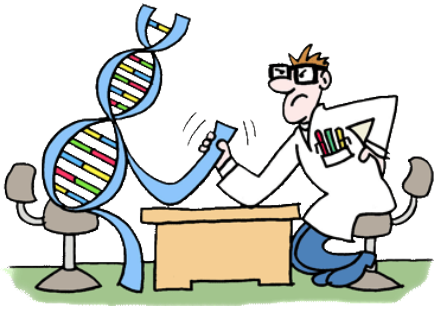
CREST

Parallel Program Analysis For Linear Genetic Programming

Recording

King's College, London, 10 November 2026, W. B. Langdon, UCL



$10000 "Humies" 28 May 2027

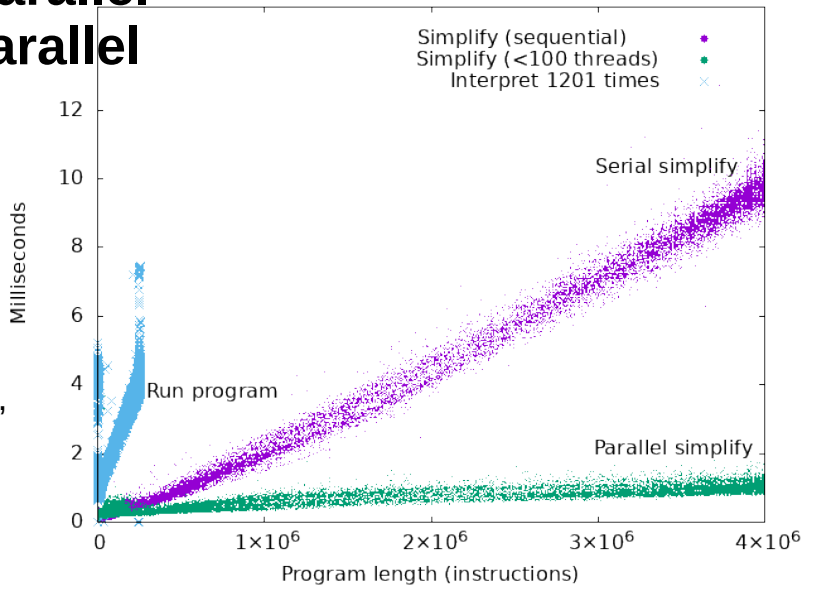


W. B. Langdon, UCL



"When I was your age I could think of six impossible things before breakfast"

Computers are parallel
Make analysis parallel



Parallel Program Analysis for Linear Genetic Programming, William B. Langdon, 25th UK Workshop on Computational Intelligence (UKCI 2026), 9-11 Sep, Springer

Long Term Evolution Experiments with Linear Genetic Programming, in Advances in Linear Genetic Programming, chapter 4. Springer

Parallel Program Analysis For Linear Genetic Programming

- Parallel computing, Moore's Law (1965)
- Why: Long Term Evolution Experiments
- How to speed up backward slice analysis
  - 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
  - Typical 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

Moore's Law: Exponential increase



Gordon Moore 1929-2023

Silicon MOS transistor 1959

Silicon transistor 1954

Transistor 1947

Ambrose Fleming, UCL thermionic valve, 1904

Electronics, Volume 38, Number 8, April 19, 1965

Moore’s Law

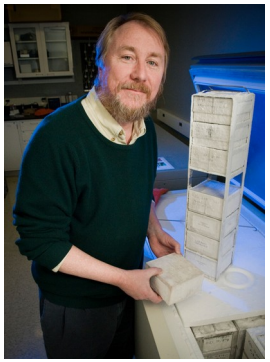


Gordon Moore 1929-2023

2026 Nvidia 336 billion transistors

2022 AMD EPYC 9554, 40 billion transistors

Long Term Evolution Experiment LTEE in Biology



- Rich Lenski's Long Term Evolution Experiment LTEE 83,000 generations of bacteria E.coli shows **continued evolution**. (Homo Sapiens 9300 generations old)
  - Experiment running since 1988 (38 years)
    - 1988 University of California, Irvine
    - 1991 Michigan State University
    - 2022 University of Texas at Austin
    - 2025 return to Michigan State University. 83,000+ generations
- What happens in computer based evolution?

  ≤ 1 million generations
  Long-Term Evolution Experiment with Genetic Programming
  Artificial Life, 2022 28(2) pp 173-204

- Run with Linear Genetic Programming

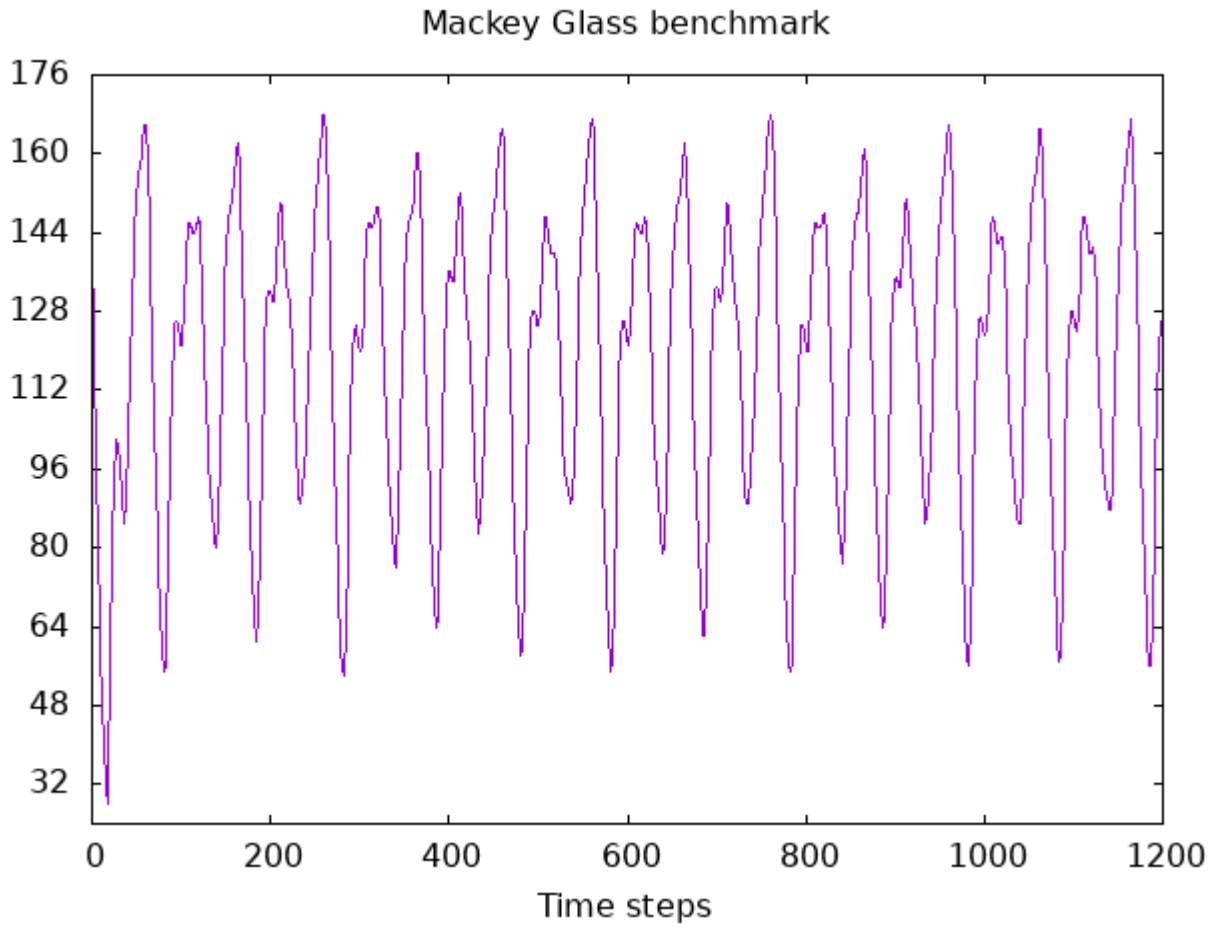
  Long Term Evolution Experiments with Linear Genetic Programming.
  In Recent Advances in Linear Genetic Programming. Springer.

