A Parallel Implementation of the Bisection Algorithm to Optimize Calculations for a Portfolio of Bonds

Anjali Smith (as6467)

December 22, 2022

1 Introduction

Bonds are fixed income securities that investors can purchase and trade. In order to determine a bond's value in relation to other bonds, investors often look at a bonds' yield to maturity, which is a measure of the bond's total anticipated return if the bond is kept until maturity and all payments are reinvested at the same rate. A bond's yield to maturity can be calculated in many different ways, but a common technique that is used to approximate a bond's yield to maturity if the bond's price, par value, and coupon rate is known, is by using a root finding algorithm to solve for the variable YTM in the following formula that draws a relationship between a bond's price and a bond's yield to maturity:

$$BondPrice = \frac{\frac{Coupon}{2}}{(1+\frac{YTM}{2})^{2t_1}} + \frac{\frac{Coupon}{2}}{(1+\frac{YTM}{2})^{2t_2}} + ... + \frac{\frac{Coupon}{2} + Parvalue}{(1+\frac{YTM}{2})^{2t_n}}$$

In my project, I optimize yield to maturity calculations for a portfolio of semi-annual compounding bonds by developing a parallel implementation of the root-finding bisection algorithm which solves for the root of the following equation:

$$0 = \frac{\frac{Coupon}{2}}{(1+\frac{YTM}{2})^{2t_1}} + \frac{\frac{Coupon}{2}}{(1+\frac{YTM}{2})^{2t_2}} + ... + \frac{\frac{Coupon}{2} + Parvalue}{(1+\frac{YTM}{2})^{2t_n}} - BondPrice$$

2 Bisection Algorithm

The bisection algorithm approximates the root x_0 of a continuous one dimensional function $f(x) = 0$ by looking for the value x_0 in an interval, and repeatedly reducing that interval until the root is bracketed. The algorithm is built on the Intermediate Value Theorem, which guarantees that a continuous function f with domain $[a, b]$, can produce any value between $f(a)$ and $f(b)$. My project implements the bisection algorithm in the following manner:

Algorithm 1 : bisection(bottom, top, error)

1: $m = (bottom + top)/2$
2: **if** $sign(f(bottom)) \neq sign(f(top))$ AND $|f(top) - f(bottom)| \leq error$ **then**
3: return m
4: **else if** $sign(f(bottom)) \neq sign(f(middle))$ **then**
5: return bisection(bottom, middle)
6: **else**
7: return bisection(middle, top)
8: **end if**

The three inputs to this recursive function are the lowest value in the interval, the largest value in the interval, and the maximum allowed error for the root approximation.

3 Sequential Implementation

3.1 Test function

My program reads bond data in from a file specified in the command line, parses the data, and runs the yield to maturity calculations on each bond sequentially. My sequential test program tests the accuracy of my approximated yield to maturities by assigning every bond a test yield value and comparing it with the approximated yield to maturity calculation. In order to compare the two values, I used each bond's assigned test yield to calculate the bond's price, and then I used that price to approximate the yield to maturity value with the bisection algorithm. In order to visualize the

comparison, I calculated the squared error for each bond's yield to maturity calculation and returned the errors as a list. The following code is my sequential test function, which performs the steps I described above to return a list of squared errors:

```
runSeq :: [(Bond, Yield)] -> [Double]
runSeq bondTups = map compareYields bondTups

main :: IO ()
main = do
    progName <- getProgName
    args <- getArgs
    let filename : prog = args
        func = case prog of
            [] -> runSeq
            ["seq"] -> runSeq
            ["parMap"] -> runParMapProg
            ["parChunk"] -> runParChunkProg
            _ -> error $ "Usage: " ++ progName ++ "<filename>
                <prog-name>, where <prog-name> = {seq, parMap,
                parChunk}, default to seq"
    bondData <- fmap lines (readFile filename)
    let bondTups = map (parseBondData . (read :: String ->(Double,
        Double, Double, Double))) bondData
    let errors = func bondTups
    print $ sum errors
```

