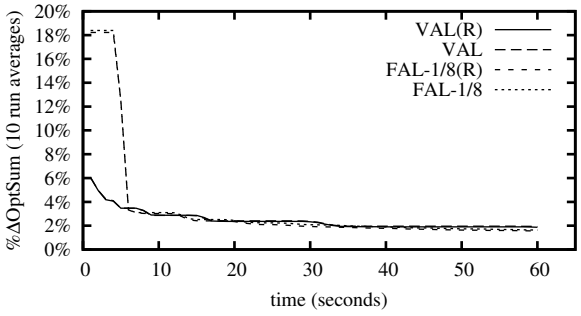

Figure 14: Reannealing Sequential: $\%\Delta$OptSum.

lengths. For example, P-FAL-1/2 overtakes P-VAL at second 26 on $\%\Delta$OptSum and second 28 on $\%\Delta$Opt, and P-FAL-1 overtakes P-VAL at second 51 on $\%\Delta$OptSum and second 56 on $\%\Delta$Opt. Though not evident from the scale of the graphs, P-FAL-1 outperforms all others by end of run, at statistically significant levels, on both metrics. We should expect P-FAL-1 to perform best at the end, as P-FAL-1 executes multiple long runs in parallel, providing the Modified Lam schedule with the actual experimental run length. This is also consistent with other findings that show a long run of SA is usually better than multiple independent short runs.

However, for 90% of the run, P-VAL strongly dominates P-FAL-1, and dominates P-FAL-1/2 for nearly half the run, on both metrics, and similarly for P-FAL-1/4 and P-FAL-1/8. P-VAL exhibits superior performance during the run.

The $\%\Delta$OptSum differences between P-VAL and P-FAL-1 are statistically significant (t-test p-values < 0.04) at every one-second interval; and the $\%\Delta$Opt results are significant (Wilcoxon signed rank test, p-values ranging from near-zero to 0.008), except where the curves cross ($p = 0.57$). The differences between P-VAL and each of P-FAL-1/2, P-FAL-1/4, and P-FAL-1/8 are statistically significant at every 1 second interval.

4.7 On the Efficacy of Reannealing

Thus far, all experimental results use independent restarts from random starting solutions, with no data sharing among parallel instances. We now explore what, if any, benefit is gained from reannealing prior solutions. Specifically, we consider VAL(R), where each restart reanneals the best of run solution rather than a random one. P-VAL(R) reanneals the current best of run solution across all parallel instances. Likewise, FAL-1/8(R) is FAL-1/8, but with reannealing of the best of run solution (similarly for P-FAL-1/8(R)).

The sequential results are found in Figures 13 and 14, and the parallel results (for $N = 8$) are in Figures 15 and 16. For variable annealing lengths, both sequential and parallel, the differences in performance between reannealing and random starting solutions are not statistically significant, except for the parallel case during the last 15-20 seconds of the run, where reannealing leads to marginally better results at statistically significant levels (p-values less than 0.01). Visually, in the graphs, it is virtually impossible to distinguish these. For fixed annealing length, both sequential and parallel, the differences in performance with and without reannealing are not statistically significant. Reannealing good solutions does not provide any benefit over independent runs in either the sequential or parallel case.

5 Conclusions

In this paper, we proposed a restart schedule for SA using the modified Lam annealing schedule. Our restart schedule eliminates the need to know the annealing length a priori. Relying on the often demonstrated property of SA that single long runs typically outperform multiple short runs, our restart schedule increases the annealing length at an exponential rate. The shorter runs at the beginning enable quickly finding "good" solutions, while the increasing an-

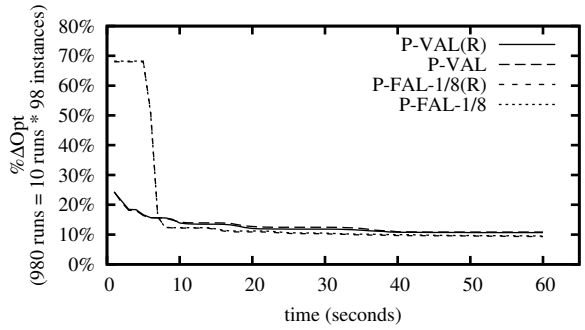

Figure 15: Reannealing Parallel ($N = 8$): $\%\Delta$Opt.

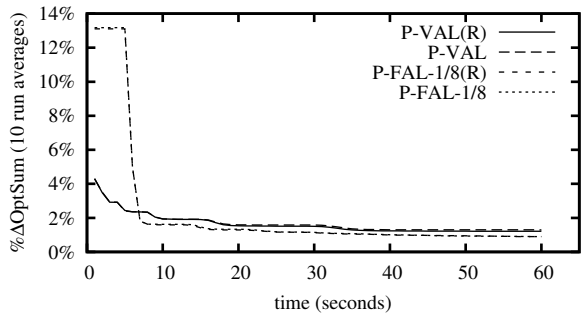

Figure 16: Reannealing Parallel ($N = 8$): $\%\Delta$OptSum.

nealing lengths of the restarts enable approximating the end of run performance of a single long run of SA.

Our restart schedule supports parallel implementation, using parallel independent SA instances that vary in initial annealing length, and with exponentially increasing restart lengths. The initial annealing lengths are staggered to ensure that longer runs are already in progress as shorter runs complete. Our aim is to balance the risk associated with errors in determining the time available for problem solving.

