Operon C++

An Efficient Genetic Programming Framework for Symbolic Regression

Bogdan Burlacu[1,2,3], Gabriel Kronberger[1,2,3], Michael Kommenda[1,2,3]

[1] University of Applied Sciences Upper Austria
[2] Heuristic and Evolutionary Algorithms Laboratory
[3] Josef Ressel Centre for Symbolic Regression

Motivation

Genetic Programming is a dynamic field of research, where **prototyping** and **empirical testing** play an important role. As **parallel hardware** becomes increasingly more prevalent, **modern concurrent** implementations with a focus on **efficient resource usage**, **performance** and **scalability** are needed. **Convention over configuration** should be preferred to provide an **out-of-box experience** for the user.

Concurrency

- Logical threads, fine-grained at the level of the individual
- Low-overhead synchronization

Efficiency

- Linear tree representation (postfix scheme)
- Tree node: *plain old type*, standard memory layout (contiguous)
- Favor modern hardware (cache locality, deep pipelines, branch prediction, data-level parallelism)
- Minimize heap allocations

Design

- Novel tree initialization algorithm – Balanced Tree Creator (BTC)
- Offspring generator operator defines how offspring are produced
- Fitness evaluation extends to dual number domain (free autodiff)
- Hybridization with local search (non-linear least squares).

Evolutionary model

Offspring generator

- Defines strategy for generating a new child individual
- Encapsulates algorithm termination logic
- Encapsulates other operators (selection, evaluation, crossover, mutation)

Parent selection

- Selection mechanism can be configured independently for each parent (e.g., proportional + random)
- Tournament, proportional, random selection

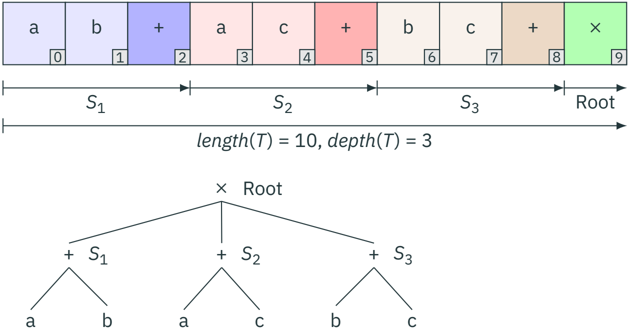
Termination criteria

- Generation limit
- Fitness evaluations limit

Encoding

Postfix representation

`((a b +) (a c +) (b c +) ⋆)`



Tree initialization

Balanced Tree Creator (BTC)[1]

Creates an expression of specified length.

1. Start with a random root node. Keep track of a horizon of expansion points (unfilled child slots, if any).
2. Fill slots in breadth-first fashion keeping track of remaining length.
    - Arity of new symbols is limited according to remaining length difference.
    - Each function node opens a number of expansion points equal to its arity.
    - If target length is reached, remaining expansion points filled with leafs.

A bias parameter used to control tree shape variability.

Since trees are compact, no depth limit is required.

Complexity $O(n)$ where n is the number of nodes.

[1] `https://github.com/foolnotion/operon/blob/master/src/operators/creator/balanced.cpp`

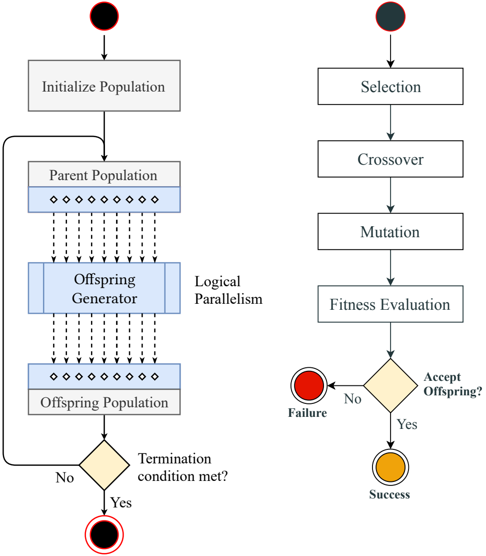
Concurrency model

Logical parallelism

- Child population generated concurrently
- Logical threads can be seen as jobs representing domain-specific work, typically organized into work queues
- Work queues may be executed in interleaved fashion on a single physical thread (depending on available machine resources)

Offspring generator

- Modeled as a *Maybe* monad (return a new offspring or *nothing*), using a `std::optional` return type
- Success or Failure depending on postconditions on offspring
- Signals algorithm termination accross threads



Logical parallelism

- Minimal synchronization overhead (atomic types used).
- Each attempt to create an offspring in a separate *logical thread*, not sharing any mutable state with other threads.
- Assignment of logical tasks to physical threads is left to the underlying scheduler.
- Backed by *Intel® Threading Building Blocks* (TBB) library.

