Brian Rodriguez

ASSIGNMENT 1:

1.

a. BadLoc.c is making reads/writes to the array in top-down order. The cache will have to fetch more chunks of memory, after reading around 2 values from the array. Depending on the OS, the fetches will happen more frequently. GoodLoc.c reads the chunks of memory sequentially. The only time goodloc.c will have to fetch from memory, is when the final read happens at the end of the cache line.

b. BadLoc = The elapsed time is 3.670000e+06 seconds
GoodLoc = The elapsed time is 6.400000e+05 seconds

2.

a. The increase of padding hinders the performance at the start, then slowly starts to get better over time. Even with the number threads being used on the program, the program doesn't see much performance improvement.

b.

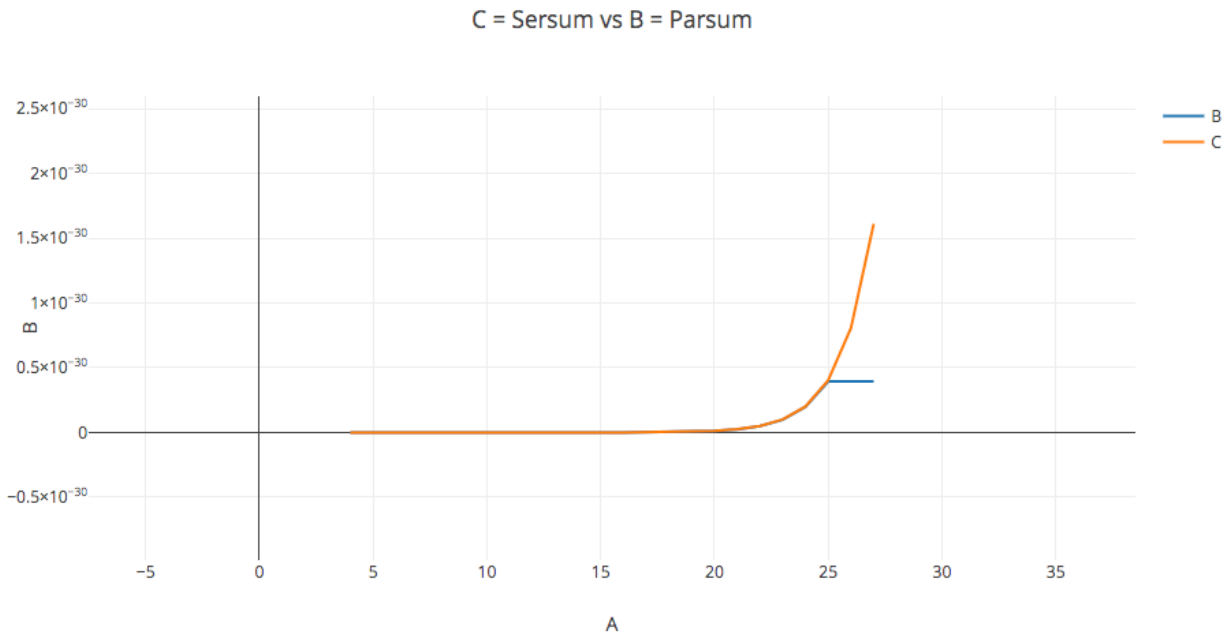
Padding	Thread #	Elapsed Time
2	2	7.299066e+00 seconds
8	4	8.764963e+00 seconds
32	6	2.871125e+00 seconds
64	2	1.993913e+00 seconds
128	4	2.762776e+00 seconds
256	8	2.868584e+00 seconds
512	2	1.995549e+00 seconds

Over time, the performance gets better, regardless of the number of threads. With 512 padding running on 2 threads, the time is still faster than the previous two running on multiple threads. It seems to me that having more padding and less threads, actually makes the program faster.

3.

a. Yes I do see that numbers increasingly diverge by a huge factor as the exponents increase.

b.



The behavior if different because the two programs are using two types of algorithms. The parsum.c is doing a partial summation of the input and adding them together, more of a divide and conquer strategy. The sersum.c is just adding the current spot in the array with the next spot and so on. This would definitely give two different types of answers as the exp range gets bigger.

