Project 1

Algorithmic Tricks

Out: In class on Thursday, 10 Sept 2009
Due: In class on Thursday, 17 Sept 2009

Last Updated: Saturday, 12 Sept 2009

This project provides you with an opportunity to think about algorithmic tricks that can have a drastic impact on performance. Implementation details can often yield surprising performance benefits, but algorithmic changes can also provide substantial gains; in some cases, you will even be able to improve the asymptotic time complexity of a program. You will also get an opportunity to practice your C programming skills. However, we do not expect you to perform parallelization or hardware-specific optimization—that will come later.

Performance tuning is an iterative process. Many promising ideas will end up reducing the program's performance. Thus, in this and in future projects, we want you to describe all the ideas you tried out—even the ones that did not work out in the end. Try to think about about why the changes you made had the results that they did, especially when those results are surprising to you; this is how you will build intuition about performance-related issues.

Introduction

As Alyssa P. Hacker arrives for her first day of work at Snailspeed Ltd., she is given four programs written by Snailspeed's previous ace programmer, Ivy H. Crimson. Unfortunately, Ms. Crimson has since begun a more personally satisfying career in fried-potato engineering, so it falls to Alyssa to maintain them.

Upon examining Ivy's code, Alyssa finds that while these programs are essentially correct (that is, they produce the intended results), they are not very well-written. While she knows that cleaning up the code and making some small tweaks will probably result in a measurable performance gain, she wants to achieve a drastic performance improvement so that she doesn't wind up flipping burgers, too. Your challenge is to help her do so by taking a completely new approach to each problem (that is, by making algorithmic improvements).

Building the code

In each program sub-directory, you can build the code by typing

```
make
```

Two different binaries will be produced. `program.32` and `program.64` are the 32-bit and 64-bit versions of your program, respectively. Since the `cagfarm-NN` machines are 32-bit, you won't be able to run 64-bit binaries on them. However, the compute nodes that will run the jobs you submit via PNQ are 64-bit, and will be able to run either 32-bit or 64-bit binaries. Therefore, when you need to run 64-bit versions of your code (or 32-bit versions, for comparison purposes) you should use PNQ; otherwise, while you are working, you should be fine running 32-bit binaries on the `cagfarm-NN` machines.

1 Data rotation

Many programs have to operate under tight constraints, and getting respectable performance under these circumstances can often be especially challenging. Part of Snailspeed Ltd.'s product portfolio targets mobile and embedded devices such as cell phones. One such application requires a large data buffer to be rotated. However, this buffer is deliberately as large as possible, and so only a small amount of memory (M bytes) is available for any internal processing required by the `rotate()` function. In the code, this is represented by the variable *tmpBufferSize* (set by command-line argument *temp-buffer-size*).

1.1

Take a look at the `rotate()` function as implemented by Ivy. In her code, given M bytes of temporary storage, how many bytes of data need to be moved when rotating a buffer of length n bytes by an offset of k?

1.2

Run Ivy's program for different buffer lengths and offsets to create a data set that is sufficient to analyze the performance of the program. Can you explain the program's performance? Does the runtime correlate with the number of moves required?

1.3

Design an algorithm that will take fewer moves than Ivy's algorithm does. Explain how it works. How many moves are required by your algorithm?

1.4

Implement your algorithm. Measure and explain the performance characteristics of your new code, and compare and contrast it with Ivy's. If your code implements several improvements, please briefly explain the impact of each one.

1.5

(Optional) Can you further improve performance by restructuring the code for faster execution? Try reordering statements (in cases where ordering doesn't matter) or unrolling loops, among other techniques.

Correctness

A solution will be judged correct if it returns the same result as Ivy's program for any input. The rotated buffer must be printed exactly as Ivy's program prints it. If you add your own output statements (for example, calls to `printf()`) you should make sure that the result which is printed remains on its own line. (Don't worry about newlines within the buffer that is being printed.)

You should also make sure not to modify the code which reports timings. The reported execution time of your program must also appear on its own line.

