The Thirty-First International Florida
Artificial Intelligence Research Society Conference (FLAIRS-31)

Impact of Random Number Generation on Parallel Genetic Algorithms

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 present a parallel genetic algorithm (pGA) with adaptive control parameters and permutation representation for weighted tardiness scheduling with sequence-dependent setups, an NP-Hard problem. This pGA provides a linear to slightly superlinear speedup relative to its sequential counterpart. As part of our research, we explore the effects of different random number generation algorithms on the runtimes of both sequential and parallel GAs. GAs and other forms of evolutionary computation rely so heavily on random number generation that our results show that we can obtain a 20% increase in the speed of a pGA, and an over 25% increase in the speed of a sequential GA, simply by careful choice of random number generator—both the underlying generator as well as algorithms for specific number types such as Gaussian often needed for mutating real-valued genes.

1 Introduction

A genetic algorithm (GA) solves problems through simulated evolution, evolving a population of candidate solutions, usually represented as bitstrings, over many generations using crossover to recombine genes of parent solutions, and mutation to randomly perturb population members (Mitchell 1998). GAs are naturally parallelized. Luque and Alba provide an excellent introduction and survey of parallel GA (pGA) implementation (Luque and Alba 2011). Some pGA maintain a single large population, dividing work of a generation among multiple processors; while others use multiple subpopulations, one per processor, with periodic migration of a few individuals among the subpopulations.

GAs rely heavily on random numbers. For example, the crossover rate is the probability of applying the crossover operator (and likewise the mutation rate). Implementation of this behavior requires uniform random floating-point numbers. Many crossover operators require uniform random integers for index selection. The most common mutation operator for real-valued representations, Gaussian mutation (Hinterding 1995), requires implementation of a Gaussian random variable. The pseudorandom number generator (PRNG) found in most programming languages is the linear congruential method (Knuth 1998), and if the language provides built-in support for Gaussians at all, it is usually via the polar method (Knuth 1998). Both are slow compared to alternatives, such as SplitMix/DotMix (Steele, Lea, and Flood 2014; Leiserson, Schardl, and Sukha 2012) instead of linear congruential, and the ziggurat method (Marsaglia and Tsang 2000) instead of the polar method. In this paper, we show that a GA's speed can vary by as much as 25%, and a pGA by as much as 20%, simply through choice of PRNG.

While investigating the effects of PRNGs on GA performance, we developed a pGA for weighted tardiness scheduling with sequence-dependent setups, an NP-Hard problem without setups (Morton and Pentico 1993) whose difficulty is greatly magnified by the sequence-dependent setups (Sen and Bagchi 1996). Many algorithms exist for this scheduling problem (Cicirello 2017; Xu, Lü, and Cheng 2014; Tanaka and Araki 2013; Liao, Tsou, and Huang 2012; Liao and Juan 2007; Cicirello and Smith 2005). We adapt two prior GAs (Cicirello 2015; 2006) into multi-population pGAs, one with adaptive control parameters and one with static control parameters. Our results show that the pGA with adaptive parameters provides a linear to slightly superlinear speedup relative to its sequential counterpart.

We first describe the existing GAs, and then present our multi-population pGA variations of those GAs. Next, we explore different PRNGs in isolation from the pGA on a simple computational task, and then with the pGAs showing the extreme effects that PRNG choice has on GA/pGA runtimes. Finally, we present results on the problem solving effectiveness of our new pGA on an NP-Hard scheduling problem.

2 Sequential Genetic Algorithms

2.1 Scheduling with Sequence-Dependent Setups

Weighted tardiness scheduling with sequence-dependent setups is an NP-Hard single-machine scheduling problem. The current best exact solver, which uses dynamic programming, can optimally solve all available benchmarks, but requires over two weeks of memory-intensive CPU time to solve the harder instances (Tanaka and Araki 2013). Thus, alternatives are desirable, such as GAs and other metaheuristics, which can more efficiently find sufficiently-optimal solutions.

