Parallel Evolvers: A Concurrent, Object-Oriented Genetic Algorithm for solving linear equations, analyzed across multiple parallelization strategies

CSCI4830 [Special Topics | Concurrent Programming]

by Zachary Doyle

Introduction

The Evolver system is a genetic algorithm; Ecosystems are host to many Evolvers, which make several adjacent guesses at a solution to an arbitrary set of linear equations. Evolvers whose best guess has a summed absolute error (SEE) less than the mean of all Evolvers in their Ecosystem will survive to reproduce, propogating their guess into the next generation. By these means, selective pressure is introduced to (somewhat) gradually coax the population of Evolvers toward the actual solution. Notably, the nature of linear algebra means that if we calcuate a coefficient matrix A and right-hand matrix B, we can set the solution, C, according to this equation:

$$AxC=B$$

-then we can compute the SEE given a supposed B and the actual A and C, without knowing the actual B.

The Evolver system is also (in all but one implementation) concurrent. While the notion of many discrete units working more or less independently of one another to shuffle across a vector space lends itself naturally to fairly naive parallelization, leveraging Java's support for

concurrent operations and migrating as much computation as possible into threads allows for maximized efficiency.

Basic Concepts

The life cycle of a Evolver is fairly consistent across all implementations. An Evolver can be initialized under two circumstances, which correspond to its two constructors – first, it can be instantiated with a random guess, as is done for Generation 0. In all following generations, it is constructed using two Evolvers as parameters, and the child's initial guess is the mean of its parents, with some noise to keep things interesting.

Once an Evolver is initialized, it is (one way or another, e.g., sequentially, in its own thread, or in a thread pool) called upon to make some guesses. Each Evolver is given a certain number of “steps” (dictated by its home Ecosystem); with each step, it may increment or decrement its guess for each variable by one- it computes the SEE each time and prefers whichever raises it the least (or lowers it). Once it has exhausted its steps, it reports its closest guess and corresponding SEE to its Ecosystem.

After this point, the Ecosystem picks a threshold error based on the mean error of the generation's evolvers; Evolvers above the threshold are culled, those below it are allowed to reproduce into the next generation, meaning they are paired with a random other Evolver (reproduction is single-gendered) and fed into the aforementioned two-Evolver constructor.

Implementation Differences

Sequential

The most naive of all, the sequential implementation simply has each Evolver make the best guess it can before moving on to the next.

Naive Concurrency

The first tier of concurrent implementations suffers from the fact that error computation was outsoured to the (singleton, and therefore sequential) Ecosystem. Nonetheless, there are some modest (and circumstantial) gains in performance, as apparently the benefit of running the guess-stepping in parallel outweighed the serious overhead from a huge, underused threadpool.

Improved Concurrency

Though this implementation moved error-calcuation into the Evolver's scope (and therefore parallelized it), it continued to suffer from massive overhead due to the fact that threads were spawned, killed, and respawned to accommodate each and every Evolver.

Pooled Concurrency

Arguably the fastest implementation (depending on input and random circumstance); relies upon Java's util.concurrent.Executors' newWorkStealingPool() method to generate a work-stealing threadpool (the ideal pool type for this problem, according to Herlihy)

