Project 1: Bit Hacks

Last Updated: September 11, 2018

This project provides you with an opportunity to learn how to improve the performance of programs using the perf tool and to experiment with ***word-level parallelism****: the abstraction of a computer word as a vector of bits on which bitwise arithmetic and logical operations can be performed.*

Word-level parallelism — more colloquially called ***bit hacks*** *— can have a dramatic impact on performance. This project will also give you the opportunity to develop and practice your C programming skills.*

Generally, when you are concerned about the performance of a program, the best approach is to implement something correct and then evaluate it. In some cases, many parts of this initial implementation (or even the entire thing) may be adequate for your needs. However, when you need to improve performance, you must first decide where to focus your efforts. This is where profiling becomes useful. Profiling can help you identify the performance bottlenecks in your program.

1 Due dates

- □ *Team formation:* 7:00 P.M. on Thursday, September 13, 2018
- □ *Team contract:* 11:59 P.M. on Friday, September 14, 2018
- □ *Beta release with 20 tests:* 11:59 P.M. on Wednesday, September 19, 2018
- □ *Progress reports:* weekly, 7:00 P.M. every Thursday, starting September 20, 2018
- □ *Beta write-up:* 11:59 P.M. on Thursday, September 20, 2018
- □ *Final release:* 11:59 P.M. on Monday, October 1, 2018
- □ *Final write-up:* 11:59 P.M. on Tuesday, October 2, 2018

2 Getting started

On your second week of work at Snailspeed Ltd., your boss asks you build a faster version of a program sold by Snailspeed's competitor Diddled Bits, Ltd. Snailspeed thinks that there is great demand for a faster bit rotator and would like you to develop a faster product to compete with Diddled Bits. You suspect that you can exploit word-level parallelism to greatly speed up their bit rotation program. The author of Snailspeed's program, Ben Bitdiddle, is an ex-Snailspeed employee. For this reason Snailspeed has a prototype of his bit rotation program. Currently, the prototype runs *very* slowly. It's your job to analyze its performance, and make it faster.

Many programs operate under tight constraints, and getting respectable performance under these circumstances can be especially challenging. Part of Snailspeed Ltd.'s product portfolio targets mobile and embedded devices such as cell phones. One such application requires that various operations on bit strings be performed within a large data buffer.

Since you can't change Snailspeed's culture overnight, please don't perform multithreaded parallelization. We'll learn more about these techniques later in the course. Code in standard C — for example, no inline assembly directives to the compiler. Furthermore, please don't use compiler intrinsics or libraries that use assembly or intrinsics to achieve the same effect. We want to see you use standard C. The x86 machines on which you will be running are little endian — bytes of words are stored in memory with least significant bytes first. You need not make your code portable to big-endian machines. You can post on Piazza if you have a question about the rules.

Forming your team

You should begin by finding a teammate and recording their name in the following Google form by midnight on **Thursday, September 13th**:

`https://goo.gl/forms/UCrivxsudQVZWBDo2`

Feel free to use Piazza to look for a teammate. If you wish the course staff to randomly assign you a partner, fill out the form, but enter 0 when asked for your partner's Kerberos ID.

Team contract

You and your teammate must agree to a team contract. A team contract is an agreement among teammates about how your team will operate — a set of conventions that you plan to abide by. The questions below will help you consider what might go into your team contract. You should also think back to good or bad aspects of team-project experiences you've already had.

Below are some questions to consider for your team contract broken into three categories: meeting and communication norms, work norms, and decision making. You needn't address all the questions, which are simply suggestions. Focus on the issues that your team considers most important. A skimpy team contract is a bad idea. The TA's will review all contracts. If you have little in your team contract and it ends up that your team has problems working together later in the group project, your TA will not be as sympathetic to your plight. All team members should write their names at the end of the contract, to indicate that they agree with it.

Meeting and communication norms

- How often will the team plan to meet outside of class? How long do you anticipate meetings will be? What will you do if things change?
- Where and when will outside-class meetings be held? What will you do if someone fails to show for a meeting?

