Exercises Week 5

Exercise 1 is by Eric B. Jul.

Exercise 1: `Prefetch.java` & CampusWire

One of the lectures was about caching, pre-fetching, read ahead and cache lines. A screenshot of latency.exe was shown: Lecture - Week 4 - slide 15.

In this exercise you are to make your own version of the latency.exe program. It should measure the time it takes to read bytes from memory in a few different ways. *Remember to do the printing after the measuring, as the `System.out.print` family of methods induce a lot of latency themselves.*

a) Program

Your program should measure these read times:

1. **Sequential read**. Read the elements in the array sequentially.

2. **Stepwise read**. Read the elements in the array by jumping a static number of steps each time. You should jump 1, 2, 4, 8, 16, 32, 64 and 128 bytes / kilobytes / megabytes.

3. **Random read**. Randomly jump around in the array.

To create an array of random bytes you can do:

```
import java.util.Random;
.
.
.
byte[] a = new byte[n];     // declaring and initializing a byte array of size n
new Random().nextBytes(a);  // filling the array 'a' with random bytes
```

> Make sure that each of these methods access the array the same amount of times before you compare them.

b) CampusWire

Once you have your results, post them, along with your CPU model, its clockspeed and its different level of cache, to the "Prefetch" note in CampusWire, and compare and discuss with your fellow students! :D

Exercise 2: `FalseSharing.java`

Write a parallel program that finds the sum of all odd numbers in a matrix. Divide the matrix by rows and let each thread work on their own set of rows. Let the threads store their local sums in a global array called `localSums`. Thread with id 0 should store its local sum at index 0, thread with id 1 at index 1 and so on. Write two versions of this program where you update this array in two different ways:

1. In the first version, update the element `localSums[threadId]` each time a thread finds an odd number.
2. In the second version, create a local variable `localSum` in the threads' `run()` method. Update this variable each time an odd number is found. Only after the thread has looped through the rows it has been assigned, update the element `localSums[threadId]` by setting it equal to `localSum`. This means that `localSums[threadId]` should only be updated once by every thread.

At the end, in both versions, the main thread should find the global sum by looping through and summing up the elements in the `localSums` array.

Test your program with `n = 10000, 15000 and 20000`. What do you notice? Remember to run both versions 7 times, find the medians and calculate the speedup of the median values.

Hint: To check if a number `num` is an odd number you can do:

```
if (num % 2 != 0) { it is an odd number }
```