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 fascinating area of optimization theory that draw from ideas set forth by Charles Darwin in *The Origin of Species*. GAs have been shown to improve optimizations that we as humans cannot obtain on our own because of our inherent 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 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 trade-offs 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 traditional 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 occurred in using the concurrent object.

Categories and Subject Descriptors

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

1. INTRODUCTION

The main objective of this research was to take a typical GA process and see if adding a concurrency model to it would increase the rate of convergence to an optimal solution. In this section, I provide general overviews of the background research, different methods/ideas, and an explanation of the application area of interest ("real world problem").

1.1 Background

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 so this simplified the process greatly. The typical object that is used to store data is a database but I wanted to go completely native and in-memory when designing my framework. All I really needed was the ability to access N elements (rows) and operate on them of data. 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 course-grained 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. However, this method does not provide any insight into speeding up GAs since we are essentially serializing the concurrency we get from the threads. I talk more about this in the Concurrent Methods section.

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 were beneficial along the way.

One of the reasons why there is not much literature on using concurrent, in-memory data structures to support GAs is as I said before: this is typically taken care of by a database. Databases are extremely optimized now-a-days such that a scientist does not need to create a framework that relies on anything other than a database. The creation of a schema that does all of the process I talk about in this paper is done all of the time and the researchers are fine with taking the computational overhead of using a database (which is a lot of cases is not even that much). However, using databases brings in a dynamic that I am trying to avoid: dependence on system tools. I wanted this project to use completely native C++11 constructs and not reply on third party systems. That way I can really dig into C++ and see what can be done natively.

1.2 Genetic Algorithms (GAs)

GAs are a fascinating area of research that draw from natural selection to help in finding optimal solutions to a given problem. One of the founding evangelists of GAs is John Koza who uses GAs to do fascinating things like find optimal design patterns for radar dishes such that interference is minimized. This leads to radar dishes that perform better in all climates and weather and can help boost signal propagation/receiving. Not all problems are well-suited for GAs and knowing where to apply them is an art in-of-itself. However, when used in the right settings, GAs offer a fascinating view at how nature can produce optimal solutions to very hard problems.

The main idea behind GAs is survival of the fittest. Given a certain cost function (fitness function), organisms in a population can be ranked against one another to figure out which ones are the most fit. This fitness functions maps an organism's genome (genetic makeup) onto a real number. After this selection and ranking process has happened, mating occurs between organisms. There are many many different types of mating: unary, binary, tertiary, partial, etc and they all have their own benefits and weaknesses. In this paper, I use binary reproduction since it is the easiest to model and provides fairly nice convergence results. However, I have created the framework in such a way that one can easily modify the type of mating that happens (More in the Methods section).

The next step that is extremely important in GAs is mutation. Without mutation, local minima are more likely to be converged to which can skew results. When doing optimization theory, local minima are usually not what are wanted, rather we look for global minima. This is because we want the best possible solution to a problem, not a 90% or even a 95% solution. Mutation allows us to jump out from local minima and potentially land on a different part of the landscape that will take us to a global minima. There are many different techniques to performing mutation on organisms. Some people never mutate the most optimal organisms while others base mutation on a random probability in each generation of new offspring.

When looking at the performance of a GA, the most important factor is the convergence rate. This is the rate at which a GA takes to get to an optimal solution. Some GAs are very fast but only return a local minima a majority of the time. Some GAs are extremely thorough and provide a very wide search space but take very long to run. There is definitely a trade-off here and that is where the concurrency model really can come in handy. If we can use threads to process more organisms in a population that have a higher degree of variety in their genome, then we can potentially work our way to an optimal solution at a faster rate. However, if we are not careful, we can easily slow down the process by adding in the over head of creating/destroying worker threads. I touch more on this in the Concurrent Methods section.

1.3 C++11 Standard and Native Threading

I chose to do this project in C++ and use the C++11 standard because I love the idea that C++ had been pushing lately of ”going native”. The idea that language can provide a rich set of features to accomplish a wide variety of tasks is awesome and I really appreciate the effort that has gone into making C++ a more robust language. I am also a Software Engineer by profession so using the latest and greatest tools is always a lot of fun!

In the ”new” C++11 standard, threads are provided natively. Before, libraries like pthreads had to be used that were rather clumsy and hard to debug. The C++11 steering committee made it point to say that this feature was needed for the benefit of all C++ programmers and I agree with them. Adding in concurrency models to the language will (hopefully) open up developers to writing more concurrent software.

The C++11 standard also offers other cool features like locks, futures, lambda functions, and atomic templated variables. This allows developers to not only write code the is dead-lock free but also the potential to write code the is lock-free and wait-free, which greatly adds to the overall performance of concurrent software.

Since I am using the C++11 standard, I am also compiling my project on multiple compilers to do a mini-test to make sure that my software is truly compiler agnostic and platform agnostic. I explain that in a little more detail in the next section.

1.4 Software Design and Engineering Principles

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 am currently compiling against 4 different compilers: GNU GCC, Clang, MSVC, and Intel. I ran into no issues implementing features of the C++11 standard on any of these compilers.

2. CONCURRENT METHODS

This part of the research was really open ended and lead to a lot of neat and interesting discoveries on my part. I went into creating the concurrent objects without really knowing *what* I needed to create. I knew I wanted to start out with creating a simple ConcurrentCounter that I would most likely use to keep track of the number of generations I had in my GA. I also knew that I would start off by creating a simple ConcurrentVector that was course-grained to store my population. After that I knew I wanted to try my hand at a certain lock-free data structure that would embody GAs. However, I did not exactly know in what direction I was going to go. These are a few of the things I came up with a some things that are currently in the works that will potentially make their way completely into my framework.