5.
The sweet spot for my machine was about 2-4 threads spawned at any given time. Any more than that the work of creating threads costed more than the actual work being done.
1 thread working on the whole chunk size works in linear time, meaning it cannot perform any fast than that. 2 threads work in n/2 time, meaning that the chunk size of the threads spawned are halved. After a certain point of thread spawning, there are diminishing returns and the returns actually increase over time.

QUIZ 1:

Q1: Multicore CPUs are having been and will be the future of computing around the world. Tim Mattson and Ed Gronchowski both pointed out that simpler architectures for CPUs save a tremendous amount of power. Tim has pointed out that the battle between software and hardware in previous years ends in the future. Parallel computing is a must for a software developer as the hardware will slowly be slower on a single CPU. The fact of the matter is, that in order to get more out of the hardware and the program being built, you must parallelize the work. The book has also stated that, multicore processors have been able to toy with different

cache sizes and core counts to have a greater diversity in processor size in terms of transistors. Meaning that more power has been abled to be saved.

Q2: A process the execution of a program, programs can create threads which allow for the program to be parallelized. A thread has its own copy of the code but shares global global variables with the other threads created by the parent thread. A task, depending on the OS can either be a program execution, or an invocation of a program. All of these things are encompassed in a multicore CPU. The multicore CPU is a chunk of processors that can parallelization of the process/task being run. Hyperthreading can be done on a multicore CPU. For every physical core present, the OS can address two virtual cores and share the same workload between them. NUMA is a type of memory design used in multiprocessing, where memory access time depends on the memory location relative to the processor. Cache coherence is a different type of memory architecture. It is the uniformity of shared resources that ends up on multiple local caches.

Q3: A 12' pizza is about 1040 calories or 1.04 kcals. If we set aside 120W for the brain and body then we are left with about 4234.7 Watts. If we take that 1 W = 1 J/s and for 1 calorie = 4184 joules. If we take 1 calorie per second then that leaves us with 4.1868 Watts. Thus giving us a total of 4354.7 Watts in a pizza.

Q4: Herb Sutter does a fantastic job at displaying the importance of of avoiding data races. He goes on to talk about how the data races can mangle the code while the programmer is debugging and explains that when debugging optimized code, it is almost impossible to see what the other threads are doing. Herb mentions that If the programmer does not create race conditions, then the optimizer can run a program that is seemingly similar to the original program. The compiler does not know what variables are shared, so he mentions that these cases must be handled with mutex's, in order to hold the illusion that the compiler is running the same program you wrote. If you write a race condition, one thread can see into another thread with the same view as the debugger, meaning that If you write a race you'll have undefined behavior.

Q5: Herb Sutter talks about the importance of acquire and release operations. These operations are important not only for concurrency but also for compiler flags. The compiler will not optimize code inside of a critical section marked by locks and unlocks. The compiler also knows not to touch atomic types. Acquire and release operations cannot be modified by the compiler. Meaning that the acquire and release operations cannot be moved passed eachother when the code is being optimized. Its import to note that the acquire release operations should always have a matching pair to prevent deadlocks in the code.

Q6: Herb Sutter mentions at the start of his talk that the program you wrote is not the same program that you thought you wrote. He goes on to say that the reads and writes that you may have thought you coded in sequential order are done in a completely different order optimized by the compiler, processor and cache. An example that Herb gives is when we are in a for loop

and writing many times to a variable, the optimizer will write the result to a register and then write the register value to the desired result. This way the processor will have less memory traffic. Another example is the switching of a double for loop when writing to an array. So instead of accessing j first we access I, which allows for a nice linear access order optimization and takes advantage of spatial locality.

Q7: Chris Wood's tutorials are very helpful in a number of ways. Although most of the things that we explained I already knew, one thing I did not know was a snippet of code that he had written. In his include folder, part1.h has a very clever snippet of code called map which takes in a function and a vector of args followed by … It took me a while to fully understand what the code was actually doing. The map function maps the function given and the arguments to execute the function with. The result is that the return vals are stored in a result vector defined inline to the function. The … is called the variadic template, it allows for many arugments to be passed into the function using the template.

```
#include <stdarg.h>

double average(int count, ...)
{
    va_list ap; // the list of args used in the variadic
    double tot = 0;

    va_start(ap, count); //Requires the last fixed parameter (to get the address)

    for(int i = 0; i<count; i++)
    {
        tot += va_arg(ap, double); //Requires the type to cast to. Increments ap to the next argument.
    }

    va_end(ap);

    return tot/count;
}
```