Long Term Evolution Linear Genetic Programming

- Peter Nordin's GPengine https://github.com/wblangdon/GPengine
- 500 random initial programs 1-14 instructions
- Evaluate programs on 1201 Mackey-Glass test cases
- Fitness RMS error predicting next timestep
- Steady state overlapping generations
- Tournament selection
  - Four randomly selected. Best 2 parents ⇛ 2 children. Children replace losers
- 90% crossover and mutation, 10% fitness based clone
- Eight short integers (bytes) registers
- Four arithmetic: + - * protected division (y!=0)? x/y : 0
- 100,000 generations, ≤ 4 million instructions
- GI 2026 optimise AVX parallel fitness evaluation

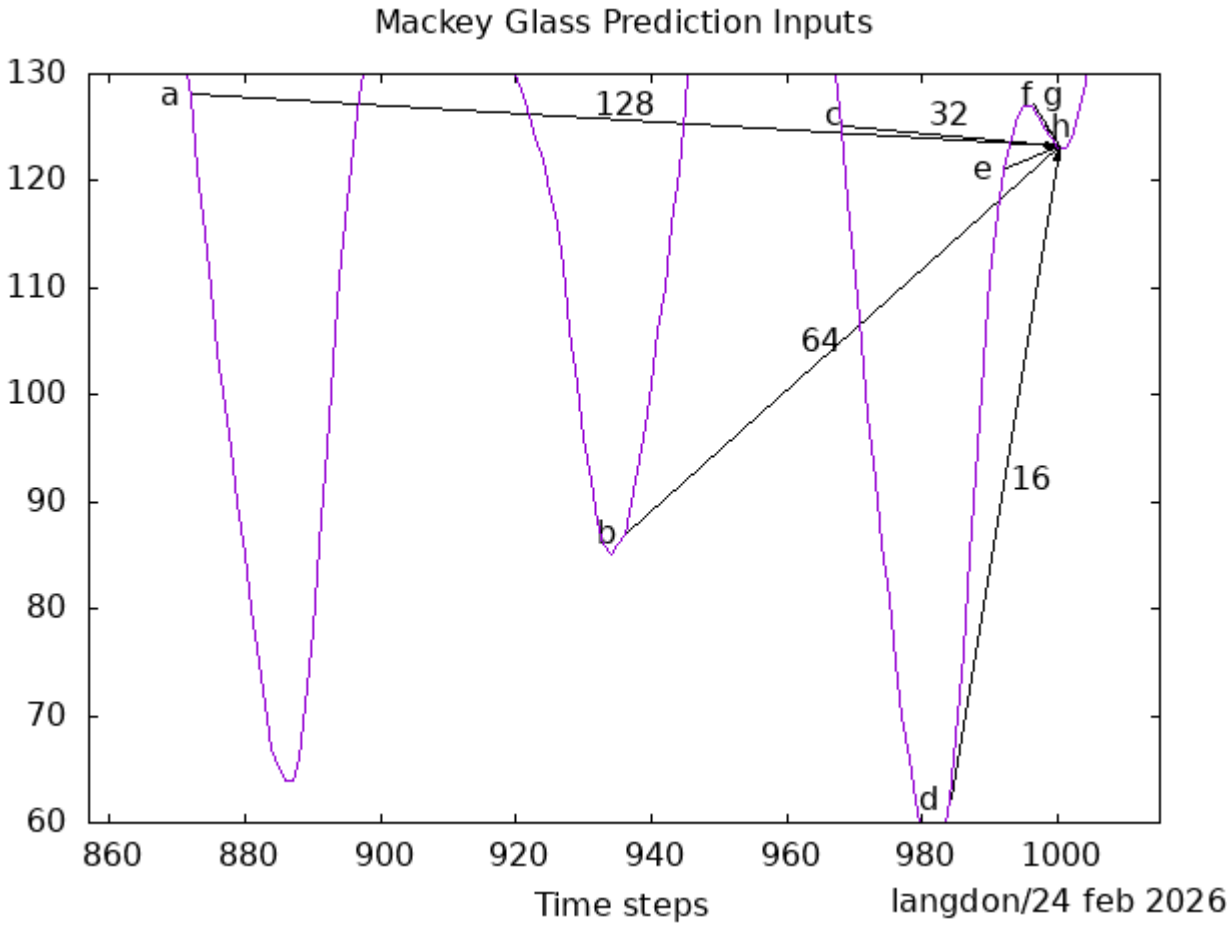
Predict Chaotic Time Sequence

IEEE Mackey-Glass prediction benchmark sequence is chaotic,
so inherently not predictable.
Nonetheless, using earlier values, it can be approximated.



