Garrett Scholtes

CS6068 Parallel Computing

Final Report

2015-12-09

Parallelization of Root Finding Methods

Introduction

For this project I have considered the problem of finding the unique root over a continuous computable function $f: \mathbb{R} \to \mathbb{R}$ over a sign changing interval. Many engineering applications require the ability to discover zeros of functions (often indicating points of equilibrium or spontaneity in a system) and as such it is worthwhile to investigate means to speed up this process.

Several methods already exist for sequential solutions to discover roots. Many of these are already converge quickly, but are often designed only to solve particular kinds of problems. For example, Newton's method rapidly converges, but only for functions which the first derivative can reliably be estimated, if it converges at all. There are many methods that are tailored to find roots of polynomials, some of which also open themselves to parallelism (e.g., Durand-Kerner method, to be discussed at the end of this report).

The implementation portion of this project focuses mainly on parallelizing the bisection root finding method. We will refer to the parallelized method as the *n*-ary section method throughout this report.

Complexity

The bisection method for finding roots over a sign-changing interval works by splitting the interval in half, and by repeatedly applying the method to whichever new interval is sign-changing. When the interval reaches desired precision, the method returns.[1] The *n*-ary section method follows a similar procedure, but by splitting the interval into *n* intervals. The bisection method is the special case *n*=2. This method opens itself to parallelization in that each of the *n* intervals can all be evaluated independently.

Pseudocode is provided in the section following this analysis.[2]

We can write a recurrence relation to find the time complexity of the bisection algorithm (in terms of interval splits), given an interval of size *k* and acceptable error *e*:

$$S(k) = S\left(\frac{k}{2}\right) + 1, \qquad S(e) = 1$$

Solving gives

$$S(k) = \log_2 k - \log_2 e + 1$$

The parallel *n*-ary section method, given *n* intervals to be computed in one step, will have step complexity given by

$$S(k) = S\left(\frac{k}{n}\right) + 1, \qquad S(e) = 1$$

Solving gives

$$S(k) = \frac{1}{\log_2 n}(\log_2 k - \log_2 e) + 1$$

This suggests that the speedup provided by the parallel version could be logarithmic on the number of intervals used. The implementation I provide runs at *n*=1024 (the number of threads per block available on OSC Oakley's GPUs). As such we can expect a maximum of 10x speedup with this scheme.

Note that parallelization is indeed required for the *n*-ary section method to provide speedup over the sequential bisection method. This is proven by analyzing the steps such a method would require if it were sequential:

$$S(k) = \frac{n}{2} S\left(\frac{k}{n}\right) + 1, \qquad S(e) = 1$$

Solving gives

$$S(k) = \left(\frac{k}{e}\right)^{1-\frac{1}{\log_2 n}} - 2\left(\frac{k}{e}\right)^{-\frac{1}{\log_2 n}} + 2$$

When *n* is substantially large, the $\frac{1}{\log_2 n}$ term tends to zero, and so

$$S(k) \approx \frac{k}{e}$$

This is exponentially worse than the binary method in sequential, and thus parallelization is require for this method to provide any speedup.

Due to the generally low number of steps (32 bits for a maximum of 32 steps required for floating point precision) required by this problem, I have predicted that the overhead due to launching a kernel and communicating with the GPU could be substantial. Therefore, I expect that this solution will be most effective with functions that are expensive to compute. Benchmarks support this prediction in later sections.

Design and optimization

I have implemented both the sequential and parallel versions of the described method. Pseudocode[3] to outline the bisection procedure is as follows

```
Procedure bisection-root-find(function F,
                              interval [a, b],
                              error e):
     if |F(a)-F(b)| < e:
          return F(a)
     end-if

     midpoint <- (a + b) / 2;

     if sign(F(a)) ≠ sign(F(midpoint)):
          return bisection-root-find(F, [a, midpoint], e)
     else
          return bisection-root-find(F, [midpoint, b], e)
     end-if
end procedure
```

The parallel method looks as follows

```
Procedure root-find-parallel(function F,
                              interval [a, b],
                              error e,
                              number of divisions n):
     if |F(a)-F(b)| < e:
          return F(a)
     end-if

     Intervals <- List of size n, splitting [a, b] into
                  equal sized intervals;

     for Intervals[i] in Intervals in Parallel:
          if Intervals[i] is sign-changing:
               nextInterval <- Intervals[i]
          end-if
     end-for
     return root-find-parallel(F, nextInterval, e, n)
end procedure
```

I have implemented both versions of the method with CUDA.

These can be located on my Github page (see end of report).

