Benchmarking for Beginners

Fall 2023

1 Intro

In this homework assignment, we are going to test how much time it takes for a program to run. Runtime is an important metric of software performance.

There are several tools we have to test a program's run time.

The remainder of this document gives you some guidelines about how to benchmark things. You will write up and submit a report in the style of an academic paper, including:

1. **Introduction** section that describes the problem.
2. **Background or Methods** section that describes your techniques (ie how you approached solving the problem).
3. **Evaluation** section where you present results. This is where most/all of your plots will go. Each plot in your evaluation section should have some textual description, including an explanation of what is being measured.
    - Don't forget axis labels.
    - Your plot should show some trend. If the data you graph is all the same, it's not interesting.

2 Simple Benchmarking

Note: the instructions for what to turn in are at the end of this section.

First, we are going to write a long-running program and use some basic tools to test how long it takes to run. Let's start out with a trivial `for` loop:

```
int main() {
  for(int k = 0; k < 1000000000; k++) {
    asm("nop"); // prevent compiler from optimizing loop
  }
  return 0;
}
```

In this program, I have included an `asm("nop")` instruction to prevent the compiler from optimizing out the loop. Without this instruction, the compiler will realize that the loop is not performing any important task and remove it completely from the program. By including the `nop` assembly instruction, we are tricking the compiler to keep the loop so we can test how long it takes to run. We can compile this program as follows:

```
user@system ~ $ gcc -O2 -o benchmark benchmark.c
```

The `-O2` command line switch turns on optimizations. The table below explains `gcc`'s optimization options.

Command Line Switch	**Meaning**
`-O0`	No optimization
`-O1`	Some performance optimization
`-O2`	More performance optimization
`-O3`	All performance optimization (even possibly unsafe)
`-Os`	Optimization for code size

To measure the runtime of our program, use the builtin `time` command:

```
user@system ~ $ time ./benchmark
./benchmark  0.33s user 0.01s system 72% cpu 0.466 total
```

On my laptop (2022 MacBook Pro, M1 CPU), this program takes about 0.33 seconds (330 milliseconds) with the loop running for 1 billion iterations. The table below shows some common timescales and their abbreviations.

Time	**Meaning**	Timescale
Millisecond (ms)	10^{-3} s	Packet round trip time within a datacenter
Microsecond (μs)	10^{-6} s	Context switch latency
Nanosecond (ns)	10^{-9} s	DRAM memory access time, or cycle time of a 1GHz clock
Picosecond (ps)	10^{-12} s	Delay through a logic gate

To Do

1. Time the loop program for several iteration counts (10^2, 10^3, 10^4, ..., 10^{10} iterations). For each loop iteration count, run the measurement several times (say 20 times or so). Compute the mean and standard deviation runtime for each iteration count. Make sure you run all your tests on the same machine under the same conditions (no other programs running, etc.). As part of your submission, tell us the specs of your machine (CPU type, speed, etc.).
2. Plot the runtime as a function of iteration count. Turn this plot in.
3. Explain the general trend in your plot. Do you notice any strange behavior in the runtime of the loop? What is the reason for the shape of the plot? Write a short paragraph explaining the shape of the plot.

3 Built-In Benchmarking

Next we're going to build some benchmarking code in to our program itself.

```
#include <stdio.h>
#include <time.h>

int main() {
  // Create timespec structs, which hold the start and end times of our computation
  struct timespec tstart={0,0}, tend={0,0};

  // Get time from the operating system. This is reported as the number of nanoseconds
  // since midnight Jan 1, 1970
  clock_gettime(CLOCK_MONOTONIC, &tstart);

  // Computation we're measuring
  for(int k = 0; k < 100000000; k++) {
    asm("nop");
  }

  // Get time again
  clock_gettime(CLOCK_MONOTONIC, &tend);

  // Print time difference
  printf("loop took about %.5f seconds\n",
          ((double)tend.tv_sec + 1.0e-9*tend.tv_nsec) -
          ((double)tstart.tv_sec + 1.0e-9*tstart.tv_nsec));

  return 0;
}
```

To Do Re-run the timing experiments, creating the same plot, using the built-in benchmarking code above. Is there a difference in the amount of time you measured using the built-in technique vs. by running your process in `time`? What explains the difference?

