Parallel Program Analysis For Linear Genetic Programming

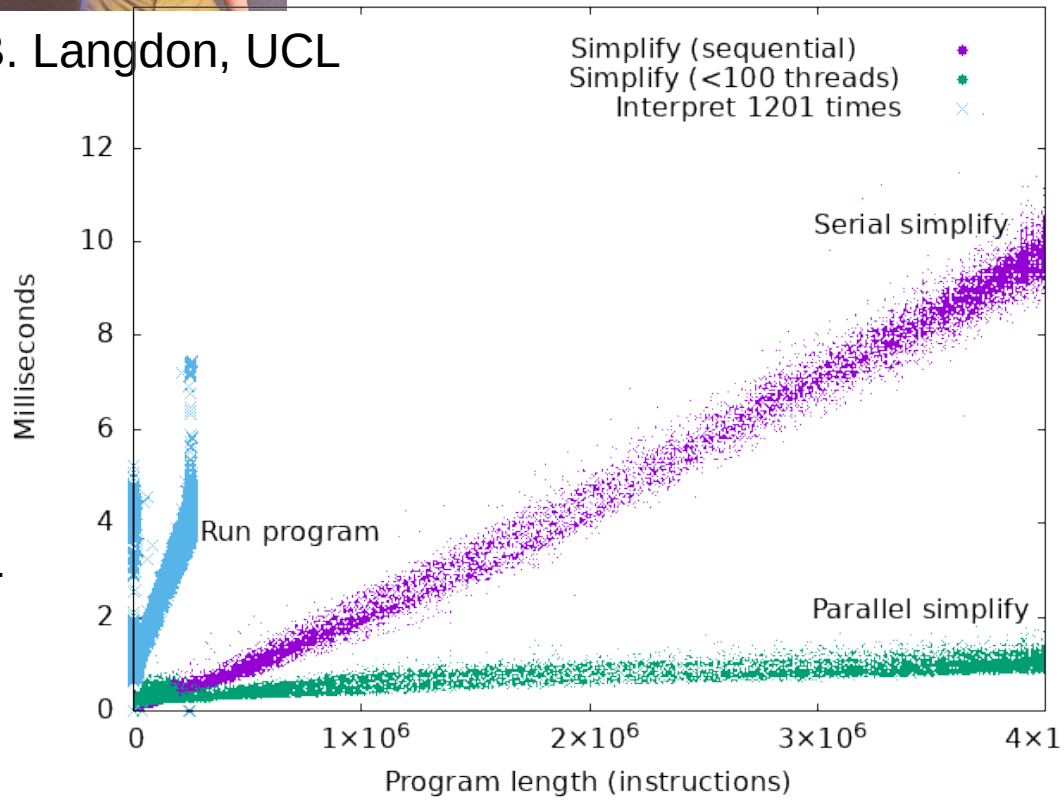
25th UK Workshop on Computational Intelligence (UKCI 2026)

Computing hardware is parallel

Sequential program analysis can be parallel.

https://github.com/wblangdon/GPengine

W. B. Langdon, UCL



William B. Langdon. Long Term Evolution Experiments with Linear Genetic Programming. In Wolfgang Banzhaf and Ting Hu editors, Advances in Linear Genetic Programming, chapter 4. Springer, 2026.

Parallel Program Analysis For Linear Genetic Programming

- Parallel computing
- Long Term Evolution Experiments
- How
  - Split simplification between CPU threads
  - Pool of ready pthreads
  - Powerset of possible results from other threads
- Ten fold speedup