The end-of-run behavior observed in experiments, both sequential as well as in parallel, with a sequence-dependent scheduling problem confirm the commonly found property that a longer SA run outperforms restarts of a shorter run. However, performance during the run is often overlooked. For example, although FAL-1 performs best at the end of run, FAL-1/2 achieved better results at the mid-way point. Our annealing length schedule VAL, and in parallel P-VAL, exhibited stronger anytime behavior throughout the run.

References

Aleti, A., and Moser, I. 2013. Entropy-based adaptive range parameter control for evolutionary algorithms. In *Proc. GECCO*, 1501–1508. ACM.

Boyan, J. A. 1998. *Learning Evaluation Functions for Global Optimization*. Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, PA.

Cicirello, V. A., and Smith, S. F. 2005. Enhancing stochastic search performance by value-biased randomization of heuristics. *Journal of Heuristics* 11(1):5–34.

Cicirello, V. A. 2003a. *Boosting Stochastic Problem Solvers Through Online Self-Analysis of Performance*. Ph.D. Dissertation, Carnegie Mellon University.

Cicirello, V. A. 2003b. Weighted tardiness scheduling with sequence-dependent setups: A benchmark library. Tech. report, ICL Lab, CMU. `http://www.cicirello.org/datasets/wtsds/`.

Cicirello, V. A. 2006. Non-wrapping order crossover: An order preserving crossover operator that respects absolute position. In *Proc. GECCO'06*. ACM. 1125–1131.

Cicirello, V. A. 2007. On the design of an adaptive simulated annealing algorithm. In *Proc. CP 2007 First Workshop on Autonomous Search*. AAAI Press.

Cicirello, V. A. 2015. Genetic algorithm parameter control: Application to scheduling with sequence-dependent setups. In *Proc. 9th Int. Conf. Bio-inspired Information and Communications Technologies*. ICST. 136–143.

Cicirello, V. A. 2016. The permutation in a haystack problem and the calculus of search landscapes. *IEEE Transactions on Evolutionary Computation* 20(3):434–446.

Cire, A. A.; Kadioglu, S.; and Sellmann, M. 2014. Parallel restarted search. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 842–848. AAAI Press.

Eiben, A. E.; Hinterding, R.; and Michalewicz, Z. 1999. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 3(2):124–141.

Gomes, C. P.; Selman, B.; Crato, N.; and Kautz, H. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning* 24(1):67–100.

Jha, S., and Menon, V. 2014. Bbmttp: Beat-based parallel simulated annealing algorithm on gpgpus for the mirrored traveling tournament problem. In *Proceedings of the High Performance Computing Symposium*, 3:1–3:7. Society for Computer Simulation International.

Lam, J., and Delosme, J. 1988. Performance of a new annealing schedule. In *Proc. 25th ACM/IEEE DAC*, 306–311.

Liao, C.-J., and Juan, H.-C. 2007. An ant colony optimization for single-machine tardiness scheduling with sequence-dependent setups. *Computers and Operations Research* 34(7):1899–1909.

Liao, C.-J.; Tsou, H.-H.; and Huang, K.-L. 2012. Neighborhood search procedures for single machine tardiness scheduling with sequence-dependent setups. *Theoretical Comp. Sci.* 434:45–52.

Luby, M.; Sinclair, A.; and Zuckerman, D. 1993. Optimal speedup of las vegas algorithms. *Information Processing Letters* 47(4):173–180.

Ludwin, A., and Betz, V. 2011. Efficient and deterministic parallel placement for fpgas. *ACM Trans. Des. Autom. Electron. Syst.* 16(3):22:1–22:23.

Morton, T. E., and Pentico, D. W. 1993. *Heuristic Scheduling Systems: With Applications to Production Systems and Project Management*. Wiley.

Rahimian, F.; Payberah, A. H.; Girdzijauskas, S.; Jelasity, M.; and Haridi, S. 2015. A distributed algorithm for large-scale graph partitioning. *ACM Trans. Auton. Adapt. Syst.* 10(2):12:1–12:24.

Ram, D. J.; Sreenivas, T. H.; and Subramaniam, K. G. 1996. Parallel simulated annealing algorithms. *Journal of Parallel and Distributed Computing* 37:207212.

Rudolph, G. 1993. Massively parallel simulated annealing and its relation to evolutionary algorithms. *Evol. Comput.* 1(4):361–383.

Sen, A. K., and Bagchi, A. 1996. Graph search methods for non-order-preserving evaluation functions: Applications to job sequencing problems. *AIJ* 86(1):43–73.

Swartz, W. P. 1993. *Automatic Layout of Analog and Digital Mixed Macro/Standard Cell Integrated Circuits*. Ph.D. Dissertation, Yale University.

Tanaka, S., and Araki, M. 2013. An exact algorithm for the single-machine total weighted tardiness problem with sequence-dependent setup times. *Computers and Operations Research* 40(1):344–352.

Wessing, S.; Preuss, M.; and Rudolph, G. 2011. When parameter tuning actually is parameter control. In *Proc. GECCO*, 821–828. ACM.

Xu, H.; Lü, Z.; and Cheng, T. C. 2014. Iterated local search for single-machine scheduling with sequence-dependent setup times to minimize total weighted tardiness. *J. of Scheduling* 17(3):271–287.