- How will you communicate outside of meetings? (Email list? Real-time messaging platform?)
- If someone in the group decides to drop the class, what obligations does that person have to the team?

Work norms

- How much time per week do you anticipate it will take to make the project successful?
- How will work be divided among team members? On which parts of the project will you do pair programming?
- What will happen if someone does not follow through on a commitment, e.g., not doing their work? What if someone gets sick?
- How will the work be reviewed? How will you manage your code branches?
- How will you deal with different work habits of individual team members (e.g., some people like to get work done early, while others like to work under the pressure of a deadline)?

Decision making

- Do you need unanimous consent to make a decision? What process for decision-making will you use if you can't agree?
- How will you prioritize the work to be done? How will you deal with the common situation in which different team members have different optimization ideas?
- What happens if everyone does not agree on the level of commitment, e.g., some team members want an A, but others are willing to settle for a B.
- Is it acceptable for some team members to do more or less work than others?

Submitting your team contract

Each member should individually submit the team contract as a PDF document via Learning Modules. In addition, once your team repository has been formed, you should commit a copy of your team contract to the top level of the repo under the name `team-contract.pdf`. Feel free to update the team contract in the repo as needed during the project, as long as all parties agree.

Getting the code

Within 48 hours of the form submission deadline, you will receive an email notifying you of your `team-name`, which you can use to get the project code via:

```
$ git clone \
/afs/athena.mit.edu/course/6/6.172/student-repos/fa18/projects/project1/team-name.git project1
```

You can browse the code by cloning a read-only repository:

```
$ git clone https://github.mit.edu/rverkuil/project1.git
```

If you would like to get started before your team repo is set up, we recommend cloning the code from Github and then changing the remote once you receive the team repo, which you can do with the following commands:

```
$ git remote remove origin
$ git remote add origin \
/afs/athena.mit.edu/course/6/6.172/student-repos/fa18/projects/project1/team-name.git
```

(You will not be able to push to the team repo until it is set up, of course.)

We strongly recommend that groups practice pair programming, where two partners work together with one person at the keyboard and the other serving as watchful eyes. After an agreed-upon time, the partners switch. This style of programming leads to bugs being caught earlier, and both programmers retain familiarity with the code.

As soon as your repo is setup, add and commit a copy of your team contract to your project repo for future reference by you, the course staff, and your MITPOSSE mentors.

Code structure

The code resides in the `everybit` directory. Take a look at `bitarray.h` and `bitarray.c`. This code implements the functions needed to allocate, access, and process large strings of bits while using a minimum of memory. In the implementation, bits are packed 8 per byte in memory, but can be accessed individually through the public `bitarray_get()` and `bitarray_set()` functions.

Your job is to improve the `bitarray_rotate()` function programmed by Ben Bitdiddle. For the following tasks, do not call `bitarray_new()` from within your implementations. You can allocate small buffers on the stack or in the BSS section (e.g., global arrays).

Your implementation of `bitarray_rotate()` will be considered correct if the contents of the bit array, as accessed through `bitarray_get_bit_sz()` and `bitarray_get()`, is the same as after running the original, slow, implementation.

The timing and testing system resides in `main.c` and `tests.c`, which call your routines. Do not make modifications to these two files, as they will be replaced with fresh copies when the staff runs your code. By the same token, do not remove or change the signatures of any of your functions that `main.c` or `tests.c` calls. In short, the provided files `main.c` and `tests.c` should always compile against your code and execute correctly!

Building the code

You can build the code by typing `make`. To build with debugging symbols, useful when debugging with gdb, type `make DEBUG=1`. The `everybit` binary accepts arguments which allows you to

to run a benchmark composed of a long running bit rotation operation, run all tests in a given test file , and run a specific test from a test file. For usage instructions you can run

```
$ ./everybit
    usage: ./everybit
    -s Run a sample short (0.01s) rotation operation
    -m Run a sample medium (0.1s) rotation operation
    -l Run a sample long (1s) rotation operation
    -t tests/default        Run all tests in the test file tests/default
    -n 1 -t tests/default   Run test 1 in the test file tests/default
```

Benchmarks

When evaluating your implementation's performance, you should use the `awsrun` command, as this is what the staff will use to grade you. You can run a sample benchmark using the command

```
$ awsrun ./everybit -l
```

The `./everybit -s`, `-m`, and `-l` options run benchmarks of varying lengths. Each benchmark geometrically increases the number of rotated bits until your algorithm takes longer than a particular threshold (.01 seconds for `-s`, .1 second for `-m`, and 1 second for `-l`). We will grade your performance off of the tier and final tier time for a benchmark similar to `-l`. (This means that incremental improvements, even if they do not move your team to the next tier, will improve your grade.)