Computing is Parallel

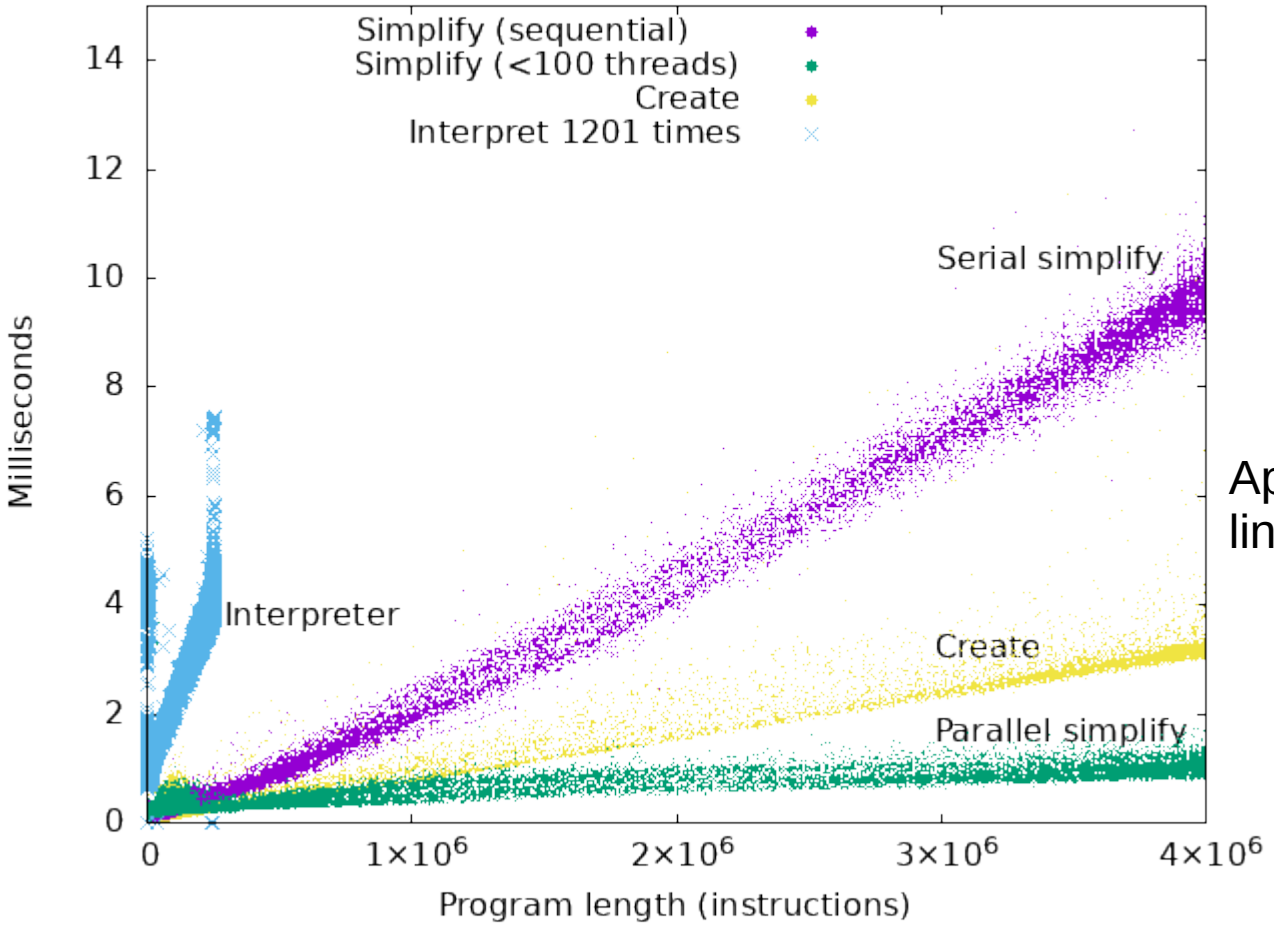
- Many computers are multi-core (256 core 64 bit AMD EPYC 9554)
- Today
  - most computers support 64 bits
  - Intel AVX up to 512 bit vector instructions
  - nvidia 32 x 32 = 1024 bit warps
  - ARM SVE up to 2048 bit vectors
- Expect no increase in clock speed
- Expect increasing numbers of cores

Long Term Evolution Experiments

- Rick Lenski's evolution of bacteria in stable environment
- Told we know what will happen; do not do experiment
- Totally wrong. E-Coli continued to evolve. Still evolving 80,000 generations and 38 years later.
- Have run genetic programming to a million generations
- Need for speed
- Time to simplify bloated redundant programs a major cost
- Simplification is sequential backward slice to remove all instructions which cannot impact output

Cost of simplify bloated redundant programs

- Before: simplification ≈10ms, creation ≈3ms, fitness ≈4ms
  Total ≈17ms
- After: simplification ≈1ms. Total ≈8ms



