CS 740, Spring 2018
Homework Assignment 2

Assigned: Thursday, February 02
Due: Thursday, February 16, 9:00AM

The purpose of this assignment is to understand the impact of architectural features on performance by developing techniques for determining the architectural features of a system.

Policy

You can work in alone or in groups of up to three. Turn in a single writeup per group, indicating the group members as indicated below.

Logistics

In this assignment you will largely be responsible for your own software—there is no starter code or makefiles You will be using the timers from Assignment 1, however; if it is awkward to use the function timer you wrote, it is acceptable to use raw timers and manually ensure the running time is long enough to minimize error. Additionally, note that to get wall-clock time, you should use `{init,get}_etime_real()`. The regular functions give virtual time, which sums across threads, even if they run in parallel. We will also be supplying a graphing script (`plot.py`) for problem 1.

When you are ready to hand in your solution, upload it to Autolab. Your submission should be a .tar file consisting of the following:

1. `writeup.pdf`, your writeup as a pdf
2. `Makefile`
3. `mountain.c` for Problem 1
4. `cores.c` for Problem 2
5. `linesize.c` for Problem 3
6. `smt.c` for Problem 4
7. `lock.c` for Problem 5
8. `mmt.c` for Problem 6

Please hand in your assignment using Autolab. You may submit as many times as you would like.

Your writeup should begin with a section which lists the name of the machine you ran your programs on and the output of `/proc/cpuinfo`.

Your Makele, when executed, should create all the executables for each problem, i.e., `mountain` for `problem 1`, cores for `problem 2`, etc.

Finally, note that all questions in this assignment will be weight roughly equally.

Problem 1: Understanding the Memory Hierarchy

Modern processors have multiple levels of cache to improve performance. The goal of this problem is to determine the number of levels of the cache and the basic paramters of each level of the cache, e.g., the caches total size and its linesize.

Modern processors have multiple levels of cache to improve performance. The goal of this problem is to determine the number of levels of the cache and the basic parameters of each level of the cache, e.g., the caches total size and its linesize.

The basic idea is to compare the time it takes to read n bytes of memory with stride of s where n varies from C_{min} to C_{max} for reasonable values of C_{min} and C_{max} . As the stride increases from 1 to s_{max} the time to read n bytes should increase until s is greater than the line size. For a given stride, as n increases the time to read each byte should increase whenever n gets bigger than a particular level in the hierarchy.

Using your timing routines developed in assignment 1, write a program, called `mountain.c` which can be used to determine the characteristics of the cache on your machine. Your program should take one parameter which is either `simple` or `better`. When it is called with simple core loop used to determine performance should be something on the order of:

```
void test(int elems, int stride)
{

    int i;
    double result = 0.0;
    volatile double sink;

    for (i = 0; i < elems; i += stride)
    {
      result += data[i];
    }
    sink = result; // So compiler does not optimize away the loop

}
```

The output of your program should be in a format that you can use matplotlib (via `plot.py`; see `python plot.py -h` for details) to create a 3D plot, where the X axis is the stride from 1 to s and the Y axis is the log of the size of the array being read from C_{min} to C_{max} and the Z axis is the MB/sec. E.g., Each line should be of the form

```
1 26 9166.2
2 26 8506.6
3 26 6283.4
...
1 25 9159.3
2 25 8529.8
3 25 6604.9
...
```

