Multicore Processors - Spring 2019
Homework 2

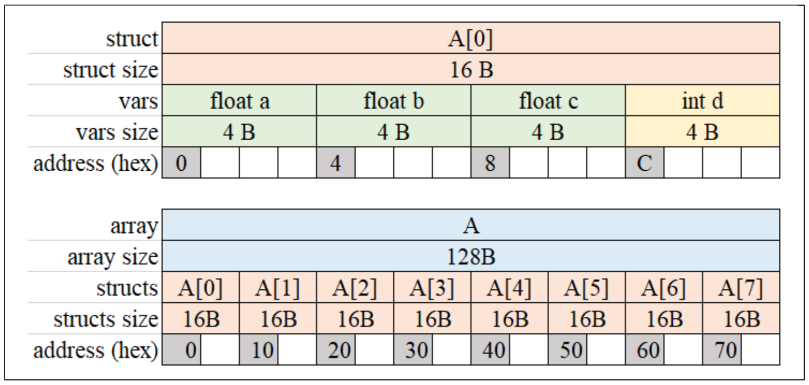
Daniel Rivera Ruiz
Department of Computer Science
New York University
drr342@nyu.edu

1. Suppose that you have an array of structures. The declaration looks as follows:

```c
    struct info{
        float a;
        float b;
        float c;
        int d;
        }A[8];
    }
```

8 threads will be accessing the different 8 elements of the array simultaneously (i.e. thread 1 accesses A[0], thread 2 accesses A[1], and so on).

(a) Draw a simple figure that shows how this array structure is stored in memory. Do not worry about exact addresses or element sizes.



struct	A[0]															
struct size	16 B															
vars	float a				float b				float c				int d			
vars size	4 B				4 B				4 B				4 B			
address (hex)	0				4				8				C			

array	A															
array size	128B															
structs	A[0]		A[1]		A[2]		A[3]		A[4]		A[5]		A[6]		A[7]	
structs size	16B		16B		16B		16B		16B		16B		16B		16B	
address (hex)	0		10		20		30		40		50		60		70	

Figure 1: Example of memory storage for the array of structures A.

(b) Based on your figure, will there be a lot of cache misses or not when the 8 threads access the array (assume we have one shared cache)? Justify.
Assuming a standard size cache line of 64 bytes, the whole array can be stored in two lines of cache, thus the number of misses will be very low. Originally there will be two misses (one per line), but once the lines are loaded all the other operations can be fulfilled without problem.

(c) If you say that the number of cache misses will be small, justify. If you say that we will have a lot of cache misses, how do we deal with that (from a programmer perspective, so don't mention a hardware technique)?
The reason why there are few cache misses is that the system has one shared cache and all threads can read and write freely from/to it. If this weren't the case, a lot of misses would be possible due to cache coherence protocols (invalidated copies of the data modified in other caches) and false sharing (trying to read/write a valid portion of an invalidated line).

2. Having too many threads in an application may not be a good idea. State all the reasons you can think of.
   - Creating a thread requires a system call, which is very expensive in terms of system resources and execution time.
   - If the amount of work performed by each thread is not enough to overcome the overhead of creating the threads, the overall execution time will increase.
   - Having more threads means the programmer has to be more careful about race conditions, data coherency, deadlocks, data transfer (in multiprocessor systems) etc.
   - The more threads, the harder it gets to debug the code for the application.
   - Having more threads doesn't necessarily mean there will be more parallelism: there is always an upper limit inherent to the application (Amdahl's law) and to the hardware (number of logical cores).

3. Assume we have p hardware threads (i.e. p cores with one-way hyperthreading or n cores with v-way hyperthreading where $v \cdot n = p$). For each of the following problems, specify how you are going to divide the problem among threads. Do not write code.

   (a) Find all prime numbers between 1 and n.
   Let us consider the following (very naive) algorithm to find all the prime numbers:
   1) Populate an array with the numbers from 2 to n.
   2) Mark all the elements of the array that are divisible by 2.
   3) Repeat 2) for all numbers up to $\lfloor (n+1)/2 \rfloor$.
   4) The unmarked numbers are prime.