We use the standard benchmark set (Cicirello 2003a; 2003b), which has been used by many researchers for a variety of algorithms, such as dynamic programming (Tanaka and Araki 2013), neighborhood search (Liao, Tsou, and

Huang 2012), iterated local search (Xu, Lü, and Cheng 2014), stochastic sampling (Cicirello and Smith 2005), ant colony optimization (Liao and Juan 2007), simulated annealing (Cicirello 2017; 2007), etc.

The problem consists of N jobs, $J = \{j_1, j_2, \ldots, j_N\}$, each with weight w_k, duedate d_k, process time p_k, and setup time $s_{i,k}$ prior to processing j_k if it follows j_i. Setup times are asymmetric, $s_{i,k} \neq s_{k,i}$, and $s_{0,k}$ is the initial setup time if j_k is first. The jobs must be sequenced to minimize:

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

$\pi(k)$ is the position of j_k. The completion time c_k is the sum of the process and setup times of j_k and all preceding jobs:

$$c_k = \sum_{\pi(x) \leq \pi(k), \pi(x) = \pi(y)+1} (p_x + s_{y,x}). \quad (2)$$

2.2 Shared Features of the Genetic Algorithms

The two existing sequential GAs (Cicirello 2006; 2015), which are the basis for our pGA, share several features.

Preprocessing: To minimize the impact of setup times on performance, we increase the process time of each job by its minimum setup time, reducing all setup times accordingly:

$$s_k^{\min} = \min_{0 \leq i \leq N, i \neq k} s_{i,k}, \quad (3)$$

$$p_k = p_k + s_k^{\min}, \quad (4)$$

$$s_{i,k} = s_{i,k} - s_k^{\min}, \forall i, i \neq k, 0 \leq i \leq N. \quad (5)$$

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

Representation and Fitness: Both GAs use the permutation representation, which offers a direct mapping between internal GA representation and solutions to the scheduling problem at hand. Let $T(P_i)$ be the weighted tardiness (Equation 1) for permutation P_i; and define fitness as follows:

$$\textit{fitness}(P_i) = 1 - T(P_i) + \max_{1 \leq k \leq \textit{PopSize}} T(P_k), \quad (6)$$

where *PopSize* is the population size. Higher fitness values correspond to better schedules (lower values of $T(P_i)$), and the least fit individual has fitness equal to 1.

Selection: Elitism selects the E most fit unique permutations, which do not undergo crossover or mutation, ensuring that the population always contains the best solution thus far, and preventing convergence to a single solution since the population always contains at least E unique permutations. Stochastic Universal Sampling (SUS) (Baker 1987) selects the remaining $\textit{PopSize} - E$ from the entire population including the elite. SUS selects P_i with probability proportional to $\textit{fitness}(P_i)$ like fitness proportionate selection. However, SUS is like spinning a wheel with k equidistant pointers once to select k members simultaneously, whereas fitness proportionate spins a 1-pointer k times. SUS reduces selection bias (Baker 1987), and it is also more efficient (e.g., only 1 random number required to select entire population).

Non-Wrapping Order Crossover (NWOX): NWOX (Cicirello 2006), a variation of Order Crossover (OX) (Davis 1985), enables child permutations to inherit edges from parents, while minimizing positional deviation relative to the parents, unlike OX which displaces elements large distances from locations in parents. Due to the sequence-dependent setups, edges directly impact fitness; and due to the weighted tardiness objective, general element position impacts duedate achievement, which in turn impacts fitness. NWOX balances these properties. Figure 1 shows how it works. Given parents, (P_i, P_j), choose two random cross points. Child C_i inherits the cross region elements' positions from parent P_j, and the relative order of the remaining elements from P_i, filling in C_i left to right, skipping the cross region (likewise for C_j). The original OX began filling in the remaining elements after the cross region, wrapping to the left.