https://github.com/wblangdon/GPengine

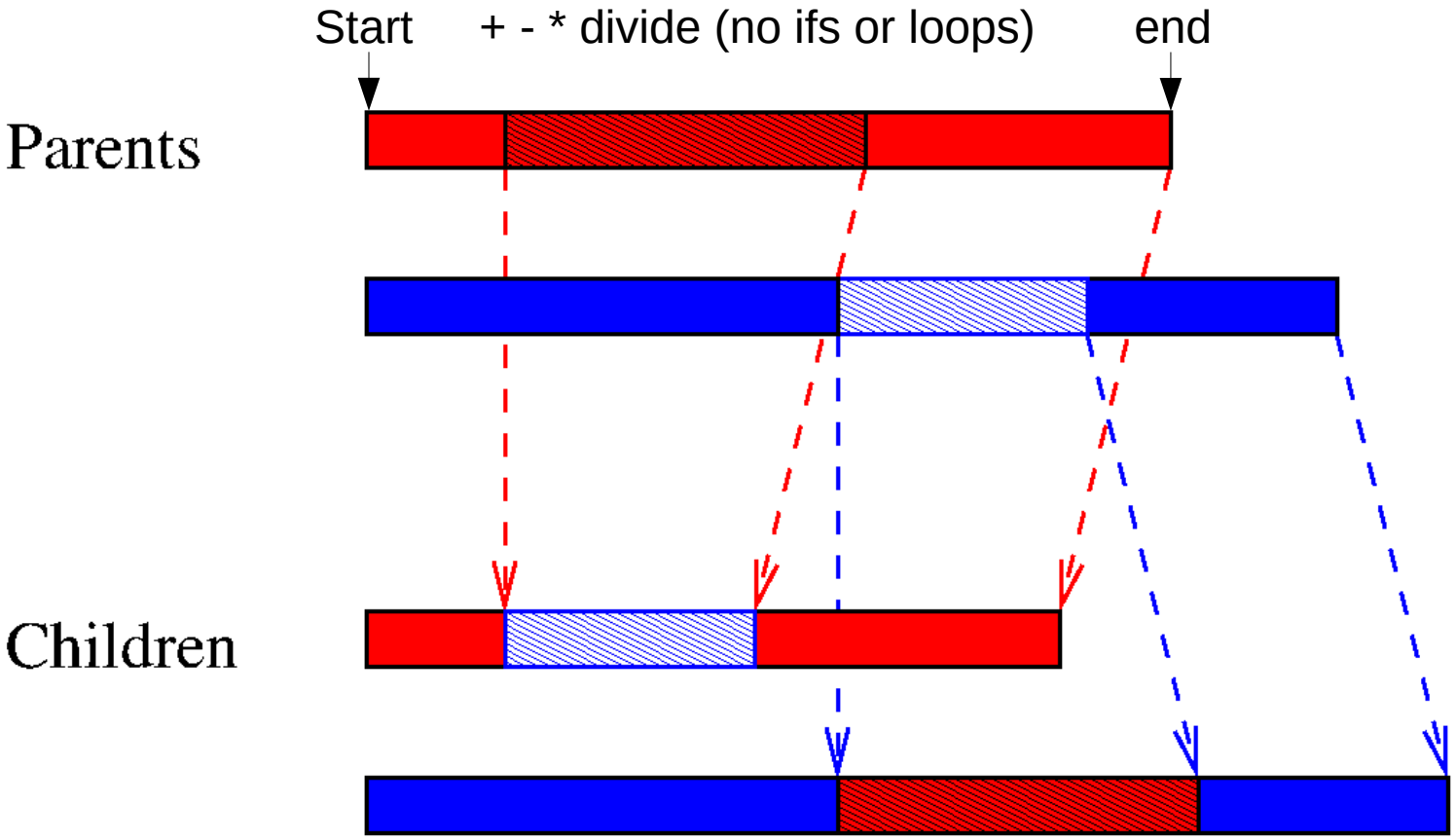
Predict Chaotic Time Sequence



Eight 8-bit (0-255) inputs:
A 128, B 64, C 32, D 16, E 8, F 4, G 2, H 1, time steps behind.

