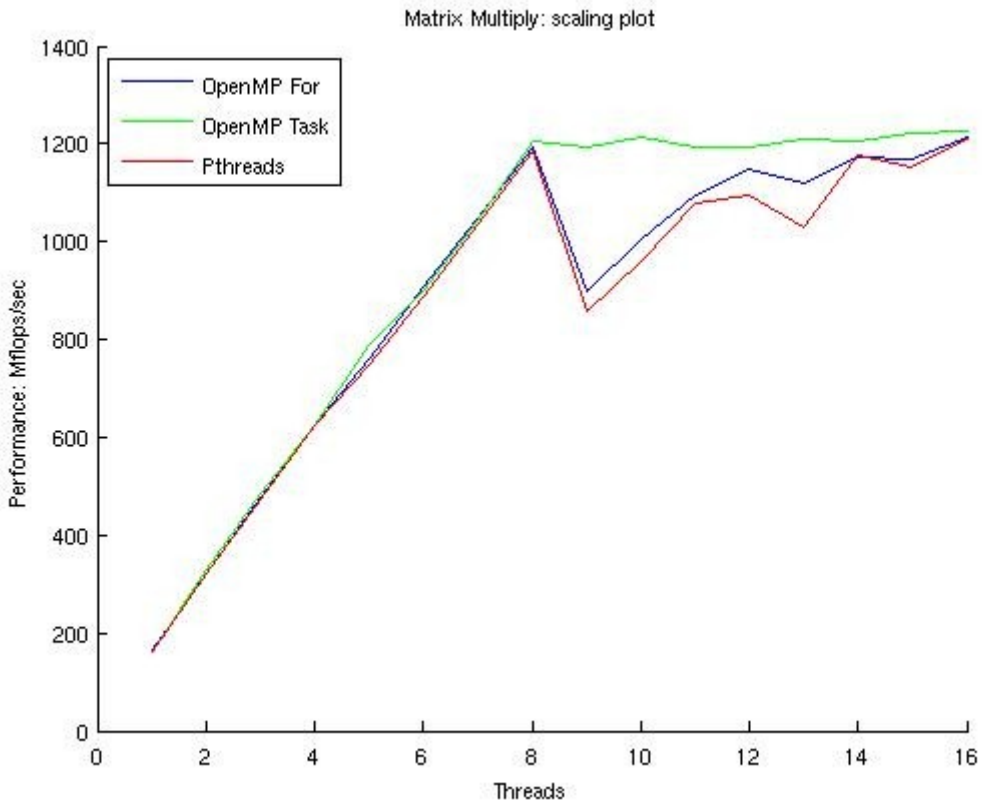
CS194-15: Engineering Parallel Software Assignment 2
Vanessa Matalon cs194-ax / 23126281

Matrix Multiply



Peak performance

The best parralel implementation was ~7.4 times faster than the scalar baseline. I achieved linear scaling for the first 8 threads (for all 3 implementations), but after that the performace did not improve much further. I believe that the reason the linear scaling ends after the 8th thread is because at that point the program is running on both microprocessors. This decreases performance because both chips need to share memory.

Measuring up

My matrix-multiply (1.2 GFlops/sec) is way less than peak performace (76.8 GFlops/sec). I am achieving less than peak performance because I need read the data that is being used and write it to memory. Additionally, I am not the only person running a program on this Hive machine. In order to improve performance I could use Intel's SSE Intrinsics in order to take advantage of SIMD and loop unroll.

Personal preferences

I like OpenMP better because it is easier to use. All the programmer needs to think about is what is parallel, how the blocks should be executed, and what data must be private to each thread. It leaves creating the threads and assigning their work to the compiler, things that must be made explicit by the programmer when using Pthreads. Assuming that I have exposed the parallelism in my code, it leaves a lot less to be done compared to using Pthreads.

How does OpenMP work?

