Concurrent GARTH (Genetic AlgoRiTHms) using C++11: A Framework for Concurrent GAs

J. Caleb Wherry
Virginia Tech
Department of ESM
Blacksburg, Virginia
cwherry@vt.edu

ABSTRACT

Genetic algorithms (GAs) are a fasinating area of optimization theory that draw from ideas set forth by Charles Darwin in *The Orgin of Species*. GAs have been shown to improve optimizations that we as humans cannot obtain on our own because of our inherint linear thinking. Non-linear optimization problems benefit most from GAs since GAs are able to search the non-linear space much farther and find optimal solutions much quicker than traditional optimization methods.

In this project we would like to take the typical framework that GAs are built on and make it concurrent and lock-free using the new C++11 standard. This will involve multiple stages in the development cycle that deal with designing the concurrent object for the GA threads to work on, implementing the GA framework using the new concurrent object, and then benchmarking the concurrent framework on some real-world applications.

We have done previous work in sequential GAs and have developed a framework in Java that runs on a uniprocessor. We will draw from this knowledge when developing our new concurrent GA framework. After the concurrent framework has been developed, we will benchmark the framework based on the number of threads that work on the GA. We will then be able to analyze the tradeoffs in using more threads and how much speedup we obtain using the concurrent object.

As of right now, the real-world application will be a physics problem that is non-trivial in quantum condensed matter physics: the optimal placement of charges on sphere with the lowest energy state[3]. GAs have been shown to optimize this problem more efficiently than tarditional methods when the number of particles exceeds 100.

The main goal of this project is to apply a concurrency model to GAs for real-world problems. The difficulty here will be in integrating the concurrent object into the framework such that we see a gain in speed when using more threads, not a speed lose that is occured in using the concurrent object.

Categories and Subject Descriptors

D.1.3 [**Algorithms**]: Misc—*genetic algorithms, concurrent objects*

1. SCHEDULE - UPDATED

1. February:
   (a) ~~Architect and design GA framework.~~
   (b) ~~Research lock-free objects that will best fit the GA process (Lock-free queue or Lock-free B-tree).~~
   (c) ~~Implement lock-free object that is decided upon from above research.~~
2. March:
   (a) Finish last details for lock-free object.
   (b) ~~Implement GA framework to work with new concurrent object.~~
   (c) ~~Formulate quantum CMP application requirements.~~
   (d) ~~Submit mid-semester report.~~
3. April:
   (a) Complete GA framework.
   (b) Apply framework to quantum application.
   (c) Run benchmarks based on varying numbers of threads.
   (d) Write final proposal and create presentation.
4. May:
   (a) Submit final report and give presentation!

2. MID-SEMESTER REPORT

I have gone through the schedule above and marked through the items that have been completed. For the most part, I am on track with that I wanted to have accomplished. The biggest 'unfinished' portion is that my current concurrent objects use course-grained locking. I really want to make them fine-grained and potentially lock-free. But more on that later. The progress so far...

I started off with research on what type of object would be best for a GA framework. There are no inherent access operations that need to be maintained for GAs. The typical object that is used to store data is a database. I just need the ability to access N elements (rows) and operate on them. With the thread model, this means that each thread will need to be able to access N-elements, do some work, and then move on to N more elements. I read through both [1] and [4] to get an idea of what would be best for this type of situation. I had originally wanted to implement a more complicated object like a B-Tree (since that is what most databases are built on) but that amount of work needed would have meant I could not apply the structure to a real world problem. The point of my project is to implement some type of concurrent object and get it working with a GA framework and do some interesting science. On top of that, there is almost no literature on using concurrent objects with GAs so I decided to start with a simple concurrent Vector as my concurrent object. I wanted to be able to get the framework working in a concurrent manner so that is why I chose this object. If I have more time towards the end of the semester, I would like to potentially try something a little more complex. However, if you feel that I should tackle something harder now, I can redo my schedule and focus on that for the next few weeks.

The one paper I did come across that has some sort of concurrency model with GAs was [2]. It did not help me much with deciding on how to implement GAs with concurrent objects but it did describe some pitfalls that they ran into that should be beneficial to what I am doing.