4 Benchmarking a Nontrivial Operation

Next we're going to benchmark a nontrivial operation like a system call (instead of a `for` loop with `nops`). There are many system calls you can make.

- `getpid()` retrieves the current process's PID.
- `open()` opens a file.
- `write()` writes data to a file.
- `fork()` creates a new process.
- and lots more...

Choose a few system calls and time them. Present your findings in a bar graph that shows the mean runtime and some error bars. Usually error bars show the *standard error*, which is computed as $\sigma/sqrtN$ where σ is the standard deviation of the sample data and N is the number of samples.

5 Benchmarking Interprocess Communication Performance

Linux uses sockets, pipes and other mechanisms for interprocess communication. You will be measuring the performance of sockets. Below I have written a simple program that creates a pair of sockets (using the `socketpair()` syscall) and sends a short message through that socket.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/socket.h>
#include <stdio.h>

#define PARENTSOCKET 0
#define CHILDSOCKET  1

void main() {
    int fd[2];
    pid_t pid;

    /* 1. call socketpair ... */
    socketpair(PF_LOCAL, SOCK_STREAM, 0, fd);

    /* 2. call fork ... */
    pid = fork();
    if (pid == 0) { /* 2.1 if fork returned zero, you are the child */
        close(fd[PARENTSOCKET]); /* Close the parent file descriptor */
        const char hello[] = "hello parent, I am child";
        write(fd[CHILDSOCKET], hello, sizeof(hello)); /* NB. this includes nul */
    } else { /* 2.2 ... you are the parent */
        char buf[1024];
        close(fd[CHILDSOCKET]); /* Close the child file descriptor */
        int n = read(fd[PARENTSOCKET], buf, sizeof(buf));
        printf("parent received '%.*s'\n", n, buf);
    }
}
```

Measure the following characteristics and make a plot for each:

- **Message latency:** Latency is the time for some activity to complete, from beginning to end. For message passing, it is the time from the start of a send to the completion of a receive. Since the clocks on two different hosts may not be sufficiently aligned, the easiest way to measure message latency is to measure the time it takes to complete a round-trip communication (and divide by two). NB: Beware of nagling on the Internet stream experiments. This mechanism can cause unexpected delays. You can disable nagling with the TCP_NODELAY socket option.

  Measure latency for a variety of message sizes: 4, 16, 64, 256, 1K, 4K, 16K, 64K, 256K, and 512K bytes.

- **Throughput:** Throughput is the amount data that is sent per unit time. In this case, a round trip measure is not necessary; you can sent a return message when the entire transfer amount has been sent. Send a large enough total quantity of data such that the single "ack" response contributes a small amount of time compared to the whole transfer. Measure throughput for a variety of message sizes, the same as above.

6 The Experimental Method

Computer Scientists are notably sloppy experimentalists. While we do a lot of experimental work, we typically do not follow good experimental practice. The experimental method is a well-established regimen, used in all areas of science. The use of the experimental method keeps us honest and gives form to the work that we do. The basic parts of an experiment are:

1. **Identify your variables:** Variables are things that you can observe and quantify. You need to identify which variables might be related and whether a variable is a cause (i.e., the message size of a send operation) or the effect (e.g., the time to complete the send). Even though this sounds obvious, you should consciously identify the variables in each experiment that you perform.

2. **Hypothesis:** The hypothesis is a guess (we hope, an educated guess) about the outcome of the experiment. The hypothesis needs to be worded in a way that can be tested in an experiment, so it should be stated in terms of the experimental variables.

3. **Experimental apparatus:** You need to obtain the necessary equipment for your experiment. In this case, it will be the needed computer and software.

4. **Performance of experiment and record the results:** This part is the one that we typically think of as the real work. Note that several important steps come before it.

5. **Summarize the results:** Summarizing means putting the data in a form that you can understand. You might put the data in tables, graphs, or use statistical techniques to understand the raw data. If you are using averages, make sure to read Jim Smith's paper in the October 1988 issue of CACM (there are many types of means, and you need to use the right one)! However, as a warning, you probably do not want to use averages; taking the minimum makes much more sense in this case.

6. **Draw conclusions:** Note that performing the experiment and summarizing the results are separate steps and both come before you draw conclusions. To present honest and understandable results, we must present the basic data first (so that the reader can draw their own conclusions) before we insert our bias.