The naive long-running version will not complete many tiers. But simple optimizations should allow your rotation code to complete much higher tiers. You should do your final benchmark using `awsrun`.

Testing

For `everybit`, the staff has provided a testing framework which allows you to write tests in a simple textual format. Each set of tests is a file in the `tests` directory. We have already provided a few simple tests in the file `tests/default`. You can add additional tests to this file, as well as into the file `tests/mytests`, which will be used to determine your test-coverage grade (please see below). You can also add additional test files in the `tests` directory in the same format. **Remember to `git add` any new test files you create!** You can run all test files in the `tests` directory with the command

```
$ make test
```

You may also run the command

```
$ make testquiet
```

which will run all tests, but output information only for tests that fail.

Testing is a fundamental component of good software engineering. You will find that having a regression suite for your projects speeds your ability to make performance optimizations, because

it is easy to try something out and localize the bug, rather than spending hours trying to figure out where it is, or worse, never realize that you have a bug in your code.

The provided tests in `tests/default` do not provide adequate coverage (especially for faster and more complex implementations), and we will be looking for you to add more test cases. Your goal with testing is to find bugs not only in your own code, but in the code written by others in the class. In particular, your regression suite will be run against other teams' projects. If another team's buggy program passes all the tests in your regression suite, you will lose points. The harder it is to find a bug, the more points the bug is worth, so your goal should be to find as many corner cases as possible.

Write test cases for all the edge cases in `everybit`. Include some general tests, but think creatively about how to keep your test suite small. You can create as many test files as you wish in the `test` directory, and the `make test` command will run then all.

As part of your code submission, edit the file `tests/mytests`, and place your 20 best tests in the file. The staff will run your 20 tests against everyone's code for the "test coverage" part of your grade. The more bugs you uncover in others' code, the higher your grade. (Make sure your own code passes your own tests!)

Research on software engineering shows that over 90% of bugs in code previously occurred in an earlier version of the same software. Whenever you find a bug in your own code that passes all the tests in your current regression suite, it is a good idea to add a test case for that bug so that you can immediately catch it if it shows up again. Thus, developing a regression suite that checks for previously encountered bugs can vastly accelerate your ability to develop good (and fast) software. Indeed, the course staff has observed in previous terms that the quality of students' regression suites has been correlated with the performance of their applications themselves. So, if you want a better grade on the performance of your application, it's a good idea to develop a good test suite.

When you encounter a bug, it's tempting to fix the bug and leave the testing for later. That's actually an inferior strategy. Most professional software developers follow the following simple but effective methodology when a bug is found:

1. Write a test case.
2. Verify that the existing code fails the test case.
3. Fix the bug.
4. Verify that the new code passes the test case (and all the other tests in your regression suite).

Indeed, many software teams require that whenever a developer fixes a bug, a test case for the bug be checked into the team's software repository along with the bug fix itself.

3 Rotating a bit string

The function `bitarray_rotate()` rotates a string of bits within a bit array by some amount to the left or right. See the documentation in `bitarray.h`. Ben Bitdiddle's prototype is slow, however, performing lots of one-bit rotations over and over again until the correct degree of rotation is achieved. If you try to run

```
$ ./everybit -l
```

you will see that a couple of rotations on even a small buffer can take a while long to run. As a note, this mode does not perform correctness/error checking, nor is it necessarily the exact set of rotations we will use for grading.

Your task is to come up with a more efficient implementation for `bitarray_rotate()`, given the rules stated earlier. Your write-up and documentation should explain clearly and succinctly how your method works.