3.2 Sequential bisection function

Through the use of two helper function, bracket and converged, I performed the bisection algorithm with the recursive function rootSolveBracket:

```
{-
bracket checks whether or not the product of f(b)*f(t) has
    opposite signs
-}
```

```
bracket :: (Ord a, Num a) => t -> t -> (t -> a) -> Bool
bracket b t f = ((f b <= 0) && (f t >= 0)) || ((f b >= 0) && (f t
    <= 0))

-- converged checks whether the two intervals have converged or not
converged :: (Ord a, Num a) => t -> t -> (t -> a) -> a -> Bool
converged b t f err = bracket b t f && (abs(f t - f b) <= err)

{-
rootSolveBracket executes the bisection algorithm to approximate
    the root of the input function "f" with an error value equal to
    the input error "err"
-}
rootSolveBracket :: (Ord a, Num a, Fractional t) => t -> t -> (t
    -> a) -> a -> t
rootSolveBracket bot top f err =
    go bot top
    where go b t =
          let m = (b + t)/2
          in if converged b t f err then m
             else if bracket b m f then go b m
                                   else go m t
```

4 Parallel Implementation

I added two main parallel layers to my program: one in the main function and one in the bisection algorithm.

4.1 Parallel yield to maturity calculations

My first parallel layer was similar to Simon Marlow's use of parallelism in one of his Sudoku solver program where he solves a list of Sudoku puzzles in parallel with parMap. I chose to run the yield to maturity calculations and the yield error calculations on each bond in the portfolio in parallel, rather than sequentially. In order to experiment with different parallel techniques, I tested my program with two different main functions, one that used the function parMap and one that used the function parListChunk with chunk sizes of 10,000:

```
-- Tester with parMap
runParMapProg :: [(Bond, Yield)] -> [Double]
runParMapProg bondTups = runPar $ parMap compareYields bondTups

-- Tester with parListChunk
runParChunkProg :: [(Bond, Yield)] -> [Double]
runParChunkProg bondTups = runEval $ parListChunk 10000 rdeepseq
    (map compareYields bondTups)

main :: IO ()
main = do
    progName <- getProgName
    args <- getArgs
    let filename : prog = args
        func = case prog of
            [] -> runSeq
            ["seq"] -> runSeq
            ["parMap"] -> runParMapProg
            ["parChunk"] -> runParChunkProg
            _ -> error $ "Usage: " ++ progName ++ "<filename>
                <prog-name>, where <prog-name> = {seq, parMap,
                parChunk}, default to seq"
    bondData <- fmap lines (readFile filename)
    let bondTups = map (parseBondData . (read :: String ->(Double,
        Double, Double, Double))) bondData
    let errors = func bondTups
    print $ sum errors
```

4.2 Parallel bisection function

My second layer of parallelism was added within the rootSolveBracket function which executes the bisection algorithm. I used the functions par and pseq from the module Control.Parallel so that I could calculate the return values of the functions bracket and converged in parallel:

```
-- bracket checks whether or not the product of f(b)*f(t) has
    opposite signs
bracket :: (Ord a, Num a) => t -> t -> (t -> a) -> Bool
```

```
bracket b t f = ((f b <= 0) && (f t >= 0)) || ((f b >= 0) && (f t
    <= 0))

-- converged checks whether the two intervals have converged or not
converged :: (Ord a, Num a) => t -> t -> (t -> a) -> a -> Bool
converged b t f err = bracket b t f && (abs(f t - f b) <= err)

rootSolveBracket :: (Ord a, Num a, Fractional t) => t -> t -> (t
    -> a) -> a -> t
rootSolveBracket bot top f err =
    go bot top
    where go b t =
            check1 `par` check2 `pseq`
            if check1 then m
              else if check2 then go b m
                                  else go m t
               where
                   !m = (b + t)/2
                   !check1 = converged b t f err
                   !check2 = bracket b m f
```

5 Performance

5.1 Settings

I performed my results on a Macbook Air(13.3 inch, 2018) with a 1.6GHz dual-core Intel Core i5 processor with 8GB of 2133MHz LPDDR3 onboard memory.