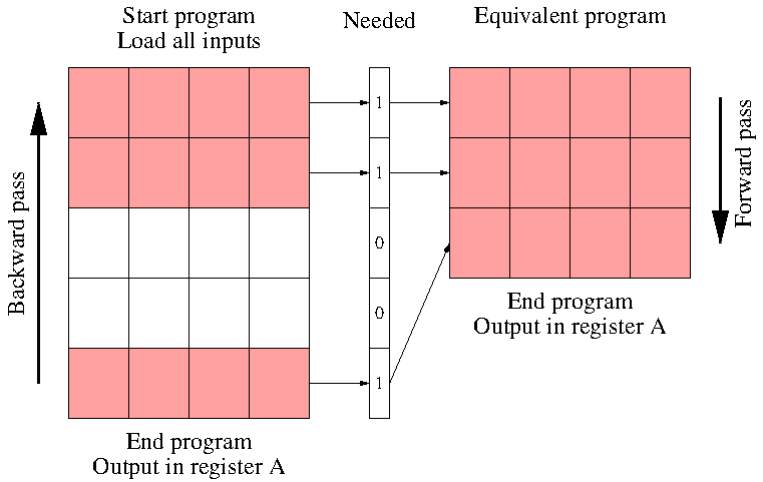
Approx
linear increases

Sequential Program Analysis

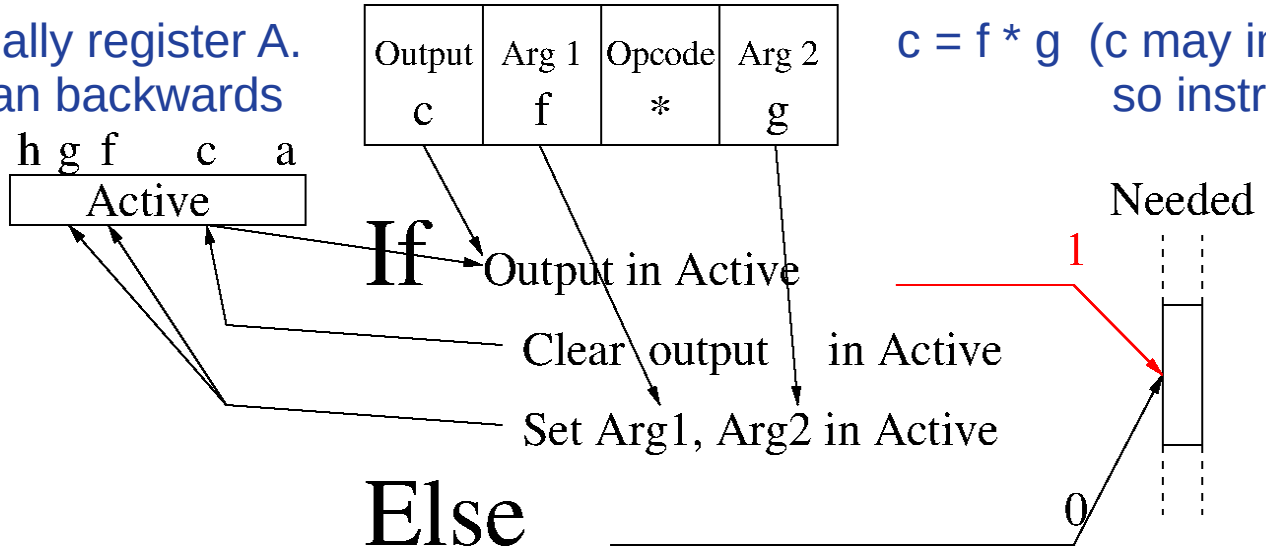
- Linear Genetic Programming sequence of register instructions,

  no ifs, no loops.
- Inputs loaded into 8 registers at start, program run start to end, answer extracted from results register A.
- Many instructions do not impact register A. Simplification can reduce program by 3 to 700 fold. Giving 3 to 700 speed up (median 15.7)
- Start at program end
- Find previous instruction which sets A's value, record its location
- Note which registers that instruction depends on
- Find previous instructions which impact them
- Maintain set of active registers as work back to start of program
- Remove all instructions except those noted as having an impact
- Fast (only two passes) but need not be optimal (e.g. does not consider data values)