Going back to the schedule, I completed the GA framework baseline work (1.a and 2.b). The model I have is a base class Zoo that everything is housed in. A ZooKeeper (another object) controls everything that goes on in the zoo. The ZooKeeper creates the initial population of Organisms and readies all of the Breeders (threads) to perform work on the population (and in turn the Organisms). Each of these threads then act on the population and mate, mutate, and create the next generation of Organisms. Once the pool of threads (number determined by the nice C++11 function std::thread::hardware_concurrency) have completed with a generation (population at a specific step), the ZooKeeper steps in and moves along the simulation to the next generation. Right now each thread interacts with the population in the manner described earlier: course grained. When a thread needs work, it locks the vector and basically makes working on the vector sequential. This is for sure a performance issue and is why I have not marked through 2.a on my schedule. I would like to implement fine-grained access and even potentially lock-free access. This way the ZooKeeper can potentially monitor the population without having to grab any locks. Right now the ZooKeeper waits until all the Breeders (threads) are done before it does anything. I think it would be nice to potentially have the ZooKeeper calculate statistics on the population mid-generation. However, since it would have to grab the locks to traverse the population, it would affect performance severely.

The last thing that I have done is to start solidifying the specific problem I want to try and tackle with this new GA framework (2.c). The main task here is to create an Organism that correctly exemplifies the problem at hand. I have created the Organism class on paper and the fitness function which will be the driving force behind how well a 'solution' is in each generation. I have not put these into code yet but am planning on doing that in the next week. Right now the framework works with a base class Organism that just has basic properties. Another bonus of designing the libraries like I have is that it will be easy for others to create their own Organisms and substitute them in. This means that this framework will serve as a general purpose framework and has the potential to work on any GA optimization problem you have, not just the one I have selected. This makes it extremely appealing for other areas of research, especially if I can show a significant speedup using concurrency!

All in all I am really enjoying the project and think I am making good progress. I am hoping that the benchmark results I have at the end dealing with how much of a speedup I get with N threads is going to be a good result. I would potentially like to add another layer of parallelism and involve MPI. That way I would push out sections of the population to different 'Islands' and have multiple Breeders working on different groups in the population. I have tried to design my software in a way that it should be able to be updated to accommodate this type of structure. The idea behind GAs is to have a very large and diverse population so that searching will be more likely to reach a global minimum instead of a local minimum. This, however, might be a future goal beyond this class. I want to have at least 1 Island working with multiple breeders that concurrently access/modify/traverse the population.

3. ADDITIONAL INFORMATION

I am releasing this project under the open source MIT License. I have a public repository designated for the development of all code/papers/presentations associated with this project. Anyone can see the complete history of the project here:

`https://github.com/calebwherry/Concurrent-GARTH`

I have full continuous integration testing enabled through Travis-CI that kicks off builds of my project on each push to the remote repository. This enables me to quickly see the health of my project. Along with this I am actively developing unit tests for the libraries I am creating. Note: This repo does get out of sync from time to time especially when I am working on new features. I try and keep everything in the repo clean and not broken.

I have built the project so that only cross-platform features of the new C++11 standard are used. I am actively testing the build on both Linux systems and Windows systems using cmake as my build environment to handle the different systems/compilers. I hope that at the end of the project, I can make useful comments about different compilers on different systems that implement the C++11 standard. If they work like they should, then I should not have any issues between the systems/compilers and should have positive cross-platform results!

As of now I am compiling against 4 different compilers: GNU GCC, Clang, MSVC, and Intel. So far I have ran into no issues implementing features of the C++11 standard on all compilers. As I dig deeper into using new standard features though, issues would arise. I hope to be able to point these out in my final results as well at the end of the semester.

4. REFERENCES

[1] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, ???, 2012.

[2] M. Köppen and E. Dimitriadou. Design and application of hybrid intelligent systems. chapter Concurrent Application of Genetic Algorithm in Pattern Recognition, pages 868–877. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2003.

[3] T. Pang. *Introduction to Computational Physics*. Cambridge University Press, Cambridge, 2006.

[4] A. Williams. *C++ Concurrency In Action: Pratical Multithreading*. Manning Publications, ???, 2012.