5.2 Testing

I tested my program with three datasets: one with a total of 55,000 bonds and corresponding test yields, one with a total of 110,000 bonds and corresponding test yields, and one with a total of 550,000 bonds and corresponding test yields. The datasets were generated by my writeBonds function which writes a combination of bond data to my file. These data values were loosely based off of the data values for bonds listed on the Financial Industry Regulatory Authority's Market Data Center:

```
-- Generates 55,000 bonds and corresponding test yields
writeBondData :: FilePath -> IO ()
writeBondData f = writeFile f $ unlines $ [show (100 :: Parval, c
    :: Coupon, m :: TimeToMat, yield :: Yield) | x <- [1..50], let
    c = x/5, y <- [1..100], let m = y/3, yield <- [0..10]]
```

5.3 Results of parallel implementation (parMap and parListChunk)

Tables 1, 2, and 3 show the runtime of my parallel implementation with the main function that uses parMap on the three different datasets where N = number of cores. Tables 4, 5, and 6 show the runtime of my parallel implementation with the main function that uses parListChunk on the three different datasets where N = number of cores.

N	time(s)	total	converted	dud	gc'd	overflowed	fizzled
1	12.430	1769268	0	1769268	0	0	0
2	7.706	1769268	0	1769268	0	0	0
3	7.024	1769268	0	1769268	0	0	0
4	8.415	1769268	0	1769268	0	0	0

Table 1: Results of parallel implementation (parMap) with 55k bonds

N	time(s)	total	converted	dud	gc'd	overflowed	fizzled
1	27.308	3580474	0	3580474	0	0	0
2	18.458	3580474	0	3580474	0	0	0
3	18.206	3580474	0	3580474	0	0	0
4	18.415	3580474	0	3580474	0	0	0

Table 2: Results of parallel implementation (parMap) with 110k bonds

N	time(s)	total	converted	dud	gc'd	overflowed	fizzled
1	159.301	18648904	0	18648904	0	0	0
2	122.394	18648904	0	18648904	0	0	0
3	157.747	18648904	0	18648904	0	0	0
4	110.843	18648904	0	18648904	0	0	0

Table 3: Results of parallel implementation (parMap) with 550k bonds

N	time(s)	total	converted	dud	gc'd	overflowed	fizzled
1	16.731	1769279	0	1769268	0	0	11
2	8.750	1769279	11	1769268	0	0	0
3	10.094	1769279	11	1769268	0	0	0
4	14.417	1769279	11	1769268	0	0	0

Table 4: Results of parallel implementation (parListChunk) with 55k bonds

N	time(s)	total	converted	dud	gc'd	overflowed	fizzled
1	26.108	3580496	0	3580474	0	0	22
2	20.937	3580496	22	3580474	0	0	0
3	20.995	3580496	22	3580474	0	0	0
4	23.797	3580496	22	3580474	0	0	0

Table 5: Results of parallel implementation (parListChunk) with 110k bonds

N	time(s)	total	converted	dud	gc'd	overflowed	fizzled
1	148.461	18649014	0	18648904	0	0	110
2	130.024	18649014	110	18648904	0	0	0
3	131.334	18649014	110	18648904	0	0	0
4	162.820	18649014	110	18648904	0	0	0

Table 6: Results of parallel implementation (parListChunk) with 550k bonds

5.4 Parallel versus Sequential

Table 7 compares the difference between the runtimes of both of the parallel implementations with the runtime of the sequential solution with all three datasets.