(1) Pick 2 random points, defining cross region (cr).	p1: [A B C | D E F | G H K L] (cr1) p2: [C F D | A B H | L E K G] (cr2)
(2) Find p1-cr2 & p2-cr1.	p1-cr2: [C D E F G K L] p2-cr1: [C A B H L K G]
(3) Initialize each child with other parent's cr.	c1: [- - - | A B H | - - - -] c2: [- - - | D E F | - - - -]
(4) Copy p1-cr2 into c1 beginning at left, skipping cross region, and retaining order (likewise for c2).	c1: [C D E | A B H | F G K L] c2: [C A B | D E F | H L K G]

Figure 1: The NWOX operator.

Mutation: Insertion mutation removes a random element, and reinserts it at a different randomly chosen location, shifting all elements between the removal and reinsertion points. Prior research on permutation search landscapes (Cicirello 2016) shows that insertion mutation is ideally suited to problems with directed edges (e.g., asymmetric, sequence-dependent setups), and when positions impact fitness (e.g., general position within permutation affects job tardiness).

2.3 Static Versus Adaptive Control Parameters

The fundamental difference between the two GAs is how control parameters are set. The first uses static control parameters that were tuned manually using a set of training problem instances (Cicirello 2006): $\textit{PopSize} = 100$, $E = 3$, crossover rate $C = 0.95$, and mutation rate $M = 0.65$. A mutation rate of 0.65 would be unusually high for a bitstring GA where it is a per-bit rate. However, the mutation rate for a permutation-based GA is per population member (i.e., probability that one mutation is applied to a population member), so mutation rates tend to be higher than for bitstrings.

The second GA dynamically adapts the control parameters using search feedback (Cicirello 2015). Each population member is defined as: $\textit{Pop}_i =< P_i, C_i, M_i, \sigma_i >$. P_i is a permutation. C_i and M_i are the crossover and mutation rates for $\textit{Pop}_i$; and σ_i is used in mutating C_i and M_i. The non-elite $\textit{Pop}_i$ are paired randomly. The C_i of an arbitrary member of each pair determines if crossover occurs. Crossover is not applied to the control parameters of $\textit{Pop}_i$. The permutation of each non-elite $\textit{Pop}_i$ is mutated with probability M_i.

We initialize C_i and M_i uniformly at random from $[0.1, 1.0)$, and σ_i uniformly from $[0.05, 0.15)$. In each generation, Gaussian mutation (Hinterding 1995) is applied to the control parameters of the non-elite members as follows:

$$C_i = C_i + N(0, \sigma_i), \tag{7}$$

$$M_i = M_i + N(0, \sigma_i), \tag{8}$$

$$\sigma_i = \sigma_i + N(0, 0.01), \tag{9}$$

where $N(0, \sigma)$ is a Gaussian random variable with mean 0 and standard deviation σ. If $C_i > 1$, it is set to 1 to ensure valid probabilities. If $C_i < 0.1$, it is set to 0.1. Likewise, M_i varies within $[0.1, 1]$ and σ_i within $[0.01, 0.2]$. *PopSize* and E were tuned manually (*PopSize* $= 100$, $E = 5$).

3 Parallel Genetic Algorithms

In this paper, we explore parallel versions of the two existing GAs. Our pGA is a multi-population model that executes k instances of a GA in parallel. This is a common approach to pGA implementation (Luque and Alba 2011). Typically, multi-population pGAs use migration, where members periodically migrate between subpopulations. Some approaches involve migration to any other subpopulation, while others define a migration topology. The migration rate is critical to performance (Lässig and Sudholt 2010). If it is too low, the subpopulations remain isolated which can lead to lower quality solutions compared to a single large population GA. The migration interval also plays an important role: if migration is too infrequent, convergence is slow, while too frequent migration leads to a loss of diversity (Skolicki and De Jong 2005). There are also approaches that adapt the migration interval dynamically (Mambrini and Sudholt 2014).

Since one of our objectives is exploring the effects of different PRNGs on pGA performance, our pGA uses k isolated subpopulations to eliminate communication overhead of migration. Small GA random initial populations tend to lie in different basins of attraction, leading isolated subpopulations to converge upon different local optima. Thus, this is like a restart strategy common for other forms of local search, though rarely used by GAs. We consider two pGAs.