The most obvious way to perform a circular left rotation is to consider the string to be rotated to be of the form ab, where a and b are bit strings. We wish to transform ab to ba. The simplest method is to copy a to an auxiliary array, move b to its final location, then copy a from the auxiliary array to its final location. This method is simple, but the need for a large auxiliary array can be problematic for cache performance when rotating long strings.

To minimize the number of bit movements, a cyclic approach can be implemented, where each bit moves ahead by the specified amount, modulo the length of the region to be rotated, eventually looping back to the first bit in the cycle. This strategy places all bits in the correct locations while using a constant amount of auxiliary space, but memory accesses are scattered, which can adversely impact caching.

Finally, there is a clever approach that moves every bit twice without using auxiliary memory. Again treating the string to be rotated as ab, observe the identity $(a^{\mathsf{R}}b^{\mathsf{R}})^{\mathsf{R}} = ba$, where R is the operation that reverses a string. The "reverse" operation can be accomplished using only constant storage. Thus, with 3 reversals of bit strings, the string can be rotated.

There may also be other competitive approaches. Think, code, and profile! The TAs are happy to discuss your ideas with you at office hours.

4 Evaluation

Grade Breakdown

We will grade your project submission based on the following point distribution:

	Beta	*Final*
Performance	20%	30%
Test coverage	12%	
Correctness	7%	10%
Addressing MITPOSSE comments		10%
Write-up	3%	5%
Team contract	3%	
Total	45%	55%

This point distribution serves as a *guideline* and not as an exact formula. The staff will also review your Git commit logs to assess the dynamics of your team in the final submission. *Please ensure that each team member authors a substantial fraction of the project commits.* We strongly recommend that you experiment with pair programming for this assignment, as you will find that it is difficult to divide the work in this project into independent components.

Performance

Your performance grade for your beta release is based primarily on how fast your *correct* program runs. That means you should first focus on obtaining a correct solution for the project before optimizing for performance, since the more test cases a submitted solution fails, the lower grade your solution will get.

For the final version, all submissions have the opportunity to receive full credit on the performance criteria. After the beta releases have been submitted, the staff will choose a baseline performance goal. Any correct final submission whose performance matches or exceeds the baseline will receive full credit on performance.

Test Coverage

While working on your project, you will add test cases by modifying the provided test file, `tests/default`, and by adding additional test files in the same format to the `tests` directory.

To grade your code for test coverage and correctness, we will aggregate the test files submitted by every group (appending a prefix to each test file to avoid naming conflicts), and then run all submitted solutions against the aggregate test suite. We will remove any test files that do not pass when run against the reference implementation we supply. For this reason, we encourage you to put your tests in multiple test files. In addition, we will penalize groups that submit incorrect tests which do not pass when run against the initial implementation — there is no excuse for failing to run your test suite against the program we supply. The more bugs your tests catch, the higher your grade.

Correctness

Correctness grading is more subjective than coverage to allow us to evaluate the severity of bugs. Generally, grades should fall into the following categories. If all tests pass, you will get full marks. If your implementation is essentially correct, but misses some corner cases, you can expect between 80% to 90% of the points. If your implementation fails more tests but the performance tests still run, you can expect 60% to 80%. If you cannot run the performance tests, then you will get a low correctness grade.

If you are aware of a correctness bug in your own implementation, you will be penalized less if you inform us of the deficiency in your submission write-up. Note what steps you have taken to debug it, even though your debugging might not have been 100% successful. If you appear completely unaware of a bug, you will receive no correctness points for that test case.

Addressing MITPOSSE Comments

The MITPOSSE will give you feedback in GitHub on your code quality. We expect that you will respond thoughtfully to their comments in your final submission. We will review the MITPOSSE comments, your final submission, and your write-up to ensure that you are addressing the MITPOSSE comments.