Implementation	Time(s)	Time(s)	Time(s)
	55K bonds	110K bonds	550K bonds
Parallel(parMap)	7.706	18.458	122.394
Parallel(parListChunk)	8.750	20.937	130.024
Sequential	12.584	32.393	174.420

Table 7: Results of parallel implementations compared with sequential implementation ($N = 2$)

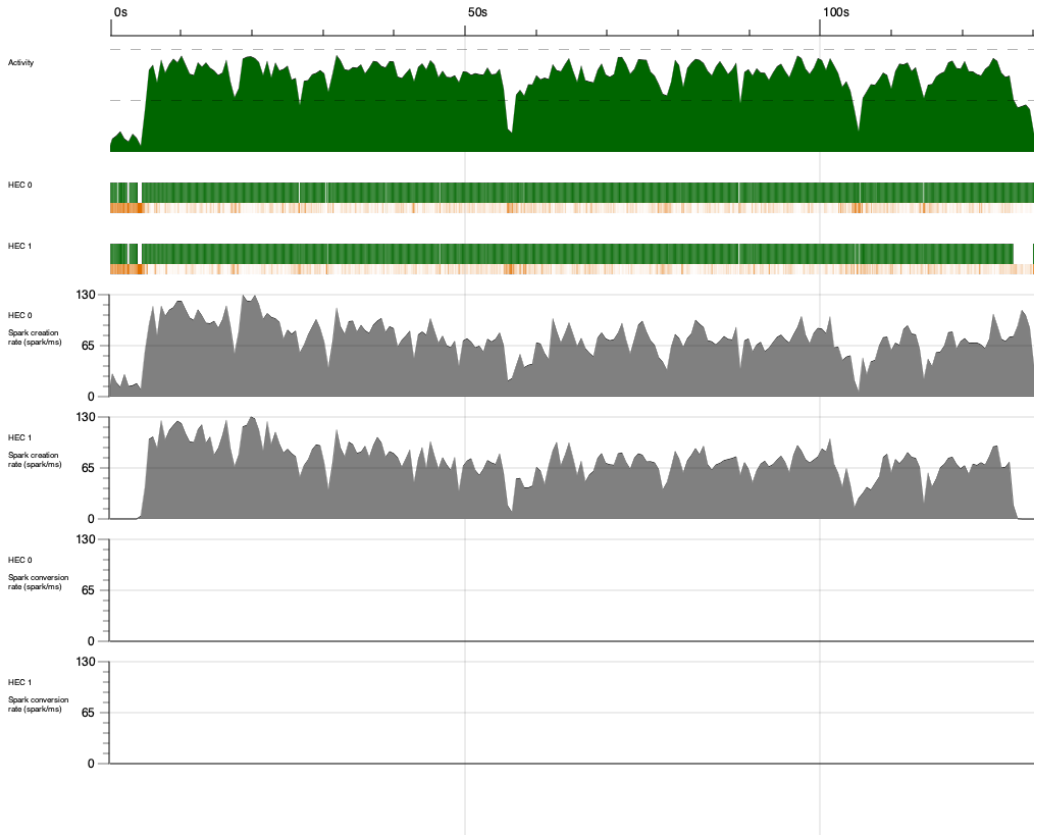
6 Performance Analysis

My parallel implementations ran faster than my sequential implementation. In particular, the parallel implementation with the use of the parMap function in the main method ran slightly faster than both the sequential solution and the parallel implementation with the use of the parListChunk function. My program ran the fastest when $N = 2$. The average speed up from the parallel implementation that used the parMap function in main across the three datasets (when $N = 2$) is:

$$\frac{174.420}{122.394} + \frac{32.393}{18.458} + \frac{12.584}{7.706} \approx 1.6$$

When analyzing the spark data and ThreadScope's spark event visualization, it is clear that either the majority or all of the created sparks in both of my parallel implementations are dud. Focusing on the parallel implementation with parMap in the main function, when I ran this program with a dataset of 550,000 bonds, my program's Eventlog on ThreadScope displays the rate of spark creation in gray indicating all dud sparks. The results displayed in the earlier tables confirm this, since we see that of the total 18,648,904 created sparks, all 18648904 sparks were dud. Even with the results of the parallel implementation that used parListChunk, the majority of created sparks were dud with a small minority being converted.