There are three possible approaches I have considered in implementing the parallel portion of this project:

1. Using only 1 block, have each of *n* threads be responsible for evaluating the function on both ends of an interval.
2. Have each thread only compute the beginning of an interval in order to reduce redundant calculations of method (1), resulting in *n*-1 intervals.
3. Use multiple blocks in order to maximize the number of intervals available and increase occupancy.

Each method has tradeoffs involved. Method 1 redundantly applies the function to each side of the interval twice (the right side of an interval is the left side of another), but is able to minimize communication between threads. Method 2 reduces the redundancies of method 1, but requires shared memory in order to communicate information about adjacent intervals between threads. Method 3 could reduce the total number of steps required, but requires substantial global communication and time waiting for blocks to complete.

Method 2 is unattractive due to the amount of shared memory communication required. There is no apparent benefit to reducing redundant calculations, so this method was not considered for implementation.

As we will see in following benchmarks, we were able to reduce the number of steps involved down to 4 with parallelization. With method 3 this could be reduced to 2 or 3 steps, but that is a rather unimpressive result, considering the costs involved putting barriers between block calls.

For these reasons, I have chosen to use method 1 in the implementation for this project, deeming methods 2 and 3 not worth the effort.

Performance Analysis and Results

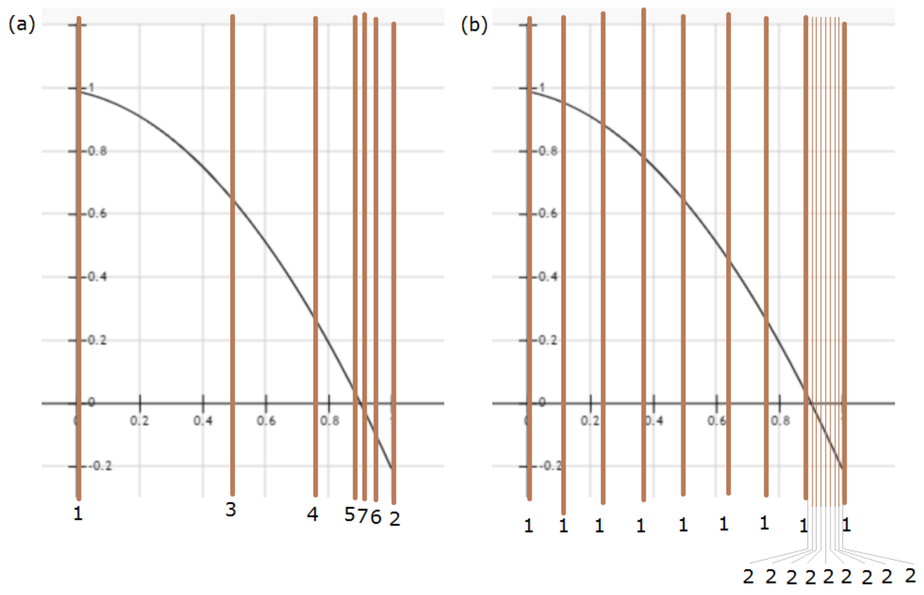
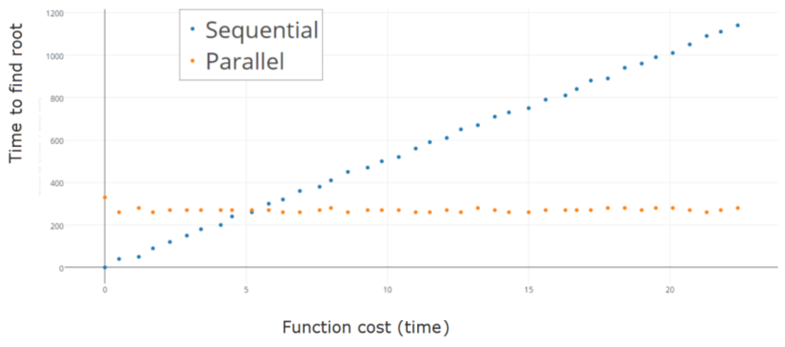


Figure 1.[4] Comparison of (a) bisection method and (b) *n*-ary section method [*n*=8] on an example function. The numbers below each interval boundary indicate during which step the function was computed at that boundary. Note that the bisection method takes three times as many steps to achieve same precision as the *n*-ary method (8th step not show in bisection method).

As discussed above, we can expect some marginal reduction in the number of steps required by the parallel version. An example of how this might work is show in Figure 1 above.

In Figure 2 below, we show benchmarks of both procedures versus the cost of a function.



Sequential: 25 steps
Parallel: 4 steps

Empirical results on

f(x) = 1 – (x + 0.1)^2, 0<x<1

Within floating precision

Figure 2.[5] An expensive function was simulated by spinning a for-loop some number of times and then performing some basic computation. The cost of the function was measured by trial and the time required to find the root with each method is also measured.