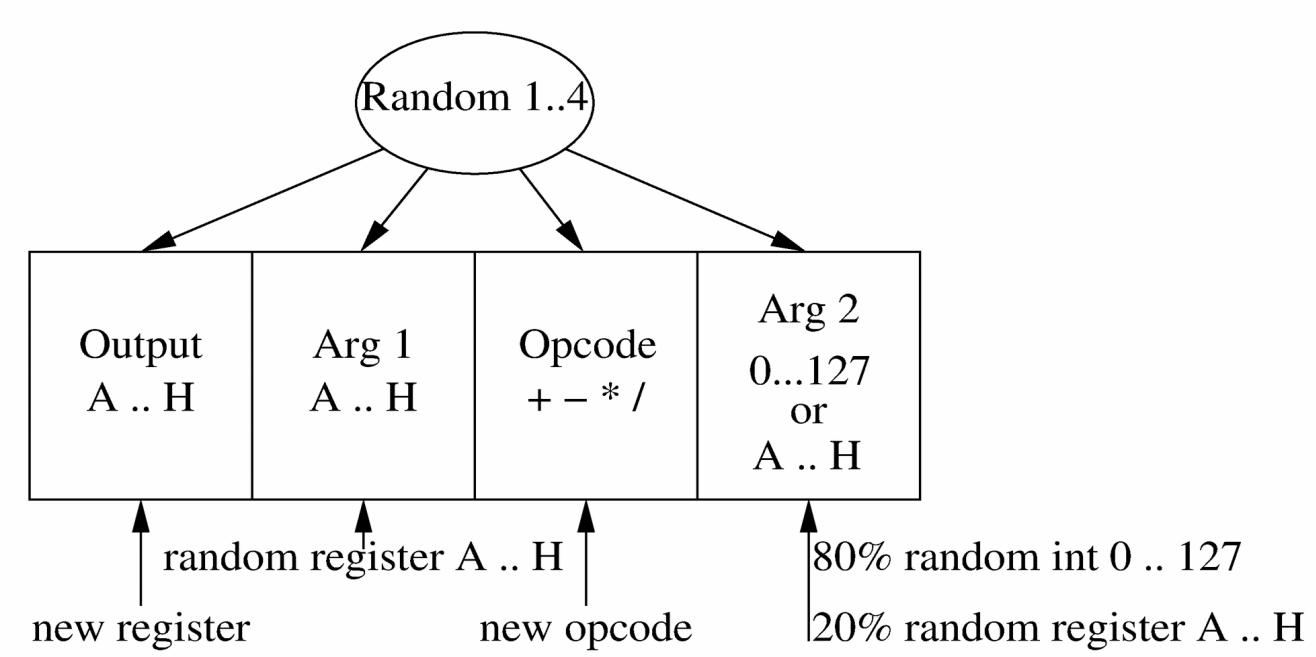
One 8-bit output: register A

Crossover: Two Parents ⇒ Two Children

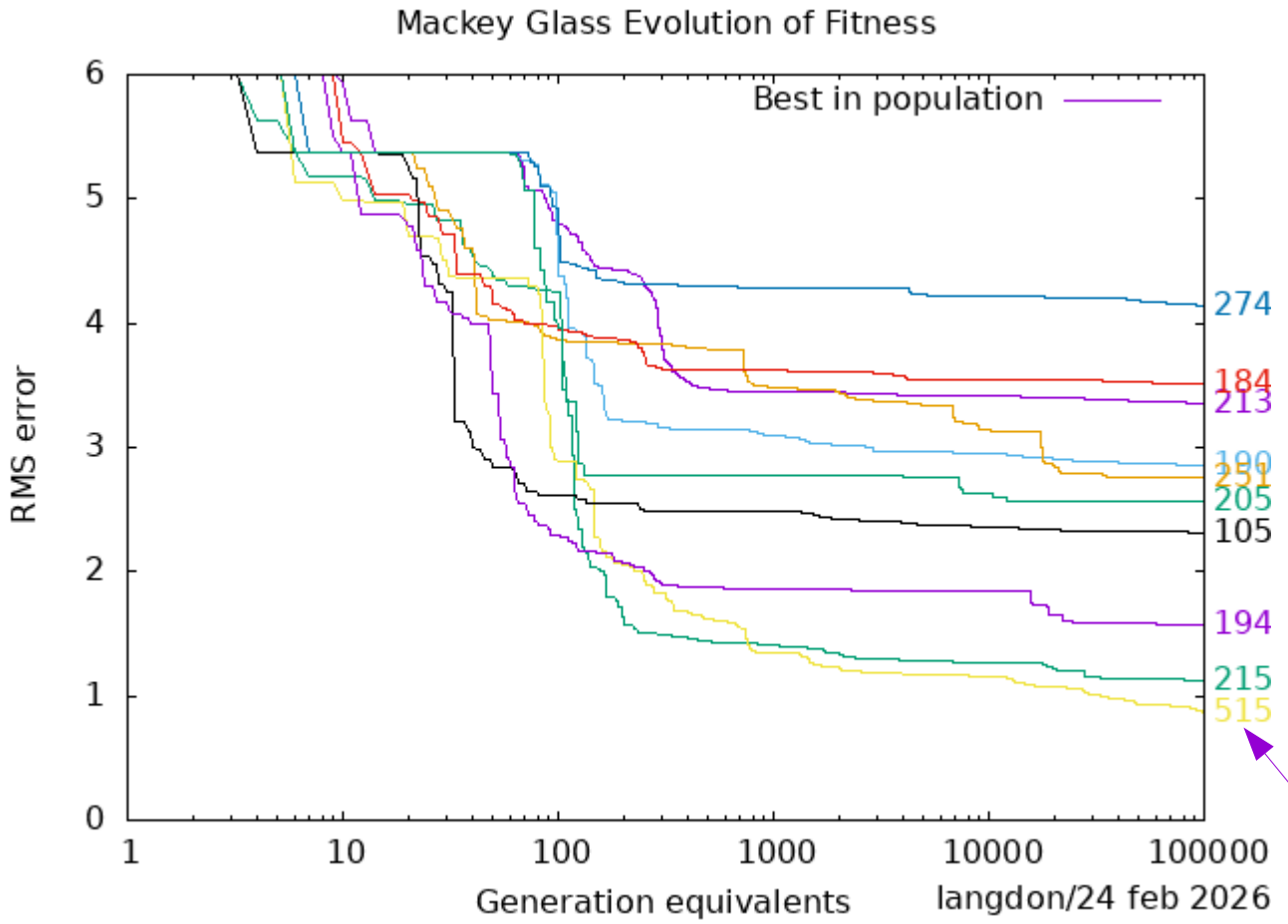


40% mutate both children four times

For each child choose independently at random four instructions to mutate.

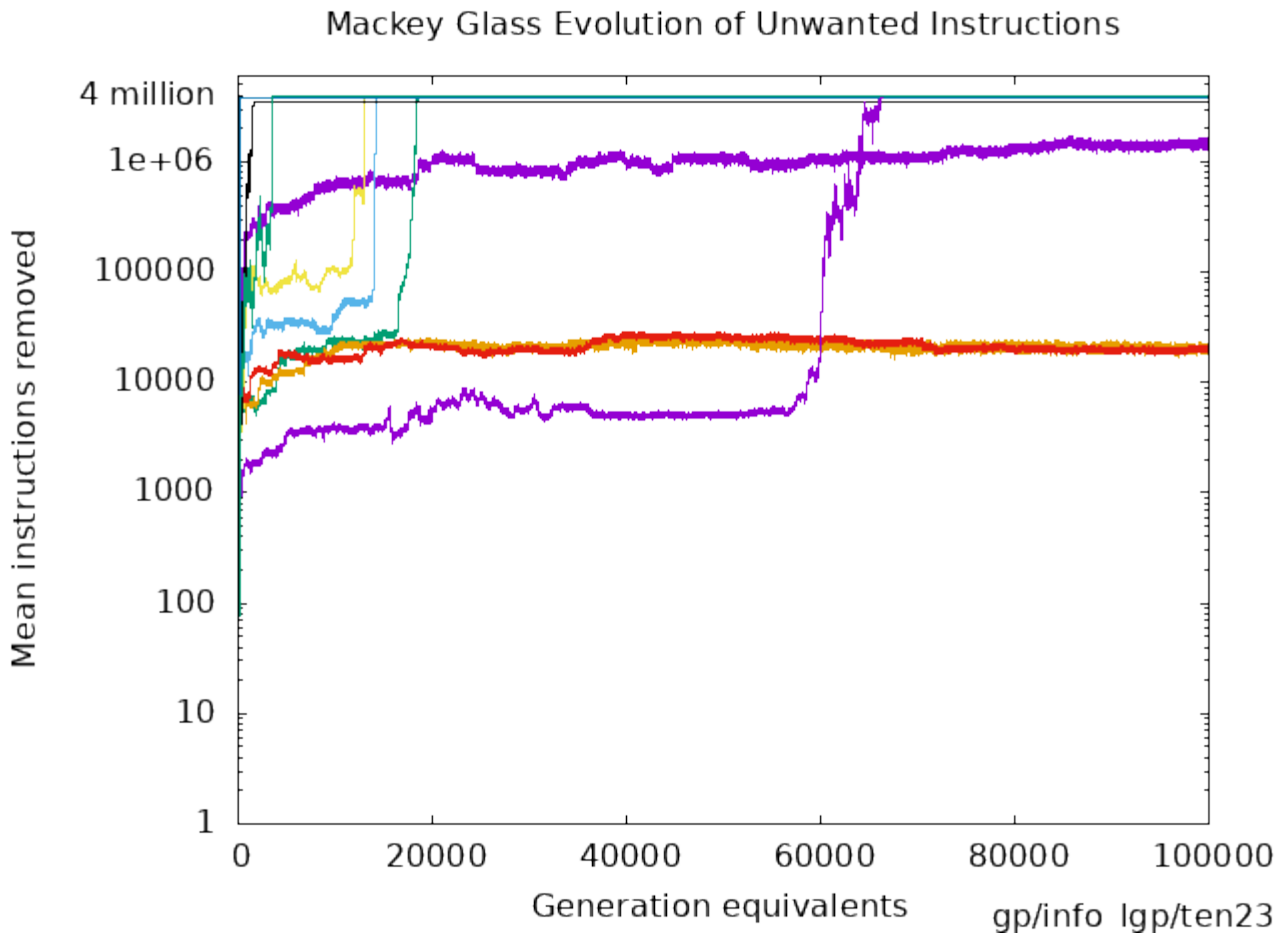


Evolution of Fitness (ten runs)