The large number of sparks that are dud indicate that my program is creating sparks for values that have already been evaluated. I tested the parallel implementation with parMap on a single bond to see the number of sparks created per bond and found that one bond creates 28 sparks, with



0 being converted and all 28 being dud. Since for each bond, the bisection function runs recursively multiple times to approximate the root, I believe that the large number of sparks are being created to evaluate the checks during each recursive call to the bisection algorithm.

Perhaps many of these sparks are dud because they have already been evaluated in previous recursive calls. For example, the parallel bisection algorithm calculates the return values of the converged and bracket functions during every recursive call. Within both of these functions, the value of $f(bottom)$ and $f(top)$ are calculated, which may be one cause for created sparks being dud since the sparks to evaluate those function calls had already been evaluated earlier. Looking at just the bracket and converged functions, I see that there are multiple repetitions of "f b" and " f t" in both functions:

```
-- bracket checks whether or not the product of f(b)*f(t) has
    opposite signs
```

```
bracket :: (Ord a, Num a) => t -> t -> (t -> a) -> Bool
bracket b t f = ((f b <= 0) && (f t >= 0)) || ((f b >= 0) && (f t
    <= 0))

-- converged checks whether the two intervals have converged or not
converged :: (Ord a, Num a) => t -> t -> (t -> a) -> a -> Bool
converged b t f err = bracket b t f && (abs(f t - f b) <= err)
```

Since both of these functions are running in parallel with the following lines:

```
check1 `par` check2 `pseq`
```

where check1 and check 2 are:

```
!check1 = converged b t f err
!check2 = bracket b m f
```

the sparks created to evaluate each check might have found the values of "f t" and "f b" already calculated, resulting in the spark being dud.

7 Potential Improvements

With more time, I would improve the parallel bisection algorithm to split each interval into more than 2 sub-intervals, and run the recursive function calls on each interval in parallel. I would also experiment with different chunk sizes in the main function to determine the optimal chunk size because when I was testing, I discovered that different chunk sizes increased the conversion rate of the sparks and decreased the number of dud sparks. In addition, as Simon Marlow describes in chapter 3 of his book, *Parallel and Concurrent Programming in Haskell*, chunking a list directly can cause overhead so it would be more efficient to prechunk the list once and then feed those chunks in to the algorithm, and use a vector instead of a list since slicing vectors run in $O(1)$ time. I would also want to add a parallel element to my function getCashFlows which generates a list of cash flows for a bond since currently, each calculation of the bond's cash flow is being done sequentially.

8 References

1. Fernando, J. (2022, December 19). Yield to maturity (YTM): What it is, why it matters, formula. Investopedia. Retrieved December 21, 2022, from https://www.investopedia.com/terms/y/yieldtomaturity.asp

2. Haskell. (2017, May 7). ThreadScopeTour/SPARK2. HaskellWiki. Retrieved December 21, 2022 from https://wiki.haskell.org/ThreadScopeTour/Spark2

3. Weisstein, Eric W. "Bisection". MathWorld.

4. Yang, X., amp; Liu, Z. (2019, December 16). Parallel Functional Programming Parallel PageRank with MapReduce. COMS 4995 Parallel Functional Programming. Retrieved December 21, 2022, from http://www1.cs.columbia.edu/ sedwards/classes/2019/4995 fall/reports/pagerank.pdf

9 Code Listing

9.1 Overview

The YTM-Project directory has four files and a subdirectory named data with one dataset for testing. In YTM-Project, there is the Bond.hs file with the Bond data type and related functions, the ParBis.hs and SeqBis.hs which have my code for my parallel and sequential implementations, and my test file ytm-project-test.hs which has the main function. Please refer to the README.txt file in the YTM-Project directory for instructions on how to run my program with stack.