Sequential Program Analysis



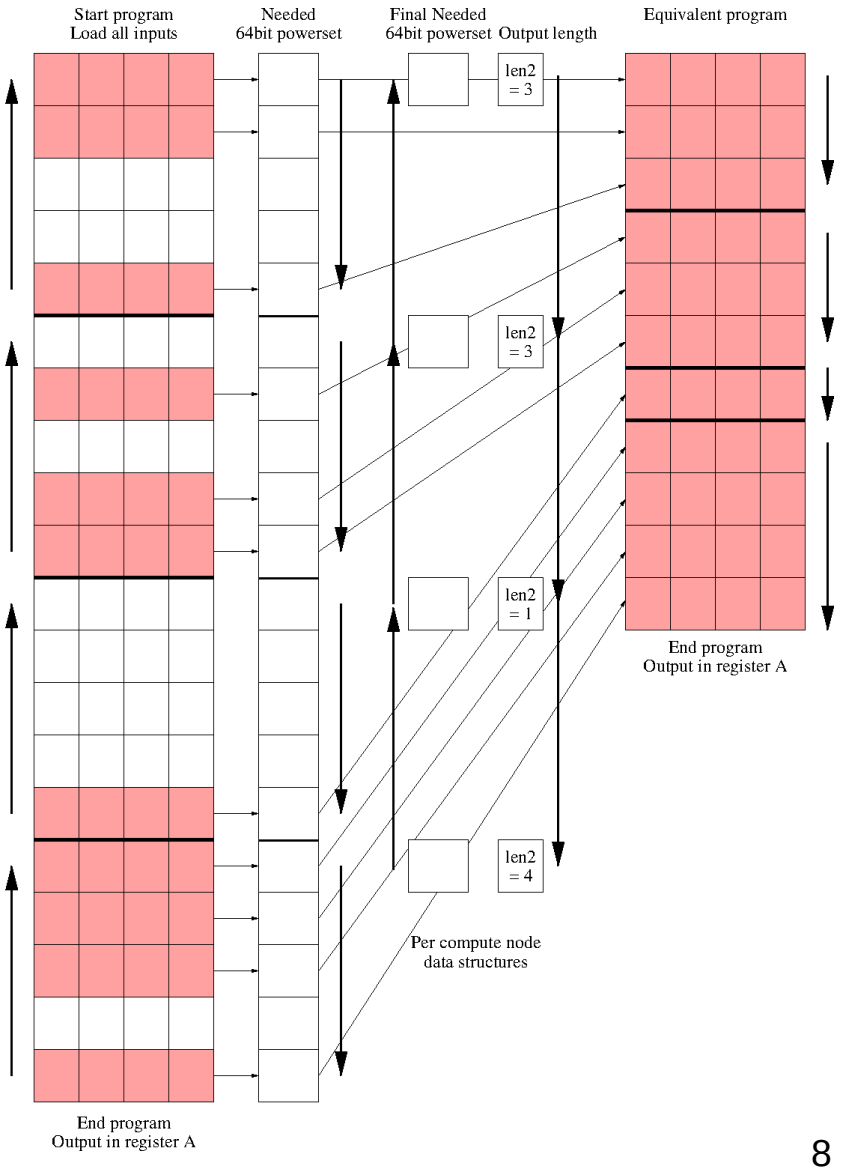
Active set initially register A.
Update as scan backwards



c = f * g (c may impact output, so instruction needed)

Parallel Program Analysis

- Split program to be analysed between threads
- Start at end of each fragment assume any combination of 8 registers may matter, so use powerset and work backwards
- At each instruction update and save powerset
- Make last powerset available to other threads
- Wait for other threads
- Start from program end (where only register A matters) use backward pass of others' powerset to determine which registers matter to this thread
- And so count number of needed instructions. Save in shared array len2 for other threads
- Wait for other threads
- From program start sum len2 to find where in output to copy own needed instructions
- Using powerset again, copy own needed instructions



Posix pthreads & vector instructions

- Posix thread is program code typically run on another CPU core
- Thread code has access to program variables which are in scope
- Easy use: create n threads, give them work, wait for them to finish
- Overhead of creating n subprocesses may be important if create 100s of threads
- At program start create pool of n threads, make sure they wait. When main thread is ready, wake up as many threads as needed. When threads finish they tell main program and go back to waiting for work. Thread pool reused many times.
- pthread_mutex_lock, pthread_cond_wait, pthread_cond_broadcast
- Vector instructions allow single instruction to act on multiple data