Deterministic execution

- Impossible to guarantee for all platforms and `stdlib` or `libm` implementations.
- Locally reproducible results via fixed seed.
- Local pre-seeded `rng` instance for every local thread.

Fitness evaluation

- Heavy lifting done by the *Eigen* library[2].
- Local search step for optimal model parameters via non-linear least squares, provided by the *Ceres* library[3].
- Seamless integration with *Jets* (dual numbers) for tree expression autodiff (via C++ templates).
- Single- or double-precision tree evaluation, data batching to reduce impact of tree interpreter control-flow.
- No need for explicit vectorization or hand-written SIMD intrinsics.

[2] http://eigen.tuxfamily.org
[3] http://ceres-solver.org/

Performance analysis

Test system

- AMD Ryzen™ 3900X processor, 12 core/24 thread, 3.8Ghz base frequency, 768Kb/6Mb/64Mb L1/L2/L3 cache.
- 32Gb DDR4-3600 CL16 memory.

Framework comparison

- DEAP (Python)[4]
- HeuristicLab (C#)[5]

Methods

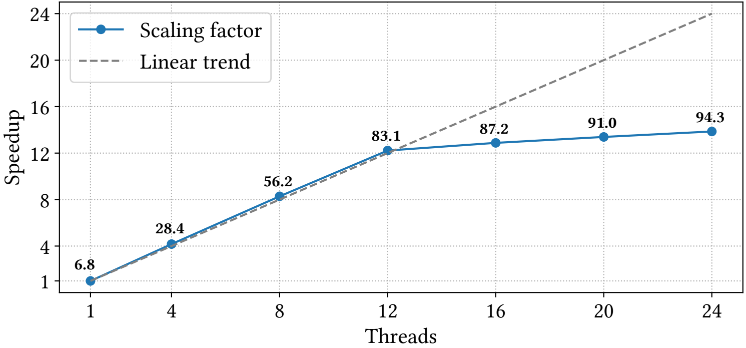
- CPU and memory profiling
- Empirical testing

[4] https://deap.readthedocs.io
[5] https://dev.heuristiclab.com

Evaluation speed (double-precision)

Test configuration

- 1000 trees of average length 50
- Arithmetic primitive set $P = (+, -, \times, \div)$
- 5000 data rows



- Datapoint labels represent evaluation speed in billion GPops/second
- Inflection point at 12 threads (physical core limit)
- Linear scaling with physical threads

Profiling analysis

Measurements

- Relative operator overhead
- Memory consumption: resident set size (RSS), heap allocated memory[6].

Test configuration

- Standard GP, 1000 individuals, 100 generations, max tree length 50
- Primitive set: $(+, -, \times, \div, \exp, \log, \sin, \cos)$.
- Varying number of rows.

Tools

- **Operon**: `valgrind`, `massif`
- **DEAP**: `cProfile`, `pyprof2calltree`, `psutil`
- **HeuristicLab**: Visual Studio profiler, `Process.PeakWorkingSet64`

[6] Applicable to *Operon* only as its allocations can be explicitly tracked.

Genetic operator relative impact and memory consumption

Operon

- Fitness evaluation in double-precision
- Very low memory overhead

DEAP

- Trees are compiled into lambda functions
- Numpy backend for evaluation
- Significant overhead for small data

HeuristicLab

- Evaluation calls into native C++ dll
- Actual fitness (e.g. R^2, MSE) computed on the managed (C#) side
- Runtime overhead due to operator graph execution model