Static Parameters: This pGA concurrently executes k GAs with NWOX, insertion mutation, SUS selection with elitism, and fixed control parameters.

Adaptive Parameters: This pGA uses the same operators, but with adaptive control parameters.

4 Random Number Generator Comparison

Before exploring how PRNGs affect pGA runtimes, we first benchmark the available PRNGs in isolation of the GA, using a simple task, summing a sequence of random numbers, that involves little computation beyond the PRNG itself.

We use Java 8, the Java HotSpot 64-bit Server VM, and consider three of Java's PRNG classes: Random, SplittableRandom, and ThreadLocalRandom.

Java's *Random* class implements the linear congruential method (Knuth 1998) with a 48-bit seed. Although threadsafe, using an AtomicLong for the seed, multiple threads sharing a single instance encounter contention issues, especially with frequent random number generation. Thus, in our experiments, we provide each thread with its own instance.

SplittableRandom implements SplitMix (Steele, Lea, and Flood 2014), an optimized version of DotMix (Leiserson, Schardl, and Sukha 2012), that is much faster, and passes more rigorous statistical testing (e.g., DieHarder (Brown, Eddelbuettel, and Bauer 2013)), than the linear congruential method. SplittableRandom is not threadsafe, but provides a split method that spawns new instances to provide worker threads with random number generation capability.

ThreadLocalRandom also implements SplitMix, but lacks the split functionality. A single static instance maintains seed data local to each thread, managing one PRNG per thread.

We vary the number of threads according to $k = 2^i$ where $i \in \{0, \ldots, 8\}$, each generating and summing $256/k$ million numbers, 256 million numbers total. We consider uniform random 32-bit integers (Java's int type), uniform random 64-bit floating-point numbers (Java's double type), and Gaussian random numbers. All three Java classes generate uniform random numbers. Random and ThreadLocalRandom generate Gaussians with the polar method (Knuth 1998). SplittableRandom does not provide Gaussian support. Thus, we use our own polar method implementation; and also a much faster alternative, the ziggurat method (Marsaglia and Tsang 2000), which we ported to Java from the GNU Scientific Library's C implementation (Voss 2005).

For each combination of PRNG, number type, and number of threads, we compute the average time over 30 runs, 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.

Figure 2 shows lin-log plots of average time as a function of number of threads. ThreadLocalRandom and SplittableRandom perform equivalently for all number types, and number of threads, except uniform random floating-point numbers with more than 32 threads, where the difference is negligible and occurs with multiple threads per core, or Gaussians with ziggurat and less than four threads, where the difference is also small and diminishes with few threads. Both implement the same PRNG, so thread management is the only real difference. We must maintain one SplittableRandom instance per thread in our code; while ThreadLocalRandom's single static instance maintains local seed data in each thread, simplifying our code. Both significantly outperform the Random class, demonstrating SplitMix's efficiency over linear congruential. Ziggurat is also much faster than the polar method, consistent with prior results (Marsaglia and Tsang 2000; Voss 2005).

The benefit of increasing the number of threads varies by number type. The least costly to generate, uniform 32-bit integers, has the smallest gain (speedup factor of 2.5 for 4 threads, and 2.8 for 8 threads using ThreadLocalRandom). For random floating-point numbers, speedup is 2.6, 3.2, and 3.3 for 4, 8, and 16 threads, respectively; and for Gaussians generated with the ziggurat method, speedup is 3.4, 5.1, and 6.5 for 4, 8, and 16 threads. Speedup is greater for the polar method, however, the ziggurat method is much faster.