9.2 Ytm-project-test.hs

```
import ParBis
import SeqBis
import Bond
import System.Environment(getArgs, getProgName)

main :: IO ()
main = do
```

```
    progName <- getProgName
    args <- getArgs
    let filename : prog = args
        func = case prog of
            [] -> runSeq
            ["seq"] -> runSeq
            ["parMap"] -> runParMapProg
            ["parChunk"] -> runParChunkProg
            _ -> error $ "Usage: " ++ progName ++ "<filename>
                <prog-name>, where <prog-name> = {seq, parMap,
                parChunk}, default to seq"
    bondData <- fmap lines (readFile filename)
    let bondTups = map (parseBondData . (read :: String ->(Double,
        Double, Double, Double))) bondData
    let errors = func bondTups
    print $ sum errors
```

9.3 Bond.hs

```
{-# LANGUAGE BangPatterns #-}
module Bond where

import Control.DeepSeq ( NFData(..) );
-------------------------------------
-- Bond related types and functions
-------------------------------------

data Bond = Bond !Parval !Coupon !TimeToMat deriving (Eq, Ord,
    Show)
data CashTime = CashTime !Cash !Double deriving (Eq, Ord, Show)

instance NFData CashTime where
    rnf (CashTime c t) = rnf c `seq` rnf t

instance NFData Bond where
    rnf (Bond p c t) = rnf p `seq` c `seq` rnf t
```

```
type Parval = Double
type Coupon = Double
type TimeToMat = Double
type Cash = Double
type Yield = Double

-- appCoup takes a bond and returns the bond's semi-annual
   interest income
appCoup :: Bond -> Double
appCoup (Bond p c _) = (c/100)*p

-- getCashFlows takes a bond and returns a list of semi-annual
   interest payments for the bond
getCashFlows :: Bond -> [CashTime]
getCashFlows b@(Bond p _ m) =
    let nt = round $ m*2 :: Integer
        coup = appCoup b/2 in
            map (\i -> let c = if i==nt then p + coup else coup in
                CashTime c (fromIntegral i*0.5)) [1..nt]


-- yieldToPrice takes a bond and a yield, and returns the present
   value of the bond
yieldToPrice :: Bond -> Yield -> Cash
yieldToPrice b y = presValSemiCashs y $ getCashFlows b

-- presValSemiCashs takes a bond's yield and list of cash flows
   (also known as a list of the bond's semi-annual interest
   payments), and returns the present value of the bond's stream
   of payments
presValSemiCashs :: Yield -> [CashTime] -> Cash
presValSemiCashs y cf = sum $ map (presentValSemi y) cf

-- presentValSemi takes a bond's yield and a single semi-annual
   interest payment, and returns the bond's present value at that
   point in time
presentValSemi :: Yield -> CashTime -> Cash
presentValSemi y (CashTime c t) = c/((1 + (y/200))**(2*t))
```

```
--parseBondData turns data read in from a file into a tuple with a
    bond and a test yield
parseBondData :: (Double, Double, Double, Double) -> (Bond, Yield)
parseBondData (p, c, m, y) = (Bond p c m, y)
```

