Sieve of Eratosthenes

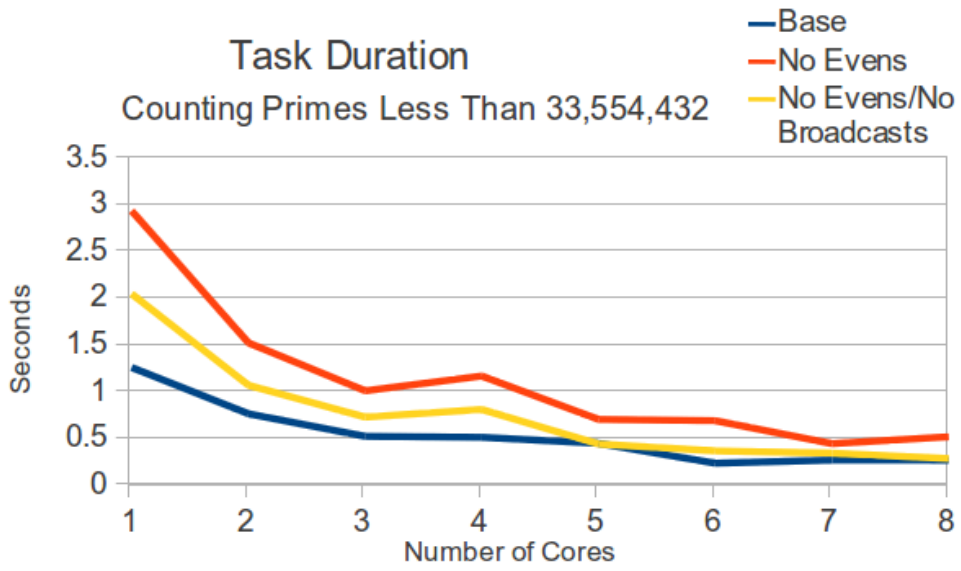
Zack Tillotson **February 23, 2012**

The given implementation of the Sieve of Eratosthenes prime number discovery algorithm was fairly naive. In the global sense it simply iteratively marks every multiple of each number from two to the largest possible prime value below a given arbitrary maximum. In the parallel implementation each process is assigned a block of numbers to improve the global run time. All numbers are initially assumed to be prime, each iteration chooses the smallest current prime number to use as the next number to mark with. This prime is broadcast to every process, which reduces replicated work. This base program was modified into two new versions.

The first task was to use less memory. In the original algorithm a given number is allocated a data element in memory. Starting with checking multiples of two, every possible multiple is marked. Even numbers are obviously not prime, but the algorithm both allocates them space and takes the time to mark each of them in the first iteration. This is a large waste of time and space. To optimize the algorithm we bypass this first round and do not store even numbers. Previously the data structure to mark numbers as prime or not was n elements large, where n was the number of elements between the smallest and largest numbers in the block. By not storing the even numbers the algorithm only needs n/2 data elements. On the other hand accessing the data elements is slightly slower as the mapping of the location of data elements in the data structure is more complex.

The second modified version of the program is an extension of the first, it also does not store even numbers. In addition this version assumes that waiting for the next prime to mark via broadcast from the root task could be a waste of time. Instead of each task waiting for the root process to tell it what prime to check next, each works independently and doesn't have to wait for the broadcasts. This method is more brute force because each task can't know which numbers to skip, but broadcasting does take a lot of time.

I tested each version using 1 through 8 cores to calculate the number of primes between 2 and 2^{25} (33,554,432). The results from these three versions of the program are given below.



There are several interesting features of these results. First is that each version of the program does improve on average as the number of cores increases. Going between 3 and 4 cores did actually slow the task on two versions of the program, I believe that is due the 4 core machines becoming over utilized. Second is that the task duration was best with the original naive algorithm. This does make some sense, as the modification which prevented even numbers from being explicitly stored was reducing memory usage rather than increasing speed. The third program's use of replicated logic rather than broadcasts was meant to speed the execution time, though, and it did.

Further work at improving algorithm could work on improving the range of the prime numbers to be discovered. The current implementation uses signed integers which limits the maximum prime values. Other work could speed by improving the replicated prime number search.