Impact of Random Number Generation on Parallel Genetic Algorithms

Vincent A. Cicirello, Ph.D.

Professor of Computer Science / Behavioral Neuroscience

cicirelv@stockton.edu https://www.cicirello.org/

Introduction

- Genetic algorithms (GA), and other forms of evolutionary computation, solve problems through simulated evolution.
  - Evolve populations of solutions using evolution inspired operators, such as mutation, crossover, selection, etc.
- We introduce a Parallel GA with adaptive control parameters for an NP-Hard scheduling problem, based on an existing Sequential GA.
- We show careful choice of pseudorandom number generator (PRNG) can accelerate GA runtime by up to 25% (sequential) and 20% (in parallel).
  - Genetic algorithm operators involve significant random behavior.
  - Rarely considered by GA implementers, replying simply on PRNGs in standard language libraries, which are slow relative to alternatives.

Usage of Random Numbers in GAs

- GAs are population based: Evolve a population over many generations.
- **Selection**: Selects the population for next generation.
  - Random selection process biased toward members with higher fitness.
- **Mutation**: Small random changes to population members.
  - For common bit-string representation, this involves iterating over all bits of all population members in each generation (1 random value per bit)
  - Other representations less extreme: Permutation mutation involves between 1 and 3 random values per population member per generation
- **Crossover**: Combine "genes" from pair of parents to form pair of children.
  - Involves generating random cross sites (i.e., random indices into bit-string, permutation, etc).

Background: Random Number Generation

- Pseudorandom Number Generators (PRNG) in standard libraries:
  - C, C#, and pre-2011 C++: linear congruential
  - C++11: linear congruential as well as Mersenne Twister
  - Java: linear congruential (Random class), SplitMix (ThreadLocalRandom since 1.7 and SplittableRandom since 1.8)
  - Python: Wichmann-Hill (prior to version 2.3), Mersenne Twister (version 2.3 onward)
- PRNG Characteristics:
  - Linear congruential (LCG): Slow, low quality randomness
  - Wichmann-Hill: combines multiple LCG, better quality, but still slow
  - Mersenne Twister: faster, and good quality (passes Dieharder tests)
  - SplitMix: much faster, and good quality (passes Dieharder tests)

More PRNG Background

- Some forms of evolutionary computation require generating random numbers from distributions other than uniform.
- Gaussian mutation for mutating real valued parameters in evolution strategies requires random values from a Gaussian distribution.
- Built-in language support for Gaussians:
  - C, C#, and pre-2011 C++: None in standard libraries.
  - C++11: polar method
  - Java: polar method (Random / ThreadLocalRandom classes, and none in SplittableRandom)
  - Python: ratio of uniform deviates method
- Other much faster algorithms available, such as Ziggurat

Sequential Genetic Algorithms

- Our 2 new Parallel GAs are based on 2 existing Sequential GAs.
- Shared Features of the GAs:
  – Permutation representation: Population of permutations.
  – Selection Operation: Stochastic Universal Sampling (SUS)
    - Probability of selecting a member of population proportional to its fitness (like weighted roulette wheel), but all at once (e.g., like spinning a wheel with N equidistant pointers).
    - Reduces selection bias (Baker, '87).
    - More efficient: One random number to select N population members, compared to N random numbers with weighted roulette wheel.
  – Elitism: We keep the *E* most-fit unique permutations unaltered.

Mutation and Crossover Operators

- Mutation Operator: Insertion
  - Remove random element of permutation and reinsert at different random location.
- Non-Wrapping Order Crossover (NWOX) [Cicirello, GECCO '06]

Step 1: Pick 2 random cross points, defining cross region (cr)

cr1

p1: [A B C | D E F | G H K L]

p2: [C F D | A B H | L E K G]

cr2

Step 2: Find p1-cr2 and p2-cr1

p1-cr2: [C D E F G K L]

p2-cr1: [C A B H L K G]

Step 3: Initialize each child with other parent's cross region

c1: [- - - | A B H | - - - -]

c2: [- - - | D E F | - - - -]

Step 4: Copy p1-cr2 into c1 beginning at left, jumping cross region, 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]