Although code quality is subjective, good programmers produce programs that are neatly formatted, contain descriptive variables and function names, are partitioned well into modular units, are well commented, and contain liberal use of assertions via the `assert.h` package. Follow the C style guide published by Google: `http://code.google.com/p/google-styleguide/`. For example, every significant loop and recursive function should have an invariant (whether self-explanatory or documented in a comment) that can be verified with an assertion. You will find that it is easy to write a program that is "bigger than your head," where you return to the code even just a few days — sometimes hours — later and find it hard to figure out what you yourself were doing without investing a significant amount in time. Comments and assertions can greatly improve your ability to work on your program over a long period of time.

It can be difficult to maintain a consistent style when multiple people are working on the same codebase. We have provided a Python script `clint.py`, which is designed to check a subset of Google's style guidelines for C code. To run this script on all source files in your current directory use the command

```
$ python clint.py *
```

The code the staff has provided contains no style errors. You should use this tool to clean up your source code for this project.

5 Submitting your work

When you complete your beta release, your team will need to submit your code, including your 20 test cases, with Git and submit a write-up of your project to Learning Modules the next day. Similarly, when you finish your final release, you will need to submit your code and a write-up. In addition, every Thursday you will individually submit a short progress report to Learning Modules that summarizes the work you have done over the prior week.

Submitting code

Submit your code with Git before the beta and final deadlines. Remember to explicitly add new files to your repository before committing and pushing your final changes:

```
$ git add new-files
$ git commit -a
$ git status
$ git push
```

If `git status` shows any modified files, then you probably haven't checked your code into your repository properly.

In keeping with good programming practice, you should check in incremental changes to your repository as you write and test your code. Remember to balance these commits among your team members. We expect you to submit your code promptly by the due date.

A quick note about `git commit -a`: this command will commit all modifications to tracked files to your local repository. If you only wish to commit changes to certain files, you can `git add` them

explicitly and `git commit` will commit only files that have been manually staged (use `git status` to check what's been staged). If this is confusing, please just stick to `git commit -a`.

Please do not forget to submit your 20 best tests in the file `tests/mytests`.

Submitting write-ups

Submit your write-ups to Learning Modules by the due date, which is one day after your code is due.

Progress reports

You must individually submit a personal progress report every Thursday via Learning Modules that briefly describes the work you performed during the past week. The reports should be short: typically, a paragraph in length, describing what project work you engaged in and about how much time you spent on the various activities. Additionally, if necessary, describe any issues that may have arisen, such as with teammates — or rather, especially with teammates.

Along with your paragraph description, include a summary of the daily number of lines of code you committed using the following script:

```
$ cd <your/projects/base/directory>
$ loc_summary.py
```

Running `loc_summary.py` produces individualized summaries on all of the code you committed locally over all branches as well as on any code your partner wrote, pushed, and you subsequently pulled locally. The default run option is to produce a summary between the time of running and a week before. In general, you should run the script at about the same time every Thursday to ensure that all your commits are counted and none are double-counted.

Please do not worry if there are days in which you make no commits. We understand that you are taking other classes, that you have other obligations, and that even work in 6.172 sometimes involves things other than coding, such as thinking. Once again, please balance your commits among the team members so that everyone can show a fair share of commits.

Write-up

We ask you to submit small write-ups for the beta and final releases with the goal of helping the staff (who understand the assignment and its general strategies but have never seen your code) to fairly grade your assignment. We suggest you include the following:

- A brief overview of your design, particularly what improvements you made over the starter code for your beta design. This is not designed to replace appropriate code documentation, though.
- The general state of completeness/expected performance of your implementation, as well as any bugs/gotchas that you are aware of.

- Any additional information that you feel would be helpful to the staff in evaluating your submission (e.g., if you spent a lot of time on approaches that didn't result in a speedup, etc).

For the final writeup, you can also include:

- An overview of changes you made to your beta release, and what motivated you to do it (surprised with performance ranking? New revelations after your MITPOSSE meeting? Ideas conceived before the beta deadline but ran out of time implementing it?).
- Some comments on meeting with your MITPOSSE mentor.

6 Good Luck, and Have Fun!

We hope you enjoy the first project of 6.172. Start early, but remember that you can *always* make your code faster, although there tend to be diminishing returns. Budget your time wisely, both within 6.172 and to keep a balance with your other classes and activities. Good luck, and happy coding!