Rows	Fitness	Crossover	Mutation	Selection	Heap (Mb)	RSS (Mb)
Operon						
50	86.80%	10.71%	0.61%	0.96%	3.7	10.3
100	92.50%	6.10%	0.35%	0.54%	3.7	10.3
500	97.52%	2.01%	0.11%	0.18%	3.7	10.4
1000	98.81%	0.97%	0.05%	0.09%	3.8	10.5
2000	99.43%	0.47%	0.03%	0.04%	3.9	10.7
5000	99.73%	0.22%	0.01%	0.02%	4.4	11.6
DEAP						
50	67.22% (47.38%)	1.78%	1.53%	2.26%	-	101.0
100	68.02% (47.36%)	1.70%	1.55%	2.18%	-	101.1
500	69.84% (42.52%)	1.61%	1.50%	2.07%	-	101.3
1000	71.47% (38.53%)	1.55%	1.37%	1.99%	-	101.3
2000	75.96% (30.75%)	1.30%	1.19%	1.66%	-	101.5
5000	80.90% (22.74%)	1.04%	0.95%	1.34%	-	102.2
HeuristicLab						
50	12.47%	18.62%	1.02%	10.69%	-	646.5
100	14.77%	18.26%	1.00%	10.32%	-	647.5
500	30.48%	13.55%	0.76%	7.60%	-	673.0
1000	44.29%	10.81%	0.57%	5.73%	-	685.3
2000	53.36%	7.28%	0.39%	3.93%	-	741.6
5000	68.76%	4.12%	0.22%	2.14%	-	761.6

GP Framework comparison

9 test problems, 50 repetitions for each configuration

Algorithm parameterization

Function set	$+, -, \times, \div, \sin, \cos, \exp, \log$
Terminal set	constant, weight · variable
Tree limits	10 levels, 50 nodes
Tree initialization	Balanced tree creator
Population size	1000 individuals
Generations	1000 generations
Parent selection	Tournament group size 5
Crossover probability	100%
Crossover operator	Subtree crossover
Mutation probability	25%
Mutation operator	Single-point mutation
Fitness function	R^2 correlation with the target

Parallelization scheme

- More advantageous parallelization scheme used for each framework
- **DEAP**, **HeuristicLab**: coarse-grained parallelization (run multiple instances in parallel)

Adjusted elapsed time

Individual run times t_i adjusted using mean parallel execution time $\overline{T}$.

$$\overline{T} = \frac{\mathrm{T}}{N} \quad \text{(mean parallel run time)}$$

$$\bar{t} = \frac{1}{N}\sum_i t_i \quad \text{(mean run time)}$$

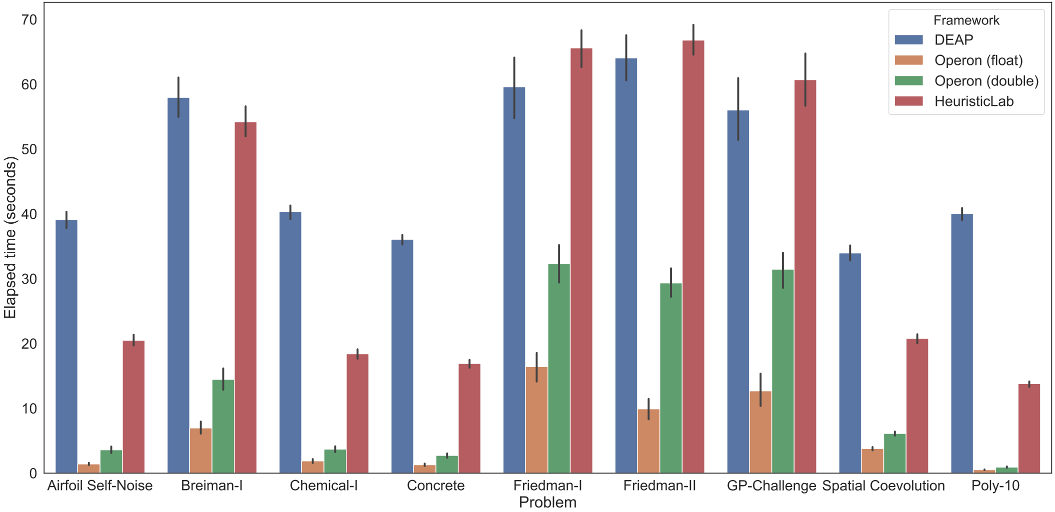
$$t_i^* = \frac{t_i}{\bar{t}} \cdot \overline{T} \quad \text{(adjusted run time)}$$

