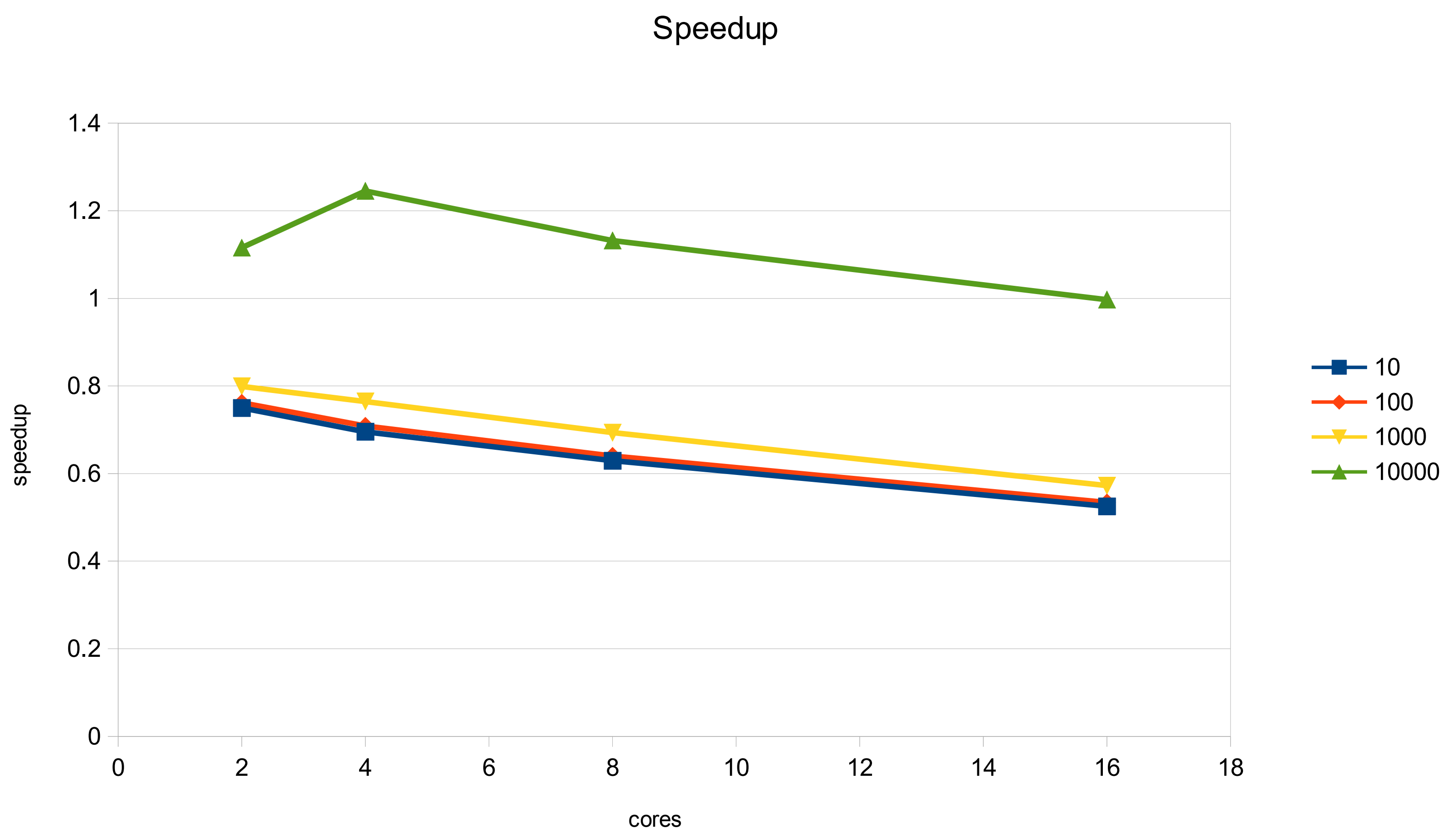
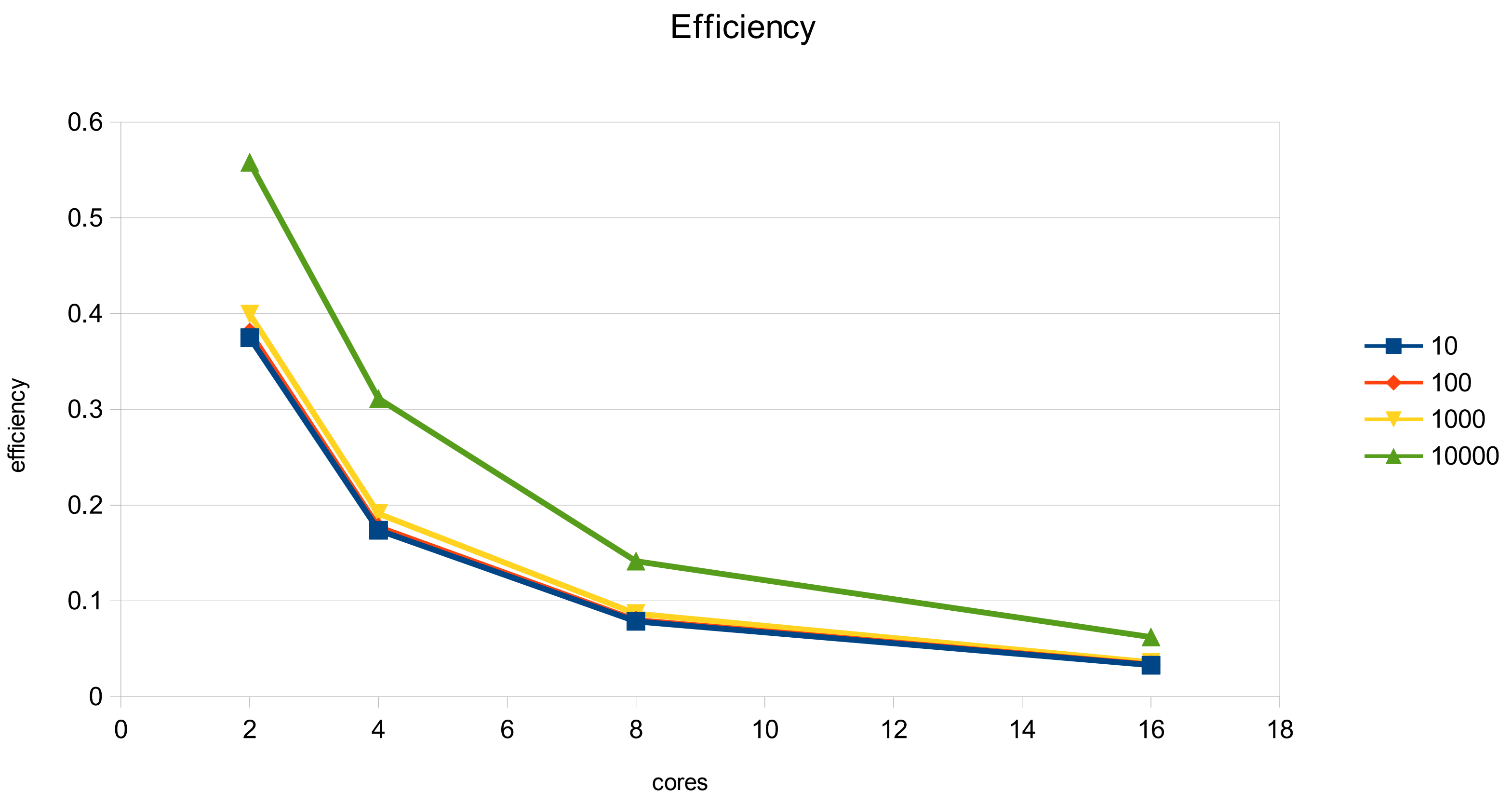
Christopher Pac
CSCI-GA.3033-009
Multicore Processors: Architecture & Programming
Lab Assignment #1