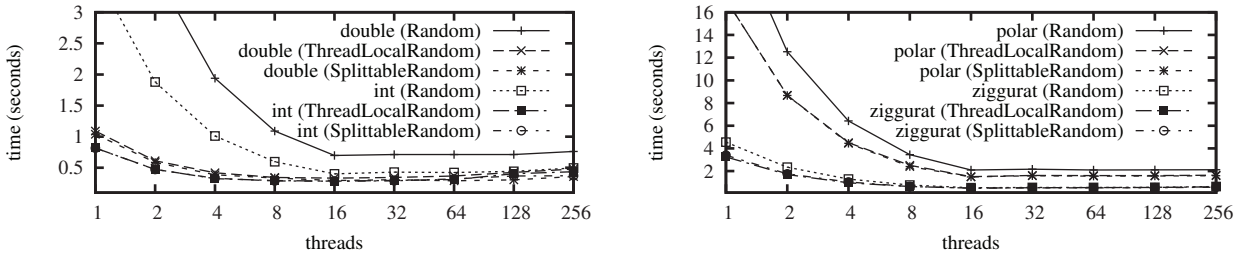


Figure 2: Simple task time: uniform doubles and integers (left), Gaussian with polar and ziggurat algorithms (right).

5 Parallel Genetic Algorithm Runtimes

We now explore the effects on the runtimes of both pGA variations: static control parameters, and adaptive control parameters. For each, we run two sets of experiments:

Fixed Subpopulation Size: A constant subpopulation size of $PopSize = 100$ (same as the existing sequential GAs) makes the total population size of the pGA: $TotPopSize = PopSize * k = 100k$, where k is the number of parallel instances. The number of generations is $Gens = 64000/k$, maintaining an approximately equal amount of total work in terms of number of applications of the genetic operators.

Fixed Total Population Size: The total population size and number of generations are held constant ($TotPopSize = 1600$, $Gens = 4000$). The subpopulation size varies with $PopSize = TotPopSize/k = 1600/k$, such that the total work is approximately that of the first set of experiments.

The benchmark set has 120 instances with varying degrees of duedate tightness and range, and setup severity. We exclude 22 of the 40 loose duedate instances, whose optimal weighted tardiness is $T = 0$ to avoid biasing runtimes, since once a solution with $T = 0$ is found, the search terminates.

For each combination of pGA, population size option, PRNG, and number of parallel instances, we execute 10 runs on each of the 98 instances. Thus, we report 980 run averages. We vary the number of parallel instances, $k = 2^i$ where $i \in \{0, \ldots, 6\}$. We exclude SplittableRandom since it exhibits nearly identical performance, and implements the same PRNG, as ThreadLocalRandom.

Figure 3 shows that using ThreadLocalRandom significantly speeds up runtime compared to using Random. For example, for adaptive parameters and fixed subpopulation size (Figure 3(c)), using ThreadLocalRandom and the ziggurat algorithm is 4%, 6%, and 7% faster than using Random and the ziggurat algorithm for 1, 2, and 4 threads, respectively. Similarly, for adaptive parameters and fixed total population size (Figure 3(d)), the pGA is 6%, 6%, and 5% faster for 1, 2, and 4 threads, when using ThreadLocalRandom and ziggurat compared to using Random and ziggurat.

For the adaptive parameters pGA, generating Gaussian random numbers with the ziggurat algorithm, for Gaussian mutation, rather than the polar method has a much larger effect than does the underlying PRNG. For example, using ThreadLocalRandom, the pGA using the ziggurat algorithm is 15%, 15%, and 13% faster than using the polar method, for fixed subpopulation size (Figure 3(c)) and 1, 2, and 4 threads, respectively; and 18%, 17%, and 13% faster for fixed total population size (Figure 3(d)). The most extreme performance difference between PRNGs is using ThreadLocalRandom and ziggurat compared to Random and the polar method. The former is 25%, 22%, and 20% faster than the latter (for 1, 2, and 4 threads) with a fixed subpopulation, and 28%, 27%, and 22% faster with a fixed total population.

6 Parallel GA Problem Solving Effectiveness

Due to the strong runtime advantage established in Section 5, we adopt Java's ThreadLocalRandom as the PRNG and the ziggurat algorithm for Gaussians. There is a negligible time advantage (Figure 3) to using more than eight concurrent instances, and once the number of concurrent instances reaches the number of logical cores of our system (16) performance degrades (e.g., due to context switching, etc). Thus, we use 8 parallel populations for all pGA variations, with subpopulation size $PopSize = 100$, and total population size of $TotPopSize = 800$. The population size of the sequential GAs is set to 100, as was originally the case.

