CSE341T/CSE549T

Homework 1

Assigned: 09/08/2014 *Due Date: 09/22/2014*

In this assignment, you will implement three solutions to the stock market problem and perform some analysis of your solutions. Finally, you will do some exercises involving work-span analysis.

Submission

This assignment requires that you submit both code and written answers.

Your written answers must be printed out and submitted in class on the day the assignment is due.

The coding portion of the assignment is distributed in a number of files in our `svn` repository. Perform an `svn update` in the directory where you have checked out your repository and you will see the directory `hw01`. Inside this directory, you will see the following files in it:

- `StudentSolution.cpp`: **the only file that should contain your solution for this coding assignment**
- `Stocks.cpp`: where the main lives. This is the code that calls your 3 methods.
- `cilktime.h`, `Timer.h`, `Timer.cpp`: files used to time and compare the speed of your 3 algorithms
- `StocksResult.h`, `StocksResult.cpp`: class that holds the result of running your algorithm (when to buy, when to sell, what the gain is)
- `Makefile`, `StocksSupport.cpp`: code you should be glad we wrote for you
- `input1`, `input2`, `input3`: sample input files of increasing sizes. the first line of the file indicates the number of lines of the file. For example, if the first line is "5", a correctly formatted file will have 5 more lines, each with a stock price.

Your assignment is to complete the 3 methods in `StudentSolution.cpp`. While you can modify other files that we provided, `StudentSolution.cpp` is the only file that should contain your solutions. The autograder will copy only `StudentSolution.cpp` during autograding, so any modifications you make to other files will be discarded. Your code must compile cleanly. Make sure that the three methods maintain their original names and signatures; if your method has the wrong name or signature, you will not receive credit for your work.

Problem 1-1. The Stock Market Problem[70 points]

The problem with the stock market is that, while it is possible to make a great deal of money buying and selling stocks, it's easy to lose even more. The long-standing—if somewhat unhelpful—maxim to make more money than you lose is "buy low, sell high."

The stock market problem is finding the best opportunity to follow this advice: for any sequence of integer prices, where the index in the sequence represents time, find maximum jump from an earlier price to a later price. For example, if the sequence of prices was

$$\langle 40, 20, 0, 0, 0, 1, 3, 3, 0, 0, 9, 21 \rangle$$

then the maximum jump is 21, which happens between the price at time 2 and time 11. More formally, the stock market problem is to compute

$$\max\{s_j - s_i | 0 \leq i \leq j < |s|\}$$

Note that this maximum is only well defined if there is at least one element in s. If this is not the case, your code will exit with an error (*"Sequence Length must be > 1"*).

In this problem, you will design, analyze and implement your solutions. The design and analysis portions of the homework should be turned in as writeups. You will implement three solutions to this problem in the file `StudentSolution.cpp`. Each solution will be a method already defined in `StudentSolution.cpp`. Each function has the following signature:

```
void solveStocksXXX(const vector<int> &stockSequence, int
    beginIndex, int seqLength, StocksResult *result)
```

The four arguments are:

- `stockSequence`: the sequence itself (a vector of integers). Note 1: your algorithms will inspect the `stockSequence`, but will not have to change it (hence it remains a `const`). You will use a reference to the same single vector for the entire life of the program. Note 2: to access, for example, the price of stock at time `i`, you simply type `stockSequence[i]`
- `beginIndex`: at times, you will want to inspect only a portion of the complete sequence. Use this argument to specify the beginning of the subset of the sequence that you are currently interested in.
- `seqLength`: use this argument to specify the length of the sequence that you are currently interested in.
- `result`: the structure in which you should store your solution.

The class `StockResult` has been defined for you. It has 5 integer member variables:

- `bestBuy`: the best price at which to buy
- `bestSell`: the best price at which to sell
- `jump`: the best gain in the stock sequence (`bestSell` - `bestBuy`)
- `low`: the lowest stock price in the sequence

- `high`: the highest stock price in the sequence

A correct answer will have the first three fields correctly populated. The final two fields of the `StockResult` class (`low` and `high`) are for you to use as you choose, but they are not printed out as part of the solution to the problem.

Compiling your code. Your code can be compiled by typing "`make Stocks`" into the command line.

Running your code. Your code can be run with many possible arguments. Here are just a few examples:

`./Stocks -help`	shows all possible arguments for customizing your run
`./Stocks`	runs on a randomly generated stock sequence
`./Stocks -len 100 -max 500`	runs on a randomly generated stock sequence of length 100 with a maximum stock value of 500
`./Stocks -f input1`	runs with the input file `input1`
`./Stocks -s 10`	runs on a randomly generated stock sequence using 10 as the initial seed.