The number of cores is always the same as the number of threads created by the program. For this lab it did not makes sense to have more cores than created threads. Also, since we are testing for parallelism and not concurrency, we should assure that m2s has at least the potential to run the threads in a parallel fashion thus the number of created threads is the same as 'physical'/simulated cores.

The graph shows that there is no speedup over the sequential version except for three points. In fact the graph shows that there is a significant slowdown in the parallel version. The line with *n*=10000 shows speedup over the sequential version for 2,4, and 8 cores. I expected that for small values of *n* there would be no speedup and as the number of threads increased the speedup would decrease. Also, as expected, once *n* grew large enough and number of threads stayed small enough there was a speedup over the sequential program.

While *n* is small the overhead of creating threads, load balancing the problem, and terminating the threads is large and significant part of the total execution time of the program. Basically, the threads don't have enough work to justify paying the price for their management. Also, as the number of threads increases so does the time to manage them. For example, for *n*=10 and 8 threads, most threads do only one loop iteration and hence do not perform enough work to justify their creation. More importantly, as the number of threads increased and the work at each thread increased, the threads were being executed in parallel on different cores. This caused memory bus contention among the different running cores. That's why *n*=10000 runs fastest on only 4 cores.



Comparing the four curves, it's clear that efficiency increases as *n* gets larger. For example, efficiency is the highest for the curve *n*=10000 and then for the curve *n*=1000. This is due to the fact that for high values of *n* each thread has more work to justify its management cost. As expected for this problem, for all four curves efficiency drops as the number of cores/threads increases. This is because each additional thread does less and less work while the overall cost of managing the extra threads increases. The problem size is just not significant enough compared to the thread management overhead.

Additionally, the efficiency decreases as the number of running cores increases due to memory access becoming slower. More cores are trying to access the same shared memory over a single bus. Memory bus contention becomes an overriding overhead for the problem. When this coupled with the relatively small amount of work performed in the by each thread we get negative returns (inefficiency).

This memory contention was especially evident in an additional experiment that I have conducted. In this experiment, threads were not allowed to proceed with a *for loop* execution till all sixteen threads were created. The exact same program executed much faster on a single core than on a sixteen core setup. In this experiment the threads had much less chance to be executed sequential and thus more cores were active at the same time increasing memory contention.