The benchmark set has 120 instances, 40 each of loose, medium, and tight duedates. We measure performance in two ways. The first is average percentage deviation from the optimals, excluding 22 loose duedate instances with $T = 0$:

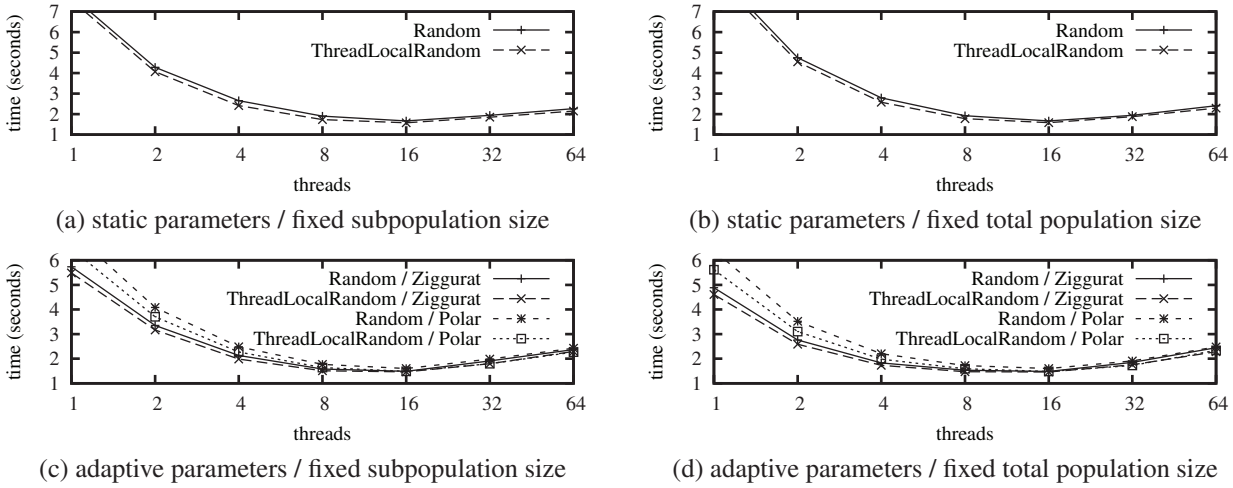
$$\%\Delta\text{Opt} = \frac{100}{N} \sum_{i=1}^{N} \frac{(S_i - O_i)}{O_i}. \tag{10}$$

S_i is the value of the solution found for instance i and O_i is its optimal solution. Since 22 instances are excluded by $\%\Delta$Opt, we also report percentage deviation of the sum across all instances relative to the sum of the optimals:

$$\%\Delta\text{OptSum} = 100 \frac{\sum_{i=1}^{N} S_i - \sum_{i=1}^{N} O_i}{\sum_{i=1}^{N} O_i}. \tag{11}$$

For each run length ($\{10^2, 10^3, 10^4, 10^5, 10^6\}$ generations) and GA (adaptive vs static parameters, parallel vs sequential), we solve each instance 10 times. Thus, reported runtimes are 1200 run averages, $\%\Delta$OptSum values are 10 run averages, and $\%\Delta$Opt values are 980 run averages.

Figure 4 shows log-log plots of $\%\Delta$OptSum and $\%\Delta$Opt as functions of time. The four GA configurations run at different rates due to: (a) impact of thread overhead on short runs, (b) parameter adaptation overhead, and (c) varying



(a) static parameters / fixed subpopulation size

(b) static parameters / fixed total population size

(c) adaptive parameters / fixed subpopulation size

(d) adaptive parameters / fixed total population size

Figure 3: pGA runtime as function of number of threads.