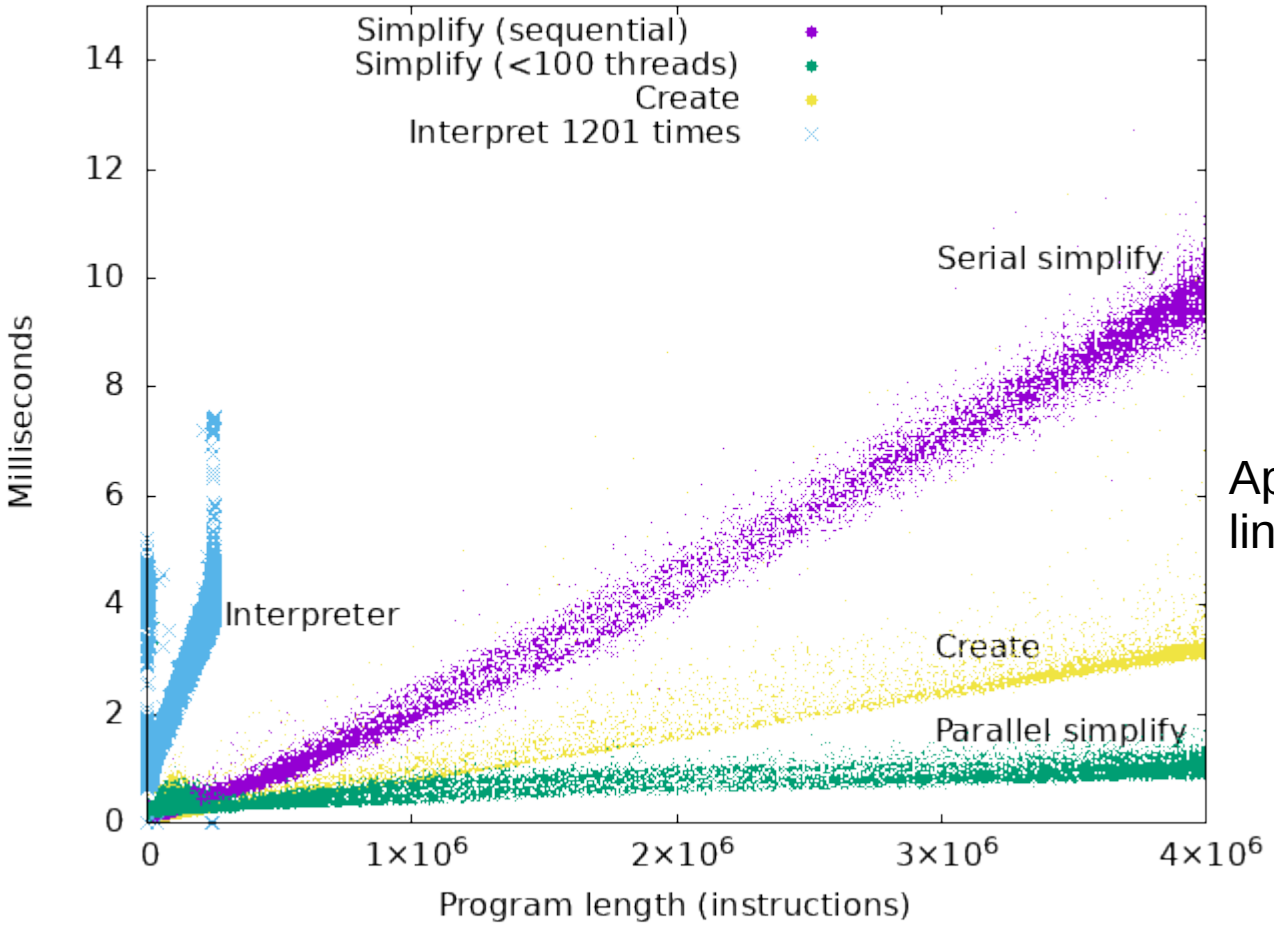
Number of fitness improvements in run

Evolution of Removed Instructions (ten runs)



Cost of simplify bloated redundant programs

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



Approx linear increases

Program Format

Linear Genetic Programming sequence of instructions (no ifs, no loops)

- Inputs loaded into 8 registers at start
- Program run start to end
- Answer extracted from results register A

Output A .. H	Arg 1 A .. H	Opcode + − * /	Arg 2 0...127 or A .. H

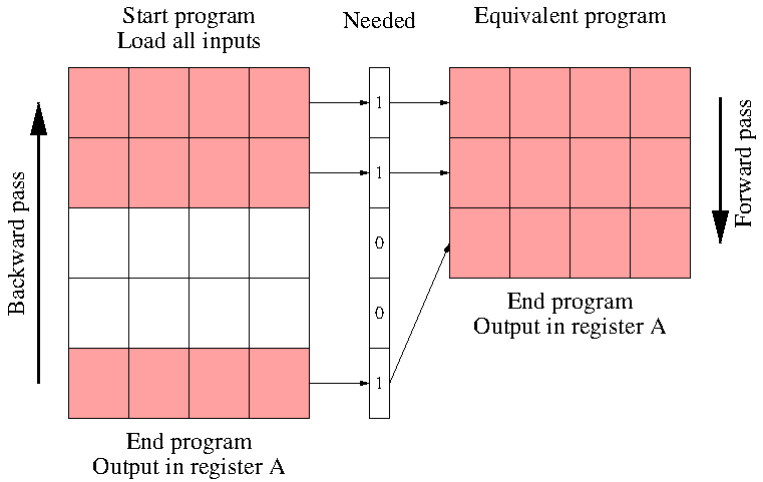
Instruction format

Output register = Arg 1 ○ Arg 2
Output and Arg 1 are always registers (A to H)
Arg 2 is register or integer constant (0 to 127)