2.1 Concurrent Counter

My ConcurrentCounter object was a pretty simple implementation very similar to the one we did at the beginning of the semester. I used an atomic variable to keep track of the main counter variable and any number of worker threads can update the counter value since it is atomic. This makes for a nice counter for the number of generations I have processed in my GA. Since the implementation is not all that interesting and is pretty simple, I will not discuss it any further here.

2.2 Course-grained Concurrent Vector

The course-grained ConcurrentVector was another simple abstraction away from the typical STL Vector library in C++. Since the Vector class is not thread-safe, I wanted to created a wrapper object that used locks to make the Vector class thread-safe. This was simple in that for each function that I needed in Vector, I placed an exception-safe lock_guard before the function call which would be destroyed when it went out of scope. This allows for clean code that is easily readable but also functional in a concurrent setting. Since this method only serializes the calls to the Vector functions, we still have contention for successful calls to highly used functions like contains(). This causes problems and is easily solved by doing away with the locks entirely (lock-free) or creating more locks and have them apply to smaller sections of the data (fine-grained locking). Since the fine-grained locking is just a simple extension of course-grained locking in this case, I did not pursue that avenue. Instead I decided to create a new data structure based on a Vector that has some extra properties that I will describe below.

2.3 Lock-Free Data Structure

I am still unsure as to what to call this data structure. It uses a fixed size Vector (probably better to call it an array) and uses a hand-over-hand method to evaluate the different elements in the Vector. However, the way the Vector is constructed, passed, and evaluated is highly specific to GAs so naming it is still underway.

The main idea behind this data structure is the fact that once a population is created in my GA framework, the size never changes and elements are only manipulated once: to mate. All other interactions can either be dealt with naively (in terms of running statistics on a population, not requiring locks) or atomically. The last point is very important. I use 1 atomic variable called currentCount in this class to keep track of where in the population a thread is currently mating organisms. This allows me to use CAS operations to change the current value to the current value plus how ever many organisms it takes to mate. For example, say I have implemented a binary mating routine (needs 2 organisms). Thread A executes the CAS operation on the currentCount variable and sets it to currentCount+2. Therefore, when Thread B performs the CAS operation and is succeeds, it is now working on a different part of the vector.

This hand-over-hand type mating provides a way for me to regulate (or load-balance) between threads without having to really do any work. If one thread is running faster than others and is getting interrupted less, it can get in and process more matings. There is a potential contention point here if the GA mating routine is very fast. If this is the case, then all threads will be fighting to perform the CAS operation on currentCount and time will be wasted. A solution to this would be to divide the Vector into distinct sections that each thread works on alone. I did not like this solution since it seemed like the easy way out. So, I went with the CAS-based Vector hand-over-hand approach.

2.4 Thread Pool

In all of the approaches above, the worker threads were created and destroyed after each generation. This is a terribly inefficient approach but the beginning goal of the project was to just get the GA framework working with a concurrent data structure. However, I am recently turned to figuring out the best way to fix the issue of having the overhead of creating/destroying threads. The main solution that most people use is a Thread Pool. However, C++11 does not yet provide Thread Pools natively. There are a few third-party libraries out there that provide this functionality but since I wanted to stay all native, I did not want to include those. I am only in the theory phase of the piece of the project and do not have any results. However, one of the most common ways of implementing Thread Pools seems to be barriers so that worker threads can all start and then wait on other threads to finish before continuing on to the next step. This fits perfectly in with the GA framework since mating, mutation, and selection all have to be performed on the same generation of a population before any of them can move on. I speak more on this in the Future Work section.

3. GENETIC ALGORITHM MODEL

...

4. APPLICATION AREA: QUANTUM CONDENSED MATTER PHYSICS

...

5. RESULTS

There are 2 main results that I wanted to present when I first started this project. The first being how my design, creation, compiling, etc went between platforms and compilers. This mainly stemmed from the idea the all current compilers should now fully support the C++11 standard. If this is true, then all compilers on all platforms, when using the C++11 compiler flag (GNU g++: -std=c++11), should run my C++ native software just fine.

The second result was supposed to be a performance benchmark between varying number of threads to see when the trade-off was too great with more threads. However, these results are extremely elementary and not complete since I ran out of time to perform a thorough analysis.

5.1 Cross-platform and Compilers

...

5.2 Concurrency Performance

...

6. FUTURE WORK

There are many areas that I would love to expand upon in this research. The one being the performance analysis based on the number of threads that are used to process organisms in a population. This is at the very heart of a GA performing well since the main ideas behind a good GA are fast convergence. Having a general performance framework that tested different GAs with different thread counts would really help in understanding this.

Another area that I am extremely interested in is not only gaining performance increases out of using concurrent models but also by using distributed models along side them. Extending my current research would mean using a third-party library like MPI to distributed sections of the population to many cores. Then each core would have a ThreadPool that would operate on the subset of the overall population. This ability would greatly increase the search space of the GAs and hopefully provide better results for global convergence.

The C++11 thread model has a lot of different features and I have only just scratched the surface. I would love to dive deeper into the standard and see what other performance tweaks I would make just by better understanding how to use constructs like weak versus shared pointers. There are memory overheads that go with using these nicer features and understanding them better could potentially lead to drastic.

The last thing I would like to look at encompasses all of the above ideas: for a framework to be beneficial to scientist who work with large data sets, the framework has to be scalable. With threading, you can only scale so high before you lose performance because of swapping of resources. Normally you will only start however many threads that is equal to the number of CPUs/cores you have. Thus, there are not many processors that have over 32 cores. However, when looking at machines that are distributed, there are thousands of processors on those machines. Being able to scale a GA framework up to work on such supercomputers as Titan at ORNL or the BlueGene serious would be a tremendous jump forward into providing non-linear optimization techniques via GAs.

7. 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.