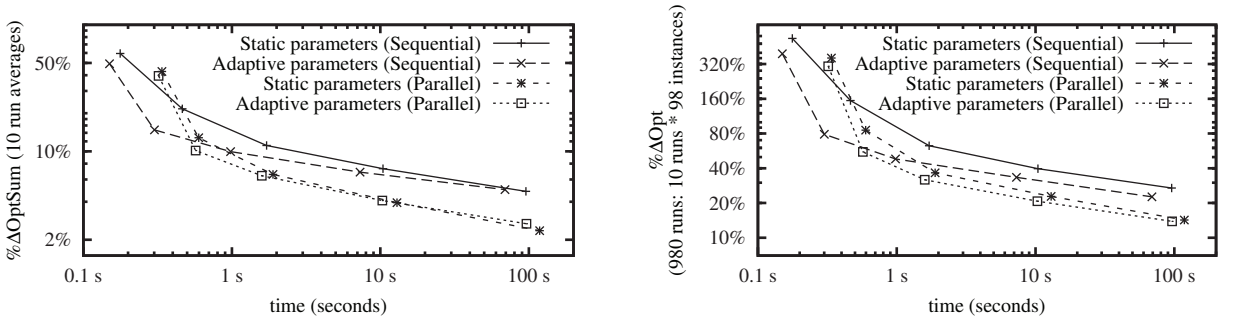


Figure 4: GA and pGA with adaptive and static parameters: log-log plots of %ΔOptSum and %ΔOpt vs run time.

Table 1: Comparison of pGA with adaptive (A) and static (S) control parameters for varying number of generations (G).

G	%ΔOptSum			%ΔOpt			Runtime	
	A	S	p	A	S	p	A	S
10^2	40.	43.	∼0	306.	361.	0	0.3	0.3
10^3	10.	13.	∼0	55.	86.	0	0.6	0.6
10^4	6.4	6.6	.09	32.	37.	∼0	1.6	1.9
10^5	4.1	3.9	.01	21.	23.	.00	10.	13.
10^6	2.7	2.4	∼0	14.	14.	.04	96.	118.

time per generation due to fluctuating control parameter values. Thus, the x-axis shows time, rather than generations.

For very short runs (100 generations), there is no benefit to parallelization due to thread management overhead. A single small population with adaptive parameters is sufficient. However, for 1000 or more generations, the pGA quickly overtakes the sequential GA. In $7s$, the adaptive pGA's %ΔOptSum is equivalent to a $69s$ run of the sequential adaptive GA, a slightly superlinear speedup factor near 10 given 8 parallel subpopulations. And in $8.8s$, the adaptive pGA's %ΔOpt is equivalent to a $69s$ run of the sequential GA, for a linear speedup factor around 8.

Table 1 lists %ΔOptSum and %ΔOpt for the pGAs (adaptive vs static control parameters). Runtimes are reported in seconds. Significance of %ΔOptSum is tested with t-tests, and that of %ΔOpt with the Wilcoxon signed rank test. Because %ΔOpt is an average across multiple instances with values of varying scale, the t-test normality requirement is not met. P-values are shown.

For %ΔOpt, the adaptive pGA, with high levels of statistical significance, outperforms the pGA with static parameters at all run lengths. The degree of statistical significance decreases with run length, which is expected since the longer the run, the nearer both pGAs are to the optimals. The runtimes highlight the relative strength of evolving the parameters, as the adaptive pGA requires less time for the same number of generations. For the sequential GA, the evolved crossover and mutation rates decline later in the run, leading to less applications of the genetic operators (Cicirello 2015). This also explains the speed difference of the two pGAs.

For %ΔOptSum, the adaptive pGA very significantly outperforms the pGA with static parameters for runs of 1000 or less generations. At 10000 generations, no statistical significance is seen, although the adaptive pGA is faster. For runs of 100000 or more generations, results are less clear: static parameters lead to slightly lower %ΔOptSum (with high

statistical significance), but the adaptive pGA is 20% faster. The apparent inconsistency across the two measures of solution quality is because $\%\Delta$Opt ignores the 22 instances with optimal solution $T = 0$, while $\%\Delta$OptSum does not.