Regulated Pooled Concurrency

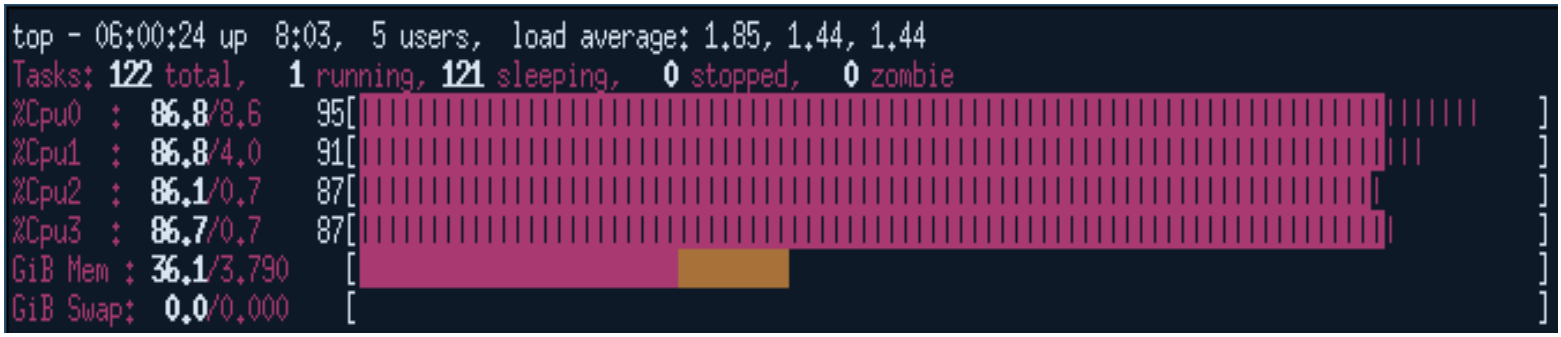
The difference between this implementation and the above lies less in the details of parallelization and more in this one's ability to self-regulate the population of Evolvers. The survival threshold is correlated logarithmically with how close the population is to the “ideal” population per detected logical thread, and a more intense “cull” takes place when the population passes a more serious overload value. While generally successful, this implementation can run for many more generations than others- making it a useful stress test for core usage and fairness.

Results

Name	Running		Total
JMX server connection timeout 14	9,018 ms	(7.8%)	114,892 ms
RMI Scheduler(0)	0 ms	(0%)	114,892 ms
RMI TCP Connection(1)-127.0.0.1	88,883 ms	(77.4%)	114,892 ms
RMI TCP Accept-0	114,892 ms	(100%)	114,892 ms
Attach Listener	114,892 ms	(100%)	114,892 ms
Signal Dispatcher	114,892 ms	(100%)	114,892 ms
Finalizer	0 ms	(0%)	114,892 ms
Reference Handler	0 ms	(0%)	114,892 ms
main	114,892 ms	(100%)	114,892 ms
RMI TCP Connection(2)-127.0.0.1	113,997 ms	(100%)	113,997 ms
*** Profiler Agent Special Execution Thread 6	0 ms	(0%)	94,027 ms
*** JFluid Monitor thread ***	0 ms	(0%)	94,027 ms
*** Profiler Agent Communication Thread	94,027 ms	(100%)	94,027 ms
process reaper	0 ms	(0%)	59,994 ms
ForkJoinPool-1-worker-0	18,949 ms	(21.8%)	86,998 ms
ForkJoinPool-1-worker-3	19,968 ms	(23%)	86,998 ms
ForkJoinPool-1-worker-2	22,937 ms	(26.4%)	86,998 ms
ForkJoinPool-1-worker-1	18,936 ms	(21.8%)	86,998 ms
*** Profiler Agent Special Execution Thread 7	0 ms	(0%)	6,005 ms
SIGINT handler	0 ms	(0%)	6,005 ms

While the relative effectiveness has been touched upon above and will be delved into in more detail below, VisualJVM, a profiling tool for Java, has an extremely helpful thread visualization view; while Java's thread notation can be somewhat obtuse, especially for someone not versed in the inner workings of the Java Virtual Machine, one can clearly identify the four worker threads associated with each core on the testing machine.

The four threads at the bottom show approximately one (very large) generation over the course of about ten seconds: the four worker threads run steadily at first, using nearly 90% of all four cores' time. Eventually, as Evolvers finish their guesses, the thread pool runs out of work and must wait a few seconds for the Ecosystem to tally up the survivors before embarking on the next generation.



During the “high tide” of generation cycles, all cores are put to the test.

Conclusions

With some finangling, this genetic algorithm can be made 'inherently parallel', in that Evolvers can retain all the information they need to in their own memory before passing it onto the sequential Ecosystem. It's not hard to conceive of scenarios where this is not the case- Evolvers might need to update a “best solution” or some similar construct, which would require far more delicate parallelization.

The compartive running times of each implementation are hard to assess; most of the moving parts of this program involve (pseudo)random numbers, making replicatable testing conditions difficult to establish. Still, even at a glance, it is obvious a thread pooling, work-stealing implementation that accomplishes the majority of computation within the parallelized portion of the code is by far the fastest, with respect to actual methods of concurrency (i.e., genetic strategies notwithstanding).