Rishabh Kabra & Anurag Kashyap
10 October 2012

6.172 Project 1: Final Write-Up

Overview

The two critical goals for the final version of the project were (1) increased performance on par with the baseline presented in class and (2) increased code quality. We present the steps we took in trying to meet these goals. In addition, we present some comments on the meetings with our MITPOSSE mentors.

Performance

At submission, our beta performed the provided longrunning rotation in 0.000089 s, which was a 597342x performance boost compared to the original program. The baseline performance announced, however, performed the same run in 0.000063 s, so we needed to **aim for a 1.41x speedup**. Our program spent nearly all of its runtime in two methods: bitarray_reverse_faster and unsigned_long_reverse. Speeding it up was thus a matter of optimizing our overall subarray reversal approach and/or reversing individual (unsigned long) words faster. Here are the new optimization strategies we came up with that worked toward that end:

- *Optimization 1:* using a char pointer to index and fetch each of the 8 bytes in a long; **1.14x speedup** to 0.000078 s. This tactic worked better than masking and bit shifting to fetch a long's bytes, which was necessary in unsigned_long_reverse for reversing longs using a byte-reverse lookup table.
- *Optimization 2:* reducing instructions that calculated leftexcess and rightexcess in bitarray_reverse_faster; **1.04x speedup** to 0.000075 s. Leftexcess and rightexcess were a measure of how many bits in the leftmost and rightmost words of a subarray required reversing and repositioning. We realized we didn't have to calculate leftexcess and rightexcess repeatedly in bitarray_reverse_faster; instead, we structured our loops so that leftexcess and rightexcess would be processed beforehand.

Unfortunately, those were the only two new optimizations that worked for our implementation, and **we fell short of the base line**. We spent a significant amount of time in attempts that were unfruitful toward our goal. To make the breadth of our efforts evident, we list the approaches we tried that weren't successful:

- *Failed Optimization 1:* **Vectorization**, which alone had the potential to place us ahead of our objective. The bulk of our implementation's work was being performed in three loops that corresponded to three different cases of symmetry in bitarray_reverse_faster. We studied the conditions that allowed a compiler to vectorize a loop, and though our loops were countable, didn't branch internally, and made no external function calls, they used non-contiguous memory access, which made vectorization impossible. We couldn't figure a way to implement our algorithm with simplified memory access.
- *Failed Optimization 2*: **Performing O(1) work per rotation**. We thought of augmenting our bitarray representation with a parameter that recorded exactly how the bitarray had been rotated. This parameter could be updated with O(1) work every rotation and then be used as an offset for bitarray_get/set. (For example, if we rotated a bitarray of length 10 by 5 bits, we could set the rotation parameter to be 5. Then, bitarray_get/set would know to get/set the 1st

bit when asked for the 6th one, thereby accounting for the bitarray's rotation.) This strategy was equivalent to using a circular list to store bits so they could be rotated in constant time. However, we asked on Piazza[1] if this approach was legal and were told it is not.

- *Failed Optimization 3:* **Experimenting with the compiler.** It is claimed that icc can be faster that gcc in several scenarios, but that was not the case for our program. We also tried several different combinations of gcc compiler flags to see if any optimized our code better.
- *Failed Optimization 4*: **Using more bit hacks**. Since unsigned long reversal was critical to our implementation, we tried *all* applicable bit reversal hacks described by Sean Eron Anderson[2], but no hack performed better than the one we have currently implemented. We couldn't come up with any faster strategies to replace the bit shifts and bitwise operations in bitarray_reverse_faster either.

Code Quality

- We accurately documented the nature of our program by including specifications for functions and data structures. This documentation provides a high-level overview of our program's functionality and data representation.
- We modified the codebase in light of our code-review comments from Caesar, which helped point out particularly unreadable sections of code (that we documented/commented) or other, more concise ways to implement certain functionality (that we implemented for clarity).
- We removed all legacy code (such as various working versions of functions which unfortunately did not enhance performance).
- We kept our test coverage above 98%.
- We ensured our commit messages were detailed and elaborate so as to maintain a readable, high-quality version log.

A code quality measure that we couldn't undertake: we wanted to modularize our bitarray_reverse_faster routine to increase its readability and isolate sub-functions. However, to divide the code into subroutines, we needed to pass parameters by reference (which, in the scope of our code, meant passing pointers to pointers), and the subroutines would then have to dereference them in every usage instance. Unfortunately, the extensive dereferencing caused a slight performance hit, and we had to forego the modularization in favor of performance.

MITPOSSE Mentor Meetings

Per the course information, MITPOSSE mentors were supposed to "give you concrete advice on your design and code." In this respect, the mentors succeeded. For instance, it was a mentor who helped us realize that our bitarray_rotate_faster routine ought to be modularized. The mentors also provided us knowledge of various C frameworks that could be used to more agilely develop and test our code. All this advice was incredibly helpful and appreciated.

While more specific tips on how to improve performance would have been appreciated, it's clear that the mentors would have to spend a significant amount of time delving into the gritty details of the codebase in order to proffer such advice. This is time they probably cannot afford. So while lack of performance improvement-targeting advice can't be held against them, providing said tips would be much appreciated.

---

[1] https://piazza.com/class#fall2012/6172/127. See the last followup.

[2] Bit Twiddling Hacks. http://graphics.stanford.edu/~seander/bithacks.html.