7 Conclusions

We introduced a new pGA for weighted tardiness scheduling with sequence-dependent setups. Although other GAs for the problem exist, this is the first parallel GA that we are aware of for this NP-Hard problem. Our multipopulation pGA evolves the control parameters, eliminating the often tedious and error prone parameter tuning phase of GA development. This new pGA exhibits linear to slightly superlinear speedup relative to its sequential counterpart.

Among our research objectives was the performance impact that choice of PRNG has on both parallel and sequential GA runtimes. We showed that implementation decisions related to PRNGs can substantially affect GA efficiency. With careful selection of PRNG and all other implementation details equal, a sequential GA can be over 25% faster than one that relies on the standard built-in language support for random number generation; and similarly, a pGA can be up to 20% faster. These results highlight just how much of a GA's time is spent generating random numbers. The GA practitioner should not necessarily rely solely on their chosen language's provided PRNG, and may need to employ alternatives from external libraries or their own implementations.

References

Baker, J. 1987. Reducing bias and inefficiency in the selection algorithm. In *Proc ICGA*. Lawrence Erlbaum. 14–21.

Brown, R. G.; Eddelbuettel, D.; and Bauer, D. 2013. *Dieharder: A random number test suite*. https://www.phy.duke.edu/~rgb/General/dieharder.php.

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, Robotics, Carnegie Mellon University.

Cicirello, V. A. 2003b. Weighted tardiness scheduling with sequence-dependent setups: A benchmark library. Tech. report, ICL Lab, CMU.

Cicirello, V. A. 2006. Non-wrapping order crossover: An order preserving crossover operator that respects absolute position. In *Proc. GECCO'06*, volume 2. 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. on 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.

Cicirello, V. A. 2017. Variable annealing length and parallelism in simulated annealing. In *Proc 10th Int Symposium on Combinatorial Search*, 2–10. AAAI Press.

Davis, L. 1985. Applying adaptive algorithms to epistatic domains. In *Proc. IJCAI*. Morgan Kaufmann. 162–164.

Hinterding, R. 1995. Gaussian mutation and self-adaption for numeric genetic algorithms. In *IEEE CEC*. IEEE Press. 384–389.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison Wesley.

Lässig, J., and Sudholt, D. 2010. The benefit of migration in parallel evolutionary algorithms. In *Proc. 12th GECCO*. ACM. 1105–1112.

Leiserson, C. E.; Schardl, T. B.; and Sukha, J. 2012. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proc PPoPP*. ACM. 193–204.

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 Computer Science* 434:45–52.

Luque, G., and Alba, E. 2011. *Parallel Genetic Algorithms: Theory and Real World Applications*. Springer.

Mambrini, A., and Sudholt, D. 2014. Design and analysis of adaptive migration intervals in parallel evolutionary algorithms. In *Proc. GECCO*. ACM. 1047–1054.

Marsaglia, G., and Tsang, W. W. 2000. The ziggurat method for generating random variables. *J. Stat. Softw.* 5(1):1–7.

Mitchell, M. 1998. *An Introduction to Genetic Algorithms*. MIT Press.

Morton, T. E., and Pentico, D. W. 1993. *Heuristic Scheduling Systems*. John Wiley and Sons.

Sen, A. K., and Bagchi, A. 1996. Graph search methods for non-order-preserving evaluation functions. *AIJ* 86:43–73.

Skolicki, Z., and De Jong, K. 2005. The influence of migration sizes and intervals on island models. In *Proc. 7th GECCO*. ACM. 1295–1302.

Steele, Jr., G. L.; Lea, D.; and Flood, C. H. 2014. Fast splittable pseudorandom number generators. In *Proc. ACM OOPSLA*. ACM. 453–472.

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.

Voss, J. 2005. *The Ziggurat Method for Generating Gaussian Random Numbers*. GSL: GNU Scientific Library, http://www.seehuhn.de/pages/ziggurat.

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.