Note that in Figure 2, the parallel version does in fact increase with function cost. Performing benchmarks on higher costs does demonstrate this, but running enough trials to get good results on the sequential version simply takes too long at this point.

It should also be noted that for less expensive functions, the overhead of launching a kernel and communicating data between the host and device is simply too much for the parallel version to provide substantial speedup. The turning point occurred around an empty for-loop that spins one hundred times, which is more expensive than many typical math functions might be expected to use (most of those use lookup tables anyways).

As predicted, we can conclude that the *n*-ary section method will only be useful for finding roots of particularly expensive functions and when alternative root finding solutions cannot be reasonably applied to the problem at hand.

Future work

Many options exist to further investigate the use of parallel computing in this area of problems. Due to the requirement that the above solution only provides respectable speedups for substantially expensive functions, I propose three areas of work related to this problem:

1. Investigate applying the above solution to a multi-core system, and use the problem to find parameters generating equilibrium points in intense simulations of physical systems.
2. Instead of working on parallelizing root-finding methods, work on parallelizing the expensive functions themselves – e.g., for a simulation, work on parallelizing the simulation instead of the root-finder.
3. Investigate how parallelization can be used to improve root finding on different classes of functions (e.g., Durand-Kerner method on polynomials).

Because I found it interesting, I will briefly discuss root finding solutions on polynomials. The following method is due to Weierstrass and later investigated by Durand and Kerner.[6]

Consider a polynomial with complex roots $c_1,\ c_2,\ \dots,\ c_n$, written as a product of binomials

$$P(x) = (x - c_1)(x - c_2)\dots(x - c_n)$$

We can solve for the *i*th root, c_i

$$c_i = x - \frac{P(x)}{(x - c_1)(x - c_2)\dots(x - c_{i-1})(x - c_{i+1})\dots(x - c_n)}$$

It turns out that if we simply choose random complex guesses for each c_i (call these z_i), we can repeatedly compose the above equation on those guesses and obtain a new guess

$$z_i \mapsto z_i - \frac{P(z_i)}{(z_i - z_1)(z_i - z_2)\dots(z_i - z_{i-1})(z_i - z_{i+1})\dots(z_i - z_n)}$$

It has been proven that, given initial guesses with a few restrictions, this method will always converge with stability (i.e., even with large degrees, there will not be runaway error as produced by other methods).

This method is attractive in that it is readily open to parallelization. Each next guess for each root is only dependent on guesses for each root from the *previous* iteration, meaning that each of *n* guesses can be computed in parallel.

As described by Cochran[6], the initial guesses must be chosen to have magnitude less than the largest magnitude coefficient in the polynomial, but the guesses will converge most reliably when the initial guesses are as spaced out as possible. As such, a parallel reduce can be used to identify this maximum.

Two phases of this algorithm can therefore be parallelized:

1. Choosing initial guesses – a max reduce followed by a map can help choose initial guesses in time logarithmic on the polynomial degree, rather than linear in sequential.
2. Computing the roots – iterative refinement of the guesses in parallel can discover the roots linearly on the number of steps needed to converge and independent of the polynomial degree, rather than linear on both the number of steps to converge and linear on the polynomial degree in sequential.

For the scope of this project, I did not include an implementation of this method, but I do suggest it as something that could be explored in the future. This concludes the project.

Division of work
This project was completed by myself so division of work amongst a team did not cause any issues. My work schedule went as the following patter:

1. Create project proposal
    a. Perform complexity analysis
    b. Consider design challenges and hypothesize results
2. Implement the proposed solution, optimizations considered
3. Run tests and benchmarks
4. Analyze results as compared to hypothesis
5. Consider future work

References

1. Burden, R. Faires, J.D. *Numerical Analysis*. 9th ed. 48p https://books.google.com/books?id=KlfrjCDayHwC&lpg=PP1&dq=Douglas%20Faires%20Bisection&pg=PA48#v=onepage&q&f=false. [cited 2015-12-06]

2. The analysis in the complexity section is adapted directly from my project proposal

3. The pseudocode is adapted directly from my project proposal

4. http://fooplot.com/ — used to generate graph

5. https://plot.ly/ — used to create scatterplot

6. Cochran W. *The Durand-Kerner Method*. http://ezekiel.vancouver.wsu.edu/~cs330/projects/dk/DK-method.pdf. [cited 2015-12-06] (English description of Weierstrass' method provided by Dr. Cochran)

Code appendix

Code, along with documentation and build instructions, provided on my Github page: *https://github.com/scholtes/ParallelFinal*

Presentation slides

Provided on Blackboard