9.4 SeqBis.hs

```
{-# LANGUAGE BangPatterns #-}

module SeqBis where

import Bond;


-------------------------------------------
-- Functions for the bisection algorithm
-------------------------------------------

-- bracket checks whether or not the product of f(b)*f(t) has
    opposite signs
bracket :: (Ord a, Num a) => t -> t -> (t -> a) -> Bool
bracket b t f = ((f b <= 0) && (f t >= 0)) || ((f b >= 0) && (f t
    <= 0))

-- converged checks whether the two intervals have converged or not
converged :: (Ord a, Num a) => t -> t -> (t -> a) -> a -> Bool
converged b t f err = bracket b t f && (abs(f t - f b) <= err)

-- rootSolveBracket executes the bisection algorithm to
    approximate the root of the input function "f" with an error
    value equal to the input error "err"
rootSolveBracket :: (Ord a, Num a, Fractional t) => t -> t -> (t
    -> a) -> a -> t
rootSolveBracket bot top f err =
    go bot top
    where go b t =
          let m = (b + t)/2
          in if converged b t f err then m
```

```
            else if bracket b m f then go b m
                                  else go m t

-----------------------------------------
-- Testing with bond data from input file
-----------------------------------------

-- compareYields compares a test yield and the calculated yield
   for a bond by generating its squared error
compareYields :: (Bond, Yield) -> Double
compareYields (b, yog) = calcErr (b, p, yog)
  where p = yieldToPrice b yog

-- calcErr returns the squared error for a bond's calcualted YTM
   and a test yield
calcErr :: (Bond, Cash, Yield) -> Double
calcErr (b, c, y) = (y - priceToYield (b,c))**2.0

-- priceToYield takes a tuple with a bond and a price and returns
   the bond's yield to maturity
priceToYield :: (Bond, Cash) -> Yield
priceToYield (b, p) =
   let f y = yieldToPrice b y - p
   in rootSolveBracket (-100) 100 f 0.000001

-------------------------------------------------------------------------------
-- Sequential run of the program
-------------------------------------------------------------------------------
runSeq :: [(Bond, Yield)] -> [Double]
runSeq bondTups = map compareYields bondTups
```

9.5 ParBis.hs

```
{-# LANGUAGE BangPatterns #-}

module ParBis where

import Bond;
```

```
import Control.Parallel;
import Control.Parallel.Strategies hiding (parMap);
import Control.Monad.Par;

-----------------------------------------
-- Functions for the bisection algorithm
-----------------------------------------

-- bracket checks whether or not the product of f(b)*f(t) has
    opposite signs
bracket :: (Ord a, Num a) => t -> t -> (t -> a) -> Bool
bracket b t f = ((f b <= 0) && (f t >= 0)) || ((f b >= 0) && (f t
    <= 0))

-- converged checks whether the two intervals have converged or not
converged :: (Ord a, Num a) => t -> t -> (t -> a) -> a -> Bool
converged b t f err = bracket b t f && (abs(f t - f b) <= err)

-- rootSolveBracket executes the bisection algorithm to
    approximate the root of the input function "f" with an error
    value equal to the input error "err"
rootSolveBracket :: (Ord a, Num a, Fractional t) => t -> t -> (t
    -> a) -> a -> t
rootSolveBracket bot top f err =
    go bot top
    where go b t =
            check1 `par` check2 `pseq`
            if check1 then m
              else if check2 then go b m
                                  else go m t
               where
                   !m = (b + t)/2
                   !check1 = converged b t f err
                   !check2 = bracket b m f

-----------------------------------------
-- Testing with bond data from input file
-----------------------------------------
-- priceToYield takes a tuple with a bond and a price and returns
    the bond's yield to maturity
```

```
priceToYield :: (Bond, Cash) -> Yield
priceToYield (b, p) =
    let f y = yieldToPrice b y - p
    in rootSolveBracket (-100) 100 f 0.000001

-- compareYields compares a test yield and the calculated yield
    for a bond by generating its squared error
compareYields :: (Bond, Yield) -> Double
compareYields (b, yog) = calcErr (b, p, yog)
  where !p = yieldToPrice b yog

-- calcErr returns the squared error for a bond's calcualted YTM
    and a test yield
calcErr :: (Bond, Cash, Yield) -> Double
calcErr (b, c, y) = (y - yt)**2.0
  where !yt = priceToYield (b,c)

-------------------------------------------------------------------------------
-- Two different tests: parallel implementations with parMap and
    parListChunk
-------------------------------------------------------------------------------

runParMapProg :: [(Bond, Yield)] -> [Double]
runParMapProg bondTups = runPar $ parMap compareYields bondTups

runParChunkProg :: [(Bond, Yield)] -> [Double]
runParChunkProg bondTups = runEval $ parListChunk 5000 rdeepseq
    (map compareYields bondTups)
```