Sequential Genetic Algorithms

- GA #1: Static Control Parameters
  - GA control parameters tuned manually with a set of training problem instances [Cicirello, GECCO '06]
  - Parameters: PopSize=100, E=3, crossover rate C=0.95, mutation rate M=0.65
- GA #2: Adaptive Control Parameters
  - Each population member is a permutation with control parameters appended
  - $Pop[i] = < P[i], C[i], M[i], \sigma[i] >$
  - P[i]: permutation of the jobs
  - C[i]: Crossover rate: If *Pop[i], Pop[j]* are paired, then crossover occurs with either probability C[i] or C[j] (chosen randomly)
  - M[i]: Mutation rate: With probability M[i], P[i] is mutated once.

Sequential Genetic Algorithms

- GA #2: Adaptive Control Parameters
  - Parameter adaptation:
    - The C[i] and M[i] are initialized randomly in [0.1,1.0)
    - $\sigma[i]$: Initialized randomly in [0.05, 0.15)
    - In each generation, the non-elite members' parameters adapt via Gaussian Mutation as follows:
      - $C[i] = C[i] + N(0, \sigma[i])$
      - $M[i] = M[i] + N(0, \sigma[i])$
      - $\sigma[i] = \sigma[i] + N(0, 0.01)$
      - Where N() is a normally distributed random variable.

Parallel Genetic Algorithms (pGA)

- pGA #1: Static Control Parameters
  - Island model with *k* sub-populations (concurrent evolution of the *k* populations)
  - Operators: SUS selection with elitism, NWOX, Insertion mutation
  - Control parameters: fixed at manually tuned single population values
- pGA #2: Adaptive Control Parameters
  - Island model with *k* sub-populations (concurrent evolution of the *k* populations)
  - Operators: SUS selection with elitism, NWOX, Insertion mutation
  - Control parameters: encoded within population and evolve with search

Experiments

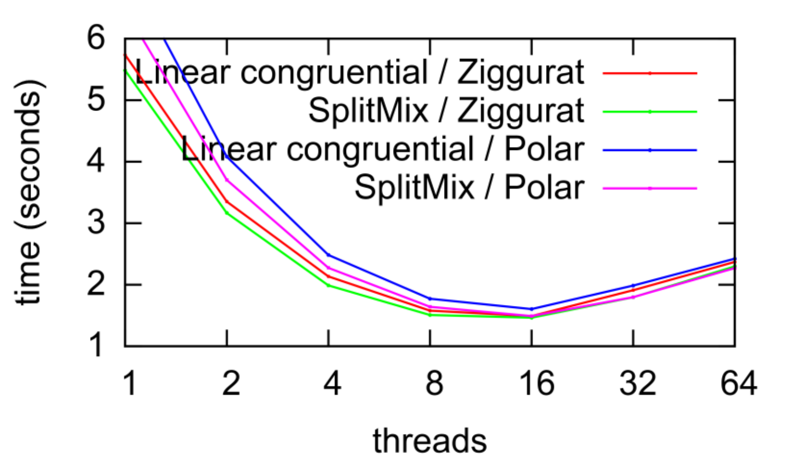
- **Platform:** Ubuntu 14.04 server, 2 Xeon L5520 quad-core (2.27GHz), 32GB; Java 8, Java HotSpot 64-bit Server VM
- **PRNGs used in experiments:**
  - Linear Congruential (Java's Random class)
  - SplitMix (Java's ThreadLocalRandom class, also SplittableRandom, but no time difference relative to ThreadLocalRandom)
- **Algorithms for Gaussian random numbers used in experiments:**
  - Polar method (included in standard library)
  - Ziggurat method [Marsaglia & Tsang, '00]: We ported the GNU Scientific Library's C implementation to Java.