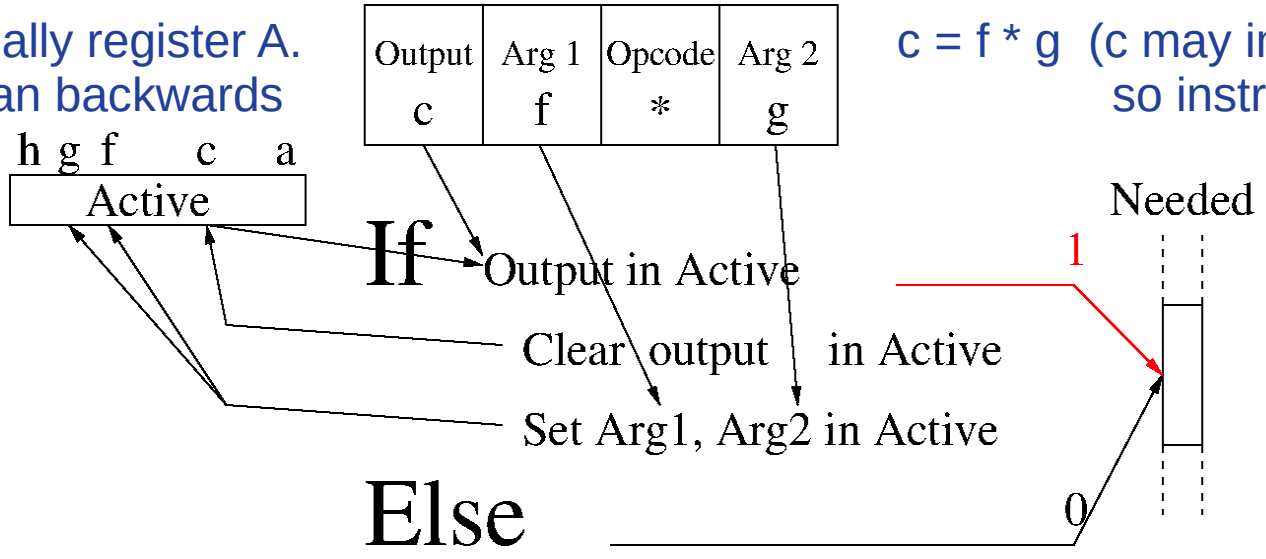
Sequential: Static Program Analysis

- 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 (one thread) Program Analysis



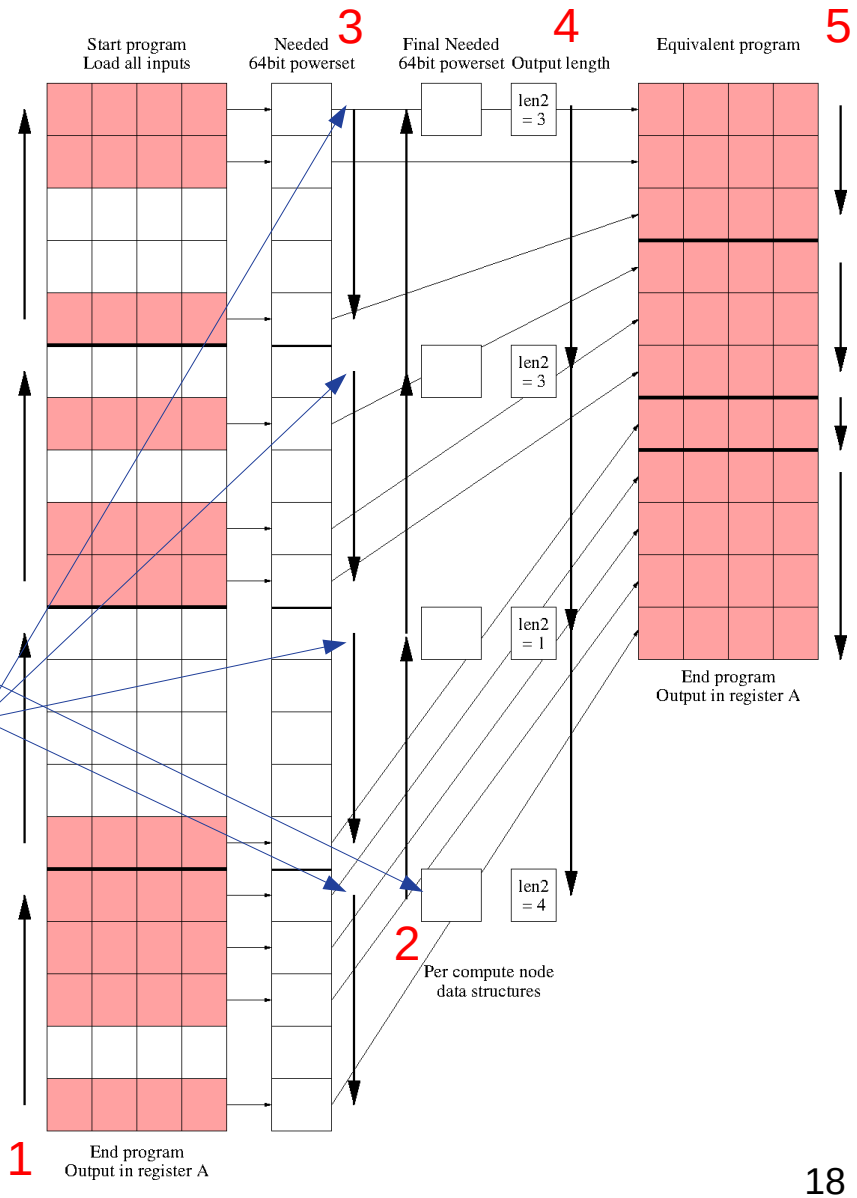
Active set initially register A.
Update as scan backwards



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

Parallel Program Backward Slicing

- 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

pthread pool: (simplified) main code

- If small program simplify_nopar
- num threads needed based on program length
- for(num threads)
  pthread_create(...simplify_thread...)
- **pthread_barrier_init**(&main_barrier,,2)
- **pthread_barrier_init**(&barrier,,num threads)
- **pthread_mutex_lock()**; nqueue = nthreads; **pthread_mutex_unlock()**
- **pthread_cond_broadcast()**
- **pthread_barrier_wait**(&main_barrier) //Wait threads

pthread pool: (simplified) thread code

```
forever
    pthread_mutex_lock()                //mutex needed for pthread_cond_wait
    while(nqueue == 0) //wait for work
        pthread_cond_wait()             //args condition and mutex
    n = __atomic_sub_fetch(&nqueue,1,__ATOMIC_SEQ_CST); //our task is n = nqueue − 1
    pthread_mutex_unlock()
    bot = n*frag //which part of program thread simplifies
    top = bot+frag−1 //but bounded by program size
    for(i=top; i>= bot; i−−)
        needed[i] = active & 0101010101010101 << out        //is out register in active
    for(i=top; i>= bot; i−−)
        active = active & (~(0101010101010101 << out))      //clear out from active
        //Use 64 bit vector instrction to set mask only if needed
        mask = ~_mm_cmpeq_pi8(needed,0)
        active = active | (mask & (base << reg1))
        if(not constant) active = active | (mask & (base << reg2)
    pthread_barrier_wait()              //wait for other threads
    for(i=nthreads−2; i>=n; i−−) //step backwards through other threads work
        mask = transfer last needed from thead i+1
    for(i=bot; i<=top; i++)
        if(needed[i] & mask) len2++
    pthread_barrier_wait()              //wait for other threads
    for(i=0; i<n; i++) //step forward through other threads work
        out += Len2[i] //find where to put our output
        if(needed[i] & mask) len2++
    for(i=bot; i<=top; i++)
        if(needed[i] & mask) copy instruction[i] to output[out] and out++
    pthread_barrier_wait()              //wait for other threads
    if(n==nthreads−1)
        pthread_barrier_wait(&main_barrier)     //exactly one thread restarts main code
```