  E.g. process eight bit masks (one for each register) simultaneously

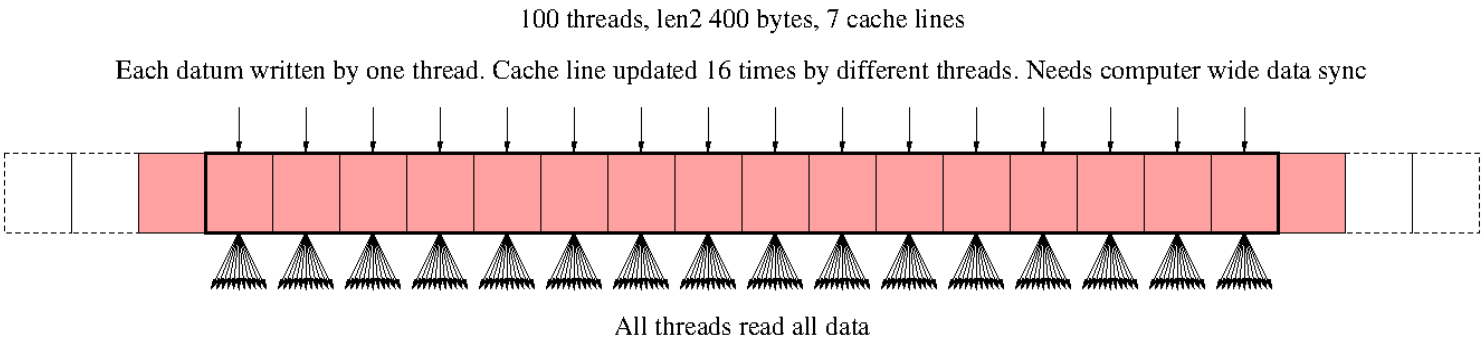
Memory caches

- Multi-core computers typically provide three levels of cache.
- Caches are in the hardware and maintain the fiction of one memory
  - L1 per core small operates at speed of CPU
  - L2 per core intermediate size and slower
  - L3 shared by all cores.

    Can be slow, especially if computer is composed of multiple silicon chips

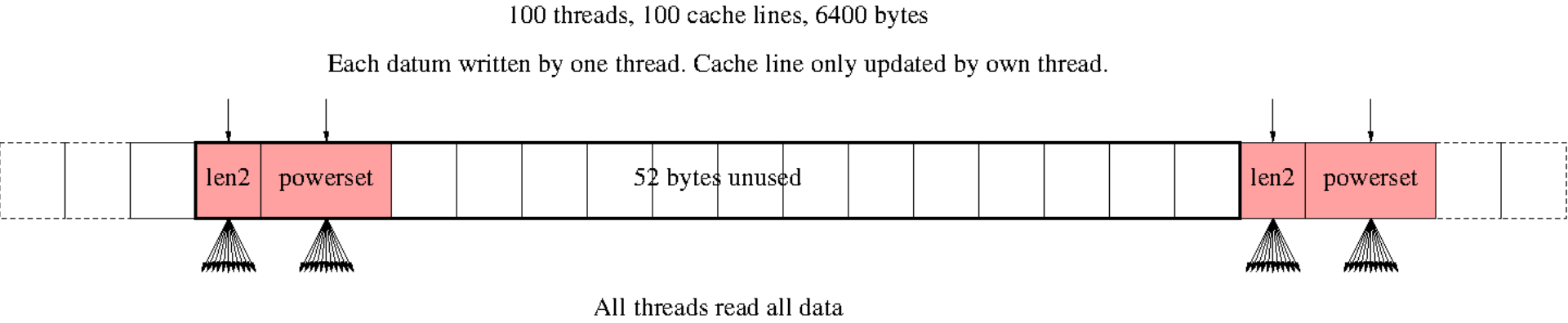
    L3 is used to pass syncronisation between cores.
- Keep all analysis data in L2 cache. As programs increase in size, increase number of cores (threads) so have more L2
- Basic memory unit is cache line (Intel x86) 64 bytes
- Presently minimise memory but may suffer from “false sharing” i.e. L3 used for synchronization (can be very slow).