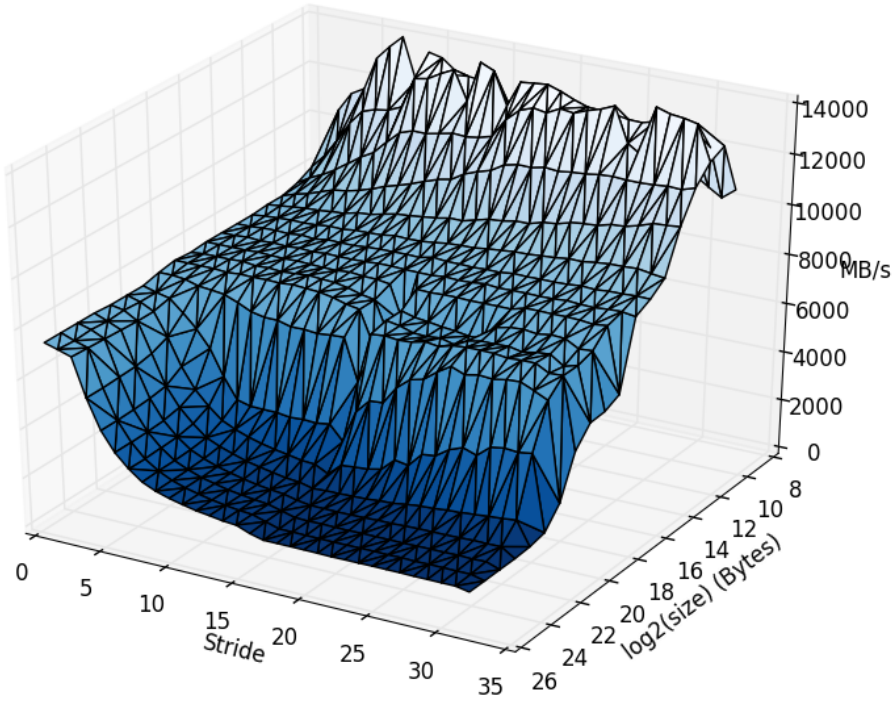

Figure 1: The Memory Mountain

Include in your writeup for problem 1a your conclusions about cache size, line size, and the graph your program generates.

You will probably notice that there is no perceptible drop-off in performance even as n exceeds the size of the largest cache on the chip. Explain why this happens in your writeup for problem 1b.

For the final part of this problem, modify `mountain.c` so that when it is give the parameter `better` it uses a different method for measuring the MB/sec for a given n and s. Include in your writeup a basic explanation of how you achieved this, the cache parameters, and the graph your obtained with this better method.

Problem 2: How many cores?

For this problem you will use the `pthreads` package to write a simple program which can be used to determine how many cores there are on a system. The output of `cores.c` should be a table which shows the performance of our microbench when run with 1 to n threads for some n which exceeds the number of cores on your system. Explain the results in your writeup for problem 2 and using those results determine how many cores the system has.

Problem 3: Determining the Shared Cache Linesize

For this problem you will determine the size of the cache linesize that is shared between cores. One way to do this is to determine when false sharing takes place. False sharing happens when two threads access and update distinct memory locations, but because of the cache coherence

protocol find that they are competing with each other because both memory locations are in the same cache line.

Using `pthreads` write a simple program which determines the linesize of the shared cache. The output of `linesize.c` should be a table which lists the linesize and the operations/sec for linesizes from 4 bytes to 1024 bytes. In your writeup indicate the linesize for the system you were running on.

Problem 4: SMT or Core?

In your solution to problem 2 (I.e., `cores.c`) you might have seen that you misrepresented the number of cores due to the processor also supporting multiple threads per core, e.g., by implementing some form of simultaneous multithreading. In this problem you should implement a microkernel that can determine how many SMT threads there are per core. `smt.c` should output a table which determines the operations/sec obtained from 1 to n threads for some reasonable n. Your writeup for this problem should describe the method you use to distinguish between SMT threads and number of cores. Then, using the output from `smt.c` show how many cores and threads per core there are on the system.

Problem 5: A Better Lock

Semaphores and other synchronization methods are notoriously slow in pthreads. Using `asm` statements implement a more efficient mechanism to atomically read, modify, and write a memory location a routine called `atomicIncr(int* location)`. Compare the performance of a multithreaded program which spawns n threads ($1 \leq n \leq 16$) which all increment a single variable $2^{20}/n$ times using either pthreads semaphores to surround the critical region of the update versus your `atomicIncr` routine. Include the output of `lock.c` in your writeup along with a brief conjecture as to why your routine is more (or less) efficient than the pthreads version.

Problem 6: Optimized Parallel Matrix Multiply

In this final problem you are tasked with using the information you obtained in the previous sections to write an efficient parallel blocked matrix multiply for square matrices. Your matrix multiply program should begin with a set of `#define` statements which can be used to parameterize the core matrix multiply routine:

```
#define THREADS t
#define BLOCK b
```

Where t are the number of threads the matrix multiple routine will use and b is the block size each thread will use. Determine the optimal t and b for the system you are running on and report the performance of your matrix multiply in MB/sec for matrix sizes of $1b \times 1b$ to $16b \times 16b$. (You may assume that the matrix size is an even multiple of b for this problem.)