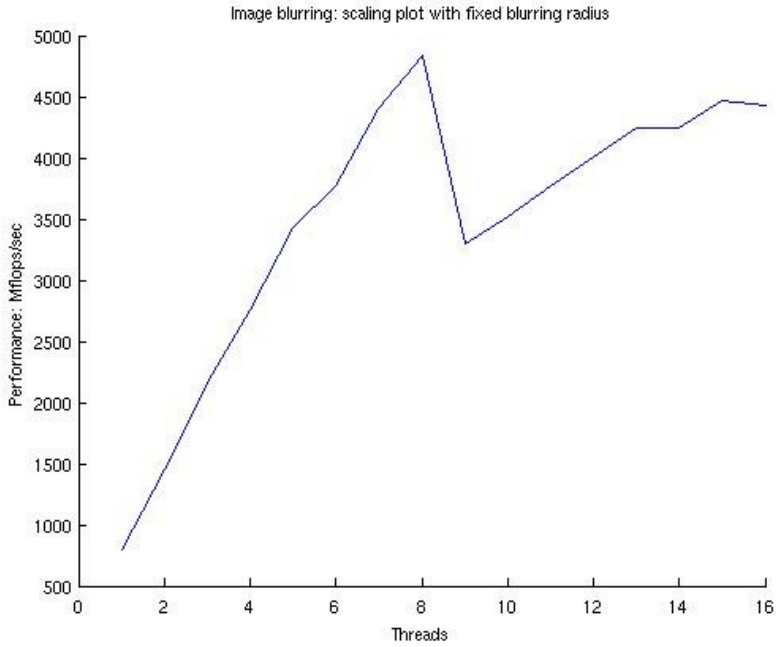
OpenMP provides the programmer with a way to implement threading at a high level. It requires the programmer to expose the paralellism in the program and direct the compiler how to generate the

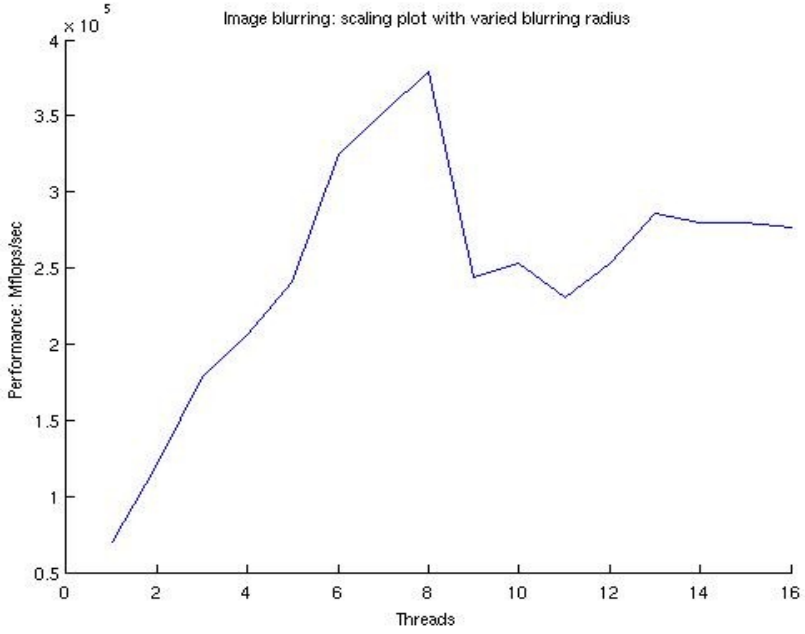
explicitly threaded code. Its implementation works by creating a team of threads, having the threads fork off from a master thread, and then join it again at the end of each parallel block. The threads can execute their tasks independently or we can achieve task parallelism by divinding a task among the threads.

How do you think OpenMP works?

The compiler parses the OpenMP directives and generates some code for each thread. It allocates memory for the shared data and places private data on the thread's stack. Then, it generates the code in the parallel block and instructs the threads to execute their task by giving them start/end information. At the end, it joins threads to master thread once they have all completed their task.

Image Blurring





When I use a varied blurring radius, the performance does not scale as well as when I use a fixed blurring radius, and I think this is because each thread is doing a different amount of work. This variablility cannot be controlled because the size of the radius is fully random. One way to counter

this could be to generate a fixed series of random radii which is shared among all threads to even out the amount of work that each has to do. The way to achieve this would be to make idx a function of x and not of both x and y.

I spent about 6 hours on this assignment.