100 threads tightly packed len2 7 cache lines



Similar picture for 64bit powerset, 100 threads needs 13 cache lines

100 threads loosely pack per thread data

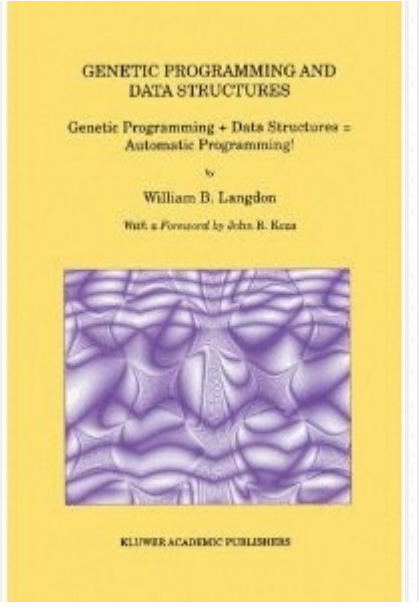


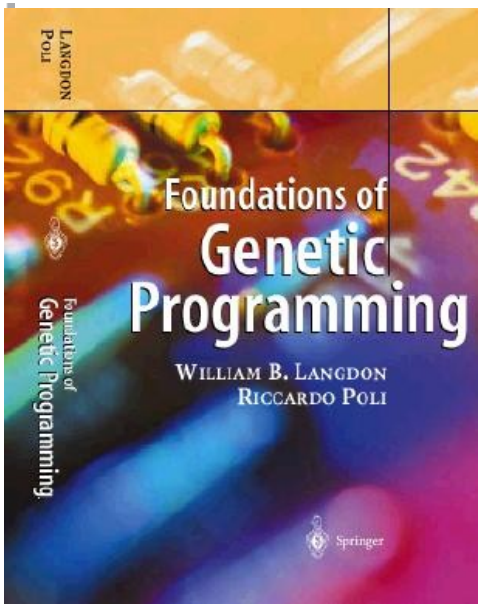
Conclusions

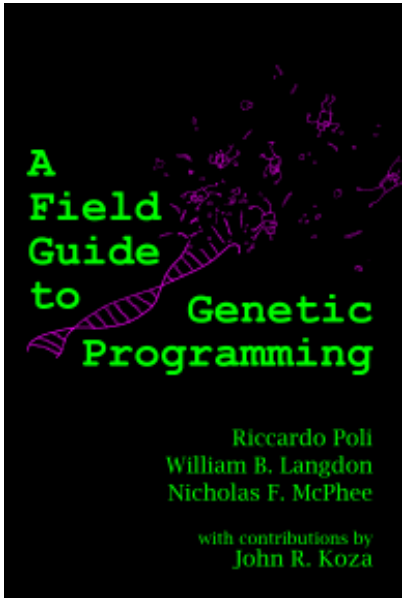
- Parallel approach removes a compute heavy bottleneck in large scale linear genetic programming
  - Effective speeds of up to 726 billion GP operations/second (effectively interpret 235 operation per clock tick)
- Future is parallel.
- Pool of pthreads reduces overhead
- Wide parallel vector instructions allow efficient powersets
- Backward slicing is usually sequential but can be
  - broken into multiple parallel operations made independent
  - allowing any outcome of earlier processing
  - vector instructions allow processing all possibilities with modest overhead
- For large programs, parallel dead code removal 10x faster
- For impact do the impossible

Genetic Programming

W. B. Langdon







Why parallel may be different

- Sequential simplification is conservative
  - Pragmatic: trade simplicity with performance of residual code
  - It gives an equivalent program which need not be the best
- Break sequential algorithm at arbitrary points.
  - Each fragment is pragmatically simplified
  - When reconnected we get an equivalent program which is often (99.9935%) but not guaranteed to be the same as the output of the serial algorithm.

Is Mackey-Glass typical?

https://github.com/wblangdon/GPengine

- Mackey-Glass is an IEEE benchmark often used as an example of a chaotic and hence difficult to predict time series.
- As expected, without counter measures, linear genetic programming bloats when predicting Mackey-Glass, giving rise (as is often the case) to huge highly repetitive programs which can be automatically simplified.
- With linear genetic programming Mackey-Glass uses 8 registers, typical linear GP uses only 3 or 4.
  - I.e. Mackey-Glass simplification is harder than usual
- Mackey-Glass has 1201 examples, typical stock market prediction may use as little as 60 days for training data.
- Automatic simplification is often used in linear genetic programming

The Genetic Programming Bibliography

http://gpbib.cs.ucl.ac.uk/

18704 references, 19000 authors

Make sure it has all of your papers!

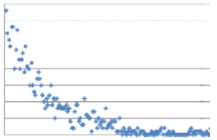
E.g. email W.Langdon@cs.ucl.ac.uk or use | Add to It | web link

Co-authorship community.
Downloads

Downloads by day

A personalised list of every author's GP publications.

blog



Your papers

Text search

Googling GP bibliography, eg:
Development and learning site:gpbib.cs.ucl.ac.uk