2 Bit-flip counter

This program counts all of the transitions from 0 to 1 or vice versa in a long sequence of bits. (For example, the bit sequence 000 has no transitions; the sequence 001 has one transition; and the sequence 010 has

two transitions.) The smallest data type in C (and the smallest data type supported by the underlying hardware) is one byte. It appears that Ms. Crimson was struggling to access each bit individually; you'll have to help Alyssa do better.

2.1

Come up with a better way to count bit-flips, and implement it. Explain how your method works.

2.2

Compare the performance of your bit-flip counter with the performance of Ivy's code. Again, if you made several improvements, please briefly explain the impact of each one.

2.3

Can you think of a tradeoff between space complexity and time complexity for this problem? (That is, can you find the answer faster if you allow yourself to use more memory, or can you use less memory if you don't mind taking more time?) These sorts of tradeoffs are relatively common. Describe situations in which one solution or the other might be preferred.

2.4

In the context of the tradeoff described by the previous question, how much can you "trade"? What bounds does reality impose upon the tradeoff?

Correctness

A solution will be judged correct if it returns the same result as Ivy's program for any input. The number of transitions detected must be reported exactly as Ivy's program reports it. If you add your own output statements (for example, calls to `printf()`) you should make sure that the result which is printed remains on its own line.

You should also make sure not to modify the code which reports timings. The reported execution time of your program must also appear on its own line.

3 n-queens problem

In the n-queens problem you must place n pieces (each of which moves like the queen from chess) on an $n \times n$ chess board such that no piece can capture any other. Thus, a solution is an arrangement in which no two queens share the same row, column, or diagonal. Ivy's n-queens program finds all of the valid solutions for a given n. You may assume that $4 \leq n \leq 30$.

3.1

Starting with $n = 4$, measure the performance of Ivy's program. You may stop once a single run takes longer than a minute. Graph and analyze the results.

3.2

Design a faster n-queens solver. Explain how your solver will work, and then implement it.

Hint: It may not be necessary to represent the full $n \times n$ board.

Hint: Can you think of a way to represent multiple booleans with a single integer?

3.3

Compare the performance of your n-queens solver with Ivy's solution. Generate a graph like the one you did for Ivy's program, and present the two next to each other. As always, if you made several improvements, please briefly explain the impact of each one.

Correctness

You may assume that $4 \leq n \leq 30$.

A solution will be judged correct if it returns the same result as Ivy's program for any valid input. The number of possible boards must be reported exactly as Ivy's program reports it. If you add your own output statements (for example, calls to `printf()`) you should make sure that the result which is printed remains on its own line.

In order to be judged correct, your program must also be capable of printing each of the boards.

You should also make sure not to modify the code which reports timings. The reported execution time of your program must also appear on its own line.

4 Anagram finder

The last program that Ivy wrote before leaving Snailspeed, Ltd. for greener, more deep-fried pastures is her anagram finder. Given a list of words, the anagram finder will identify all groups of words which are anagrams of one another.

4.1

Describe how Ivy's program identifies anagrams. Measure its performance.

4.2

Design a better algorithm for finding anagrams, and implement it. Explain how your algorithm works. What fundamental insights is your algorithm based on? (What can you work out about the problem that Ivy apparently didn't?)

Hint: Is it profitable to preprocess the data before you attempt to identify anagram groups?

4.3

Measure the performance of your anagram finder. Compare this with what you measured about Ivy's implementation. As always, if you made several improvements, please separately measure and briefly explain the impact of each one. (We won't always tell you this in the future, but we'll always be expecting it.)

Correctness

You may assume that words will consist only of the characters present in the sample inputs available to you.

A solution will be judged correct if it returns the same result as Ivy's program for any valid input. Neither the order in which anagram pairs are reported nor the order of the words within each pair matters. You should make sure that each pair is reported on its own line and that the two words in each pair are separated by exactly the same delimiter as in the output of Ivy's program.

You should also make sure not to modify the code which reports timings. The reported execution time of your program must also appear on its own line.

6.172 Performance Engineering of Software Systems

Fall 2009