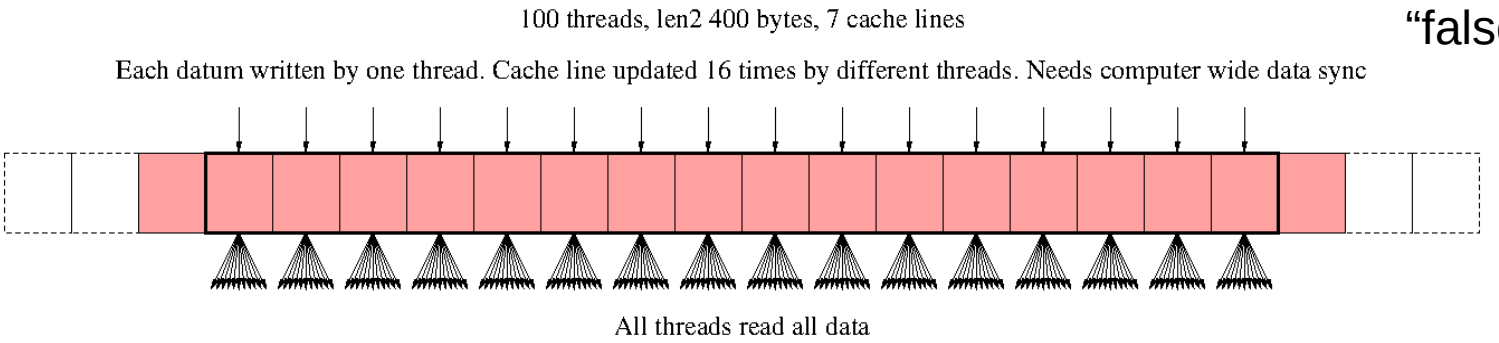
Memory caches

- 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 64 bytes)
- At present minimise memory but may suffer from “false sharing” i.e. L3 used for synchronization (can be very slow).

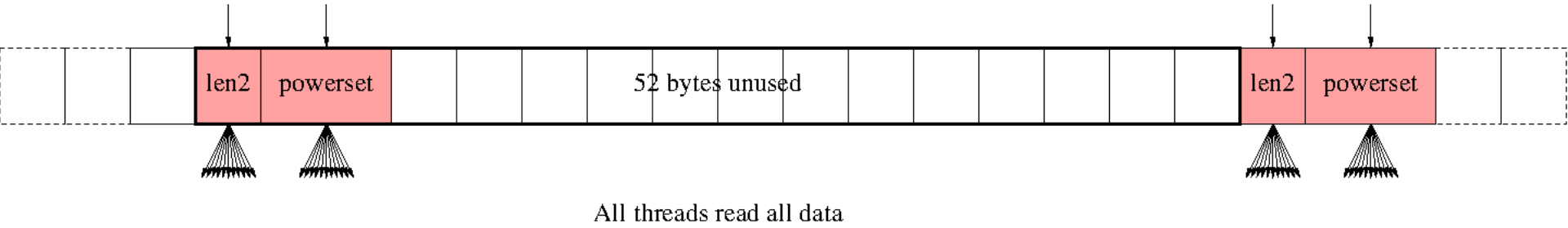
100 threads tightly packed len2 7 cache lines



"false sharing"

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

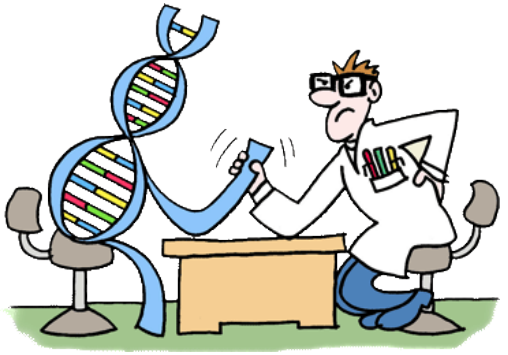
100 threads loosely pack per thread data



Conclusions

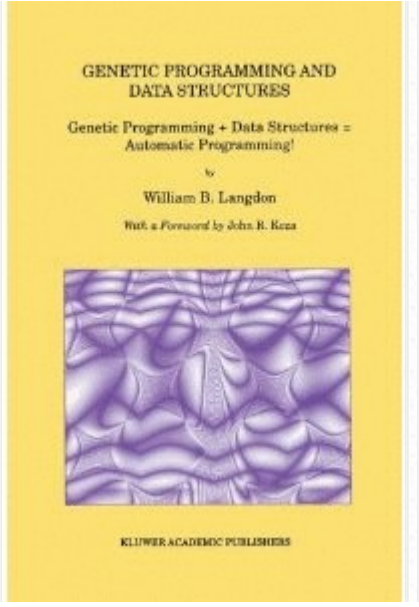
- Parallel approach removes a compute heavy bottleneck in large scale linear genetic programming
  - Effective speeds of up to 726 billion GP operations/second (effectively interpret 235 operations per clock tick)
- Future is parallel:
  - Pool of pthreads reduces overhead (**multi core**)
  - Wide parallel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