Testing Your Code. Be sure to test your code for correctness with a variety of sequence lengths and examples to convince yourself that your implementations are correct.

Three input files of stock prices `input1-3` are provided for you in the `hw01` directory. Because the random number generator in the code always has the same seed, the randomly generated numbers will always be the same. This may help you in your debugging.

Feel free to create new input files, or randomly generate sequences of different lengths with different seeds. We will run your student solution against our reference solution and the solutions must match in order to get full credit. Also, the timing of your algorithms should not be off by too much.

(a) Naïve Implementation[10 points]

Implement a quadratic work brute-force solution to the stock market problem in a function called `solveStocksNaive()`. (Note: if you run the `Stocks` program on an input sequence of length larger than 10,000, the program will not call the quadratic version of your code. This is in fairness to other users of the machine and to prevent the auto-grader from dying a long and painful death.)

(b) Divide and Conquer [20 points]

Solve the stock market problem by divide-and-conquer recursive programming. Describe your solution in the written portion of your assignment and calculate the asymp-

totic running time. Then implement your solution in the function called `solveStocksRecurse()`. Run your solution and make sure that it is indeed a faster recursive solution.

(c) Analysis [10 points]

Use mathematical induction on the length of the input sequence to prove that your divide and conquer implementation is correct. Be sure to carefully state the theorem that you're proving and to note all the steps in your proof. Remember that a longer proof is not necessarily a more correct proof.

(d) Parallelize your divide and conquer implementation [20 points]

Parallelize your recursive implementation. Explain the implementation in your written handin, calculating the work and the span of your implementation. Draw the DAG for the solution for a small problem size. Implement it by making use of the Cilk Plus keywords `cilk_spawn` and `cilk_sync`. Write your implementation in a function called `solveStocksRecPar()`. Can you think of ways to make the solution faster? If so, suggest them in your writeup and also implement them in your code.

(e) Measure the work, span, and parallelism of your parallel divide and conquer implementation [5 points]

Include in your written handin the values that you see for the work, span, and parallelism (not the burdened one) reported by `cilkview` under the "Cilk Parallel Region(s) Statistics" section. Report the values for a small input (e.g., -len 100) and a large input (e.g., -len 10000), and explain the parallelism you saw.

In `cilkview`, you can use the provided annotations to ask the tool to selectively measure the work and span of a particular region. We have annotated `Stocks.cpp` to include the annotations (`__cilkview_start/stop`), but they are commented out right now. Uncomment the annotations and invoke `cilkview` (i.e., `cilkview -- ./Stock <input>`) to get these measurements.

(f) Parallel for loop? [5 points]

Recall that a for loop can be parallelized in Cilk Plus using the keyword `cilk_for`. A Cilk Plus for-loop can run each iteration of the loop in parallel. Consider your initial naive implementation of the stock market solution. Is it advisable to use a Cilk Plus for-loop in this solution? If so, where you would use the Cilk Plus keyword? What is the work and span of this program? If not, explain why you would not do so.

Problem 1-2. Asymptotics [10 points]

Carefully prove or disprove each of the following statements. Remember that verbose proofs are not necessarily careful proofs: your answers for these questions will likely fit on one page.

1. O is a transitive relation on functions. That is to say, for any functions f, g, h:

$$f \in O(g) \wedge g \in O(h) \implies f \in O(h)$$

.

2. O is a symmetric relation on functions. That is to say, for any functions f and g:

$$f \in O(g) \implies g \in O(f)$$

3. O is an anti-symmetric relation on functions. That is to say, for any functions f and g:

$$(f \in O(g) \wedge g \in O(f)) \implies f = g$$

Problem 1-3. Work/Span Analysis of parallel_for [20 points]

As an exercise, you were supposed to analyze the work and span the following code that uses a parallel for loop. In class, we saw that we can convert a parallel for loop into code that just uses spawns and syncs using a divide and conquer algorithm. This creates an additive span of $O(\lg n)$. You will now analyze two other variants for parallelizing this loop.

(a) An Intel engineer comes to you tells you that they have decided to make a parallel for loop more efficient by doing a divide by 3 divide-and-conquer instead of divide-by-two as you learned in class. Give the pseudocode for this new version of parallel for loop and analyze it. Does this modification improve the work and span asymptotically?

(b) We are now going to coarsen the base case. Analyze the work and span of the following computation in terms of G and n.

```
COPYARRAY(A, B, start, end)
1  if end − start < G
2     then for j ← start to end
3               do B[j] ← A[j]
4          return

5  mid ← ⌊(start + end)/2⌋
6  spawn COPYARRAY(A, B, start, mid)
7  COPYARRAY(A, B, mid + 1, end)
8  return
```