Empirical results

Framework	NMSE (train)	NMSE (test)	Elapsed (s)
Airfoil Self-Noise (1000 training rows)			
Deap	0.294 ± 0.068	0.335 ± 0.107	38.6 ± 4.6
HeuristicLab	0.219 ± 0.035	0.256 ± 0.058	20.5 ± 3.0
Operon (double)	0.242 ± 0.061	0.270 ± 0.084	3.5 ± 3.1
Operon (float)	0.233 ± 0.047	0.262 ± 0.062	1.3 ± 0.5
Breiman-I (5000 training rows)			
Deap	0.114 ± 0.015	0.122 ± 0.015	59.1 ± 15.9
HeuristicLab	0.115 ± 0.046	0.122 ± 0.042	52.1 ± 15.1
Operon (double)	0.106 ± 0.019	0.112 ± 0.020	12.8 ± 8.5
Operon (float)	0.108 ± 0.013	0.115 ± 0.011	5.4 ± 3.4
Chemical-I (711 training rows)			
Deap	0.228 ± 0.024	0.368 ± 0.176	40.0 ± 2.9
HeuristicLab	0.204 ± 0.026	0.324 ± 0.171	18.7 ± 2.4
Operon (double)	0.194 ± 0.025	0.284 ± 0.140	4.2 ± 2.5
Operon (float)	0.194 ± 0.026	0.320 ± 0.189	1.5 ± 1.7
Concrete Compressive Strength (1000 training rows)			
Deap	0.161 ± 0.014	0.576 ± 0.138	36.2 ± 1.6
HeuristicLab	0.151 ± 0.017	0.595 ± 0.191	17.2 ± 1.9
Operon (double)	0.152 ± 0.022	0.630 ± 0.215	2.7 ± 1.2
Operon (float)	0.148 ± 0.014	0.574 ± 0.162	1.0 ± 1.2

Framework	NMSE (train)	NMSE (test)	Elapsed (s)
Friedman-I (5000 training rows)			
Deap	0.165 ± 0.040	0.158 ± 0.031	55.2 ± 24.7
HeuristicLab	0.138 ± 0.006	0.139 ± 0.005	64.4 ± 13.9
Operon (double)	0.138 ± 0.003	0.139 ± 0.004	31.2 ± 13.7
Operon (float)	0.139 ± 0.006	0.139 ± 0.006	16.6 ± 10.9
Friedman-II (5000 training rows)			
Deap	0.126 ± 0.058	0.129 ± 0.059	62.5 ± 15.5
HeuristicLab	0.041 ± 0.020	0.042 ± 0.022	67.7 ± 11.9
Operon (double)	0.041 ± 0.009	0.042 ± 0.010	29.4 ± 12.4
Operon (float)	0.043 ± 0.015	0.044 ± 0.016	7.7 ± 9.2
GP-Challenge (5000 training rows)			
Deap	0.097 ± 0.013	0.098 ± 0.015	56.2 ± 25.0
HeuristicLab	0.079 ± 0.013	0.079 ± 0.016	60.4 ± 21.6
Operon (double)	0.074 ± 0.010	0.073 ± 0.011	34.5 ± 15.8
Operon (float)	0.076 ± 0.012	0.075 ± 0.012	8.7 ± 14.0
Poly-10 (250 training rows)			
Deap	0.140 ± 0.124	0.182 ± 0.217	40.5 ± 1.8
HeuristicLab	0.170 ± 0.296	0.193 ± 0.403	14.1 ± 0.8
Operon (double)	0.076 ± 0.134	0.089 ± 0.177	0.9 ± 0.5
Operon (float)	0.078 ± 0.138	0.088 ± 0.172	0.5 ± 0.2
Spatial Coevolution (676 training rows)			
Deap	0.003 ± 0.012	0.146 ± 0.258	34.0 ± 6.0
HeuristicLab	0.003 ± 0.005	0.024 ± 0.126	21.3 ± 1.6
Operon (double)	0.001 ± 0.002	0.008 ± 0.032	6.1 ± 1.2
Operon (float)	0.002 ± 0.001	0.005 ± 0.011	4.0 ± 1.4

Execution speed



Conclusion

The Operon framework

- New tree initialization algorithm - Balanced Tree Creator
- Significant savings in execution time and resource usage can be gained by careful design.
- Scalable concurrency model enables fine-grained parallelization of GP experiments.
- Memory efficiency enables large-scale experiments
- Flexibility in defining different GP evolutionary models via the offspring generator concept.
- Lacks flexibility in defining new types of problems or primitive types.

Future roadmap

- Test the limits of the framework's scalability
- Implement more algorithms
- Serialization support
- Python wrapper

Project page

https://github.com/foolnotion/operon

Results, code and data

https://dev.heuristiclab.com/trac.fcgi/wiki/AdditionalMaterial#GECCO2020

THANK YOU!