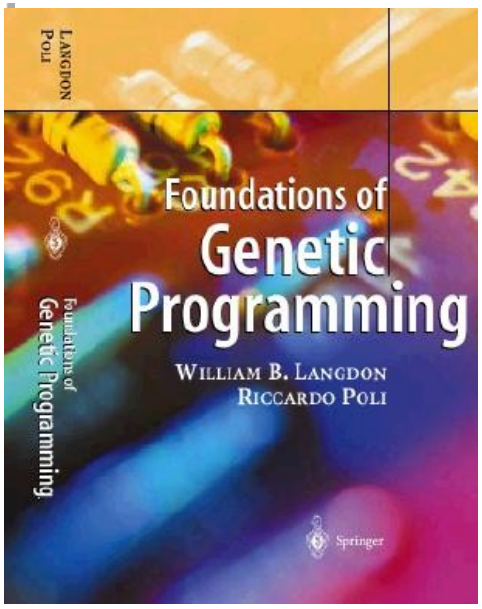


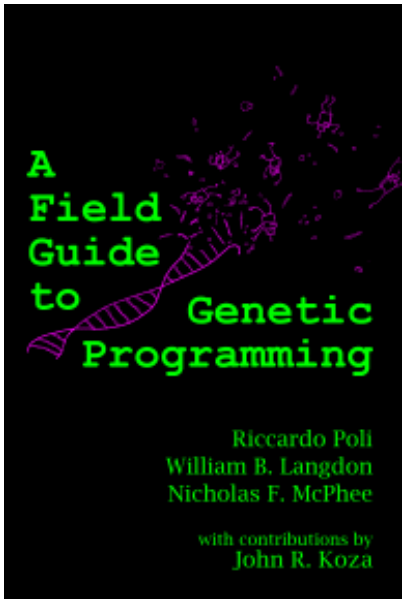


Genetic Programming Bibliography
https://gpbib.cs.ucl.ac.uk/

$10000 Humies
Submit by
28 May 2027







Is Mackey-Glass typical?

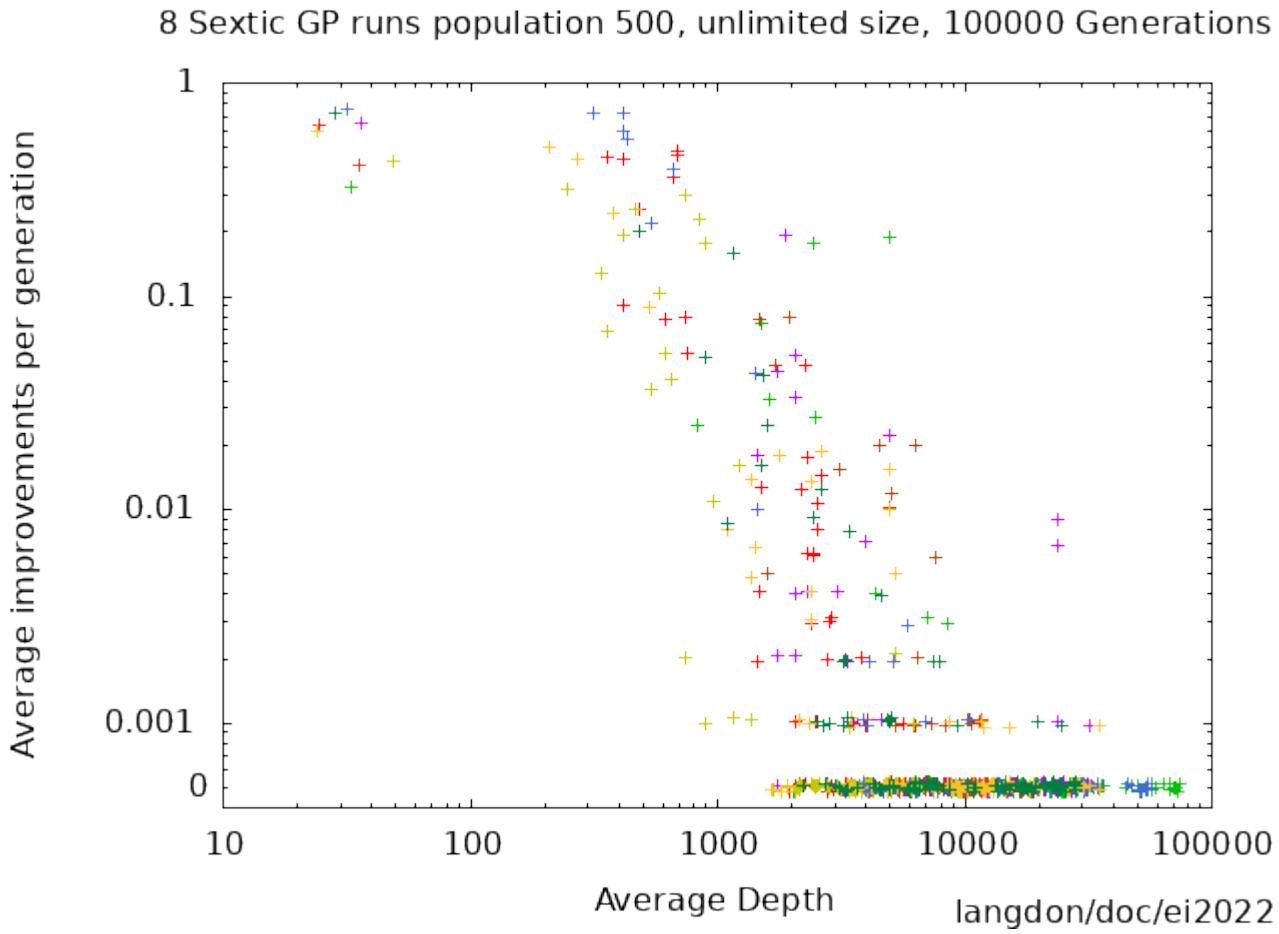
https://github.com/wblangdon/GPengine

- 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

Tree GP Long Term Evolution Experiments

- **LTEE** shows E.Coli continued innovation 83000 generations
- Genetic Programming continued fitness improvement a million generations BUT GP slows as expressions get deeper
  - Impact of mutations lost, e.g. due to rounding error
  - In deep integer trees 92% to 99.97% of evaluation changes have no effect
- Exponential decay with depth
  - Need to be close to error for tests to find them
  - On average <7 more than 50% errors detected
- Each case may differ in detail but information theory says entropy loss will occur and will lead to robustness

Deeper Tree programs harder to evolve



As the GP populations evolve they find thousands of improvements but at a slower rate as the trees get deeper. Note log scales.

Long Term Evolution Experiments with Linear Genetic Programming

- Invited book chapter *Advances in Linear Genetic Programming*, Banzhaf+Hu, Springer
- Run Linear Genetic Programming to a 100,000 generations
  - Exploit AVX512 (GI 2026) and multicore
  - Improvement continues but rate falls with program size
- Program's information content (entropy)
  - initially = entropy of inputs
  - falls to near entropy of desired output distribution
  - Almost all instructions do not change entropy

The Genetic Programming Bibliography

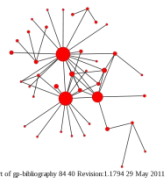
Mirror https://gpbib.pmacs.upenn.edu/

18791 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

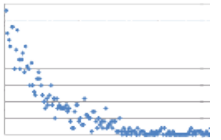
Downloads by day





A personalised list of every author's GP publications.

blog



Your papers



Text search

Googling GP bibliography, eg:
Explainability site:gpbib.cs.ucl.ac.uk