   Based on this algorithm, we need to check divisibility of all the elements in the array by $m = \lfloor (n+1)/2 \rfloor - 1$ numbers. Given the p threads, we can assign $\lfloor m/p \rfloor$ numbers to each thread (some threads might have one additional element to complete the m numbers). Each thread will perform the divisibility test for its assigned numbers and update the marked/unmarked status of the array elements accordingly.

   (b) Find whether a number x is a prime number.
   Let us consider the following algorithm to check for primality:
   1) Get the remainder of x divided by 2: $r = x \bmod(2)$. If $r = 0$ return false (x not prime) and terminate.
   2) Repeat 1) for all numbers up to $n = \lfloor \sqrt{x} \rfloor$.
   3) If $x \neq 0$ for all values $\{2, 3, \cdots, n\}$ return true (x is prime).

   To implement this sequential algorithm in a parallel manner with p threads, the easiest approach is to assign to each thread $\lfloor (n-1)/p \rfloor$ divisions (some threads might have one additional division to complete $n-1$). After performing all its assigned divisions, a thread will have either a true (prime) or false (not prime) value. At this point we need to perform a reduce operation with the boolean operator `and`. The reduction should be performed in cascade, i.e. at the first level of the cascade there will be $p/2$ and operations, then $p/4$, and so on until reaching the last level of the cascade with only 1 operation which returns the final result.

4. A sequential application with a 20% part that must be executed sequentially, is required to be accelerated three-fold. How many CPUs are required for this task? How about five-fold speedup?
   By Amdahl's law we have:
$$S = \frac{1}{(1-p) + \frac{p}{s}}$$
   where S is the overall speedup of the application, p is the proportion of execution time that the part benefiting from parallelization occupies originally, and s is the speedup of the part that benefits from parallelization. Solving for s we get
$$s = \frac{pS}{1 - S(1-p)}$$
   For the three-fold speedup we have $S = 3$ and $p = 0.8$, which yields $s = 6$. This means that we need 6 CPUs to get a three-fold overall speedup.
   For the five-fold acceleration we have $S = 5$ and $p = 0.8$, which yields $s = +\infty$. This means that there is no way to get a five-fold overall speedup for the current application.

We can get to same conclusion if we look at the original expression for S and try to maximize it by increasing s:

$$\lim_{s\to+\infty} S = \lim_{s\to+\infty} \frac{1}{(1-p)+\frac{p}{s}} = \frac{1}{1-p}$$

If we plug in the value for $p = 0.8$ we get the maximum (theoretical) speedup value $S = 5$. This value is only theoretical because it would require an infinite number of CPUs.

5. Suppose we have a system with three level of caches: L1 is close to the processor, level 2 is below it, and level 3 is the last level before accessing the main memory. We know that two main characteristics of a cache performance are: cache access latency (How long does the cache take before responding with hit or miss?) and cache hit rate (how many of the cache accesses are hits?). As we go from L1 to L2 to L3, which of the two characteristics become more important? and why?
   Cache hit rate is more important in lower levels of cache memory. The following example helps understand why:
   Let's assume standard access values for L1, L2 and L3 of 1ns, 10ns and 50ns respectively. If L1 had a 100% hit rate, it would take 100ns to retrieve 100 pieces of data. However, if we reduce the hit rate by only 1%, it will mean that 1 out of the 100 pieces of data is either residing in L2 or L3, without considering the possibility that it could be in main memory or even in disk, which is even worse. Considering the standard access values, this 1% reduction in hit rate in L1 translates to 10% - 50% increase in execution time. If we look at the same scenario but in L2, 1 miss for every 100 pieces of data will only signify a 5% increase in execution time.
   We can arrive to the same conclusions if we think of the system in terms of latency. The overall increase in execution time arises from accessing data in L2 and L3, which have much higher access times than L1. If this values could be decreased (i.e. improving latency) the overall performance of the system would improve significantly.