Experiments

- **Problem:** Scheduling with sequence-dependent setups, minimizing weighted tardiness
  – NP-Hard
  – Used common benchmark set (120 problem instances, with varying levels of duedate tightness, setup time severity)
  – Best exact solver, dynamic programming, > 2 weeks CPU time solving hardest instances. (Tanaka & Araki, 2013)
  – Variety of algorithms applied to problem: neighborhood search (Liao et al, 2012), iterated local search (Xu et al 2014), ACO (Liao & Juan 2007), among others
  – Performance metric: Average % deviation from the optimals
    - Average over 10 runs on each of 98 problem instances (average of 980 runs).

pGA Runtime

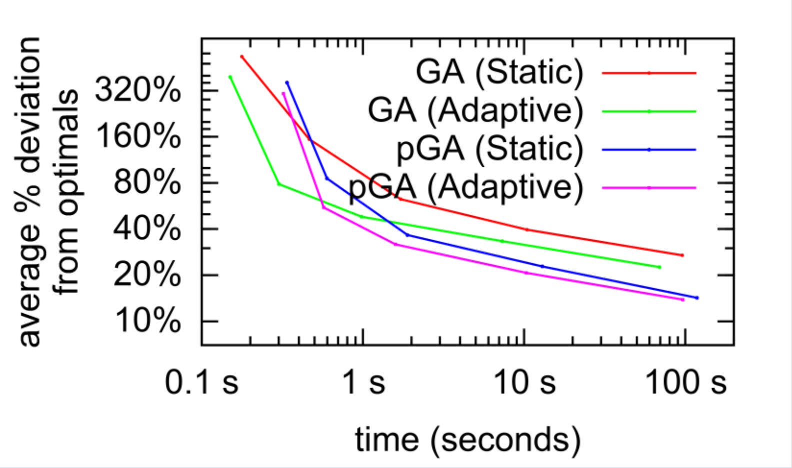
Adaptive parameters



At any fixed number of threads, no statistical difference (Wilcoxon signed rank test) on the scheduling optimization objective across the benchmark.

- Experimental parameters
  - Subpopulation size = 100,
  - Total population size = 100 k, where k is number of threads
  - Generations = 64000 / k
- pGA with adaptive parameters (4 threads) is 20% faster with SplitMix & Ziggurat than with LC & Polar.
- Sequential GA with adaptive parameters is 25% faster with SplitMix & Ziggurat than with LC & Polar.

Scheduling performance



- Sequential GA settings:
  - Population size = 100 (as was originally the case)
- pGA settings:
  - Subpopulation size = 100
  - 8 subpopulations (8 threads)
  - Total population size = 800
- Short runs: no benefit to parallelization (due to thread management overhead)
- In 8.8s, the adaptive pGA achieves an average percent deviation equivalent to that of a 69s run of the sequential GA.

pGA: Adaptive vs Static Parameters

	Average % deviation From optimals			Runtime in seconds	
G	**Adaptive**	**Static**	**p**	**Adaptive**	**Static**
10^2	306.0%	360.7%	$< 10^{-8}$	0.3s	0.3s
10^3	55.4%	85.6%	$< 10^{-8}$	0.6s	0.6s
10^4	31.8%	36.6%	$< 10^{-8}$	1.6s	1.9s
10^5	20.7%	22.8%	0.003	10s	13s
10^6	13.9%	14.2%	0.043	96s	118s

G = number of generations
p is p-value from Wilcoxon signed rank test

- The pGA with adaptive parameters leads to better quality solutions than static parameters for any fixed length run in #generations.
- The statistical significance of differences decreases with run length
  - The longer the run, the nearer to the optimals both versions become
- The pGA with adaptive parameters requires less time than with static parameters for the same number of generations
  - Crossover rate tends to decline later in the run with adaptive parameters

Conclusions

- We introduced a new pGA for an NP-Hard scheduling problem
  - Only parallel GA for this scheduling problem (several sequential GAs exist).
  - Multipopulation pGA with adaptive control parameters
  - The pGA achieves approximately linear speedup (for 8 threads) relative to its sequential counterpart
    - Note: Our test system has 8 physical cores, so unknown if linear speedup continues beyond 8 threads.
- Showed choice of pseudorandom number generator, and associated algorithms, has potential to drastically affect GA and pGA runtime
  - Often overlooked implementation decision, commonly opting for language built-in
  - Can accelerate runtime by 25% for a sequential GA, and 20% for a parallel GA
  - Similar results may be found for other metaheuristics that rely on randomness

Questions