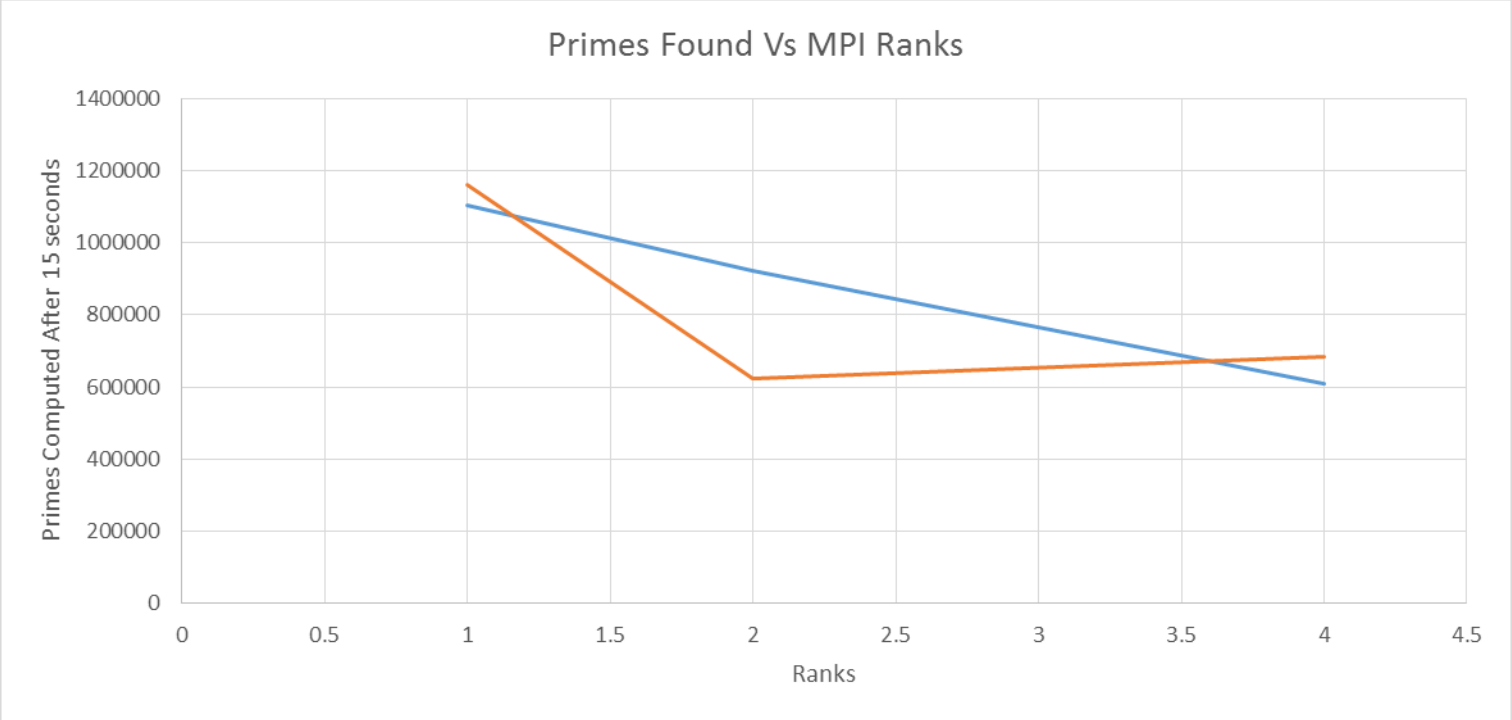
Matthew Mohr and Eric Lawson

SPEEDUP GRAPH AND ANALYSIS FOR HOMEWORK 4



RANKS	Primes Computed on Mac Sierra OS	Primes Computed on Ubuntu 14.04	Max N Reached (Mac Sierra OS)	Max N reached (Ubuntu)
1	1102323	1160994	17184131	18162475
2	922452	624960	14204472	9360638
4	607744	682546	9085174	10289758

ANALYSIS: Ideally, one would expect to see speedup as the number of ranks increase given that this problem requires very minimal dependency between ranks (After all, each rank is simply checking if a number is divisible by some lesser number, which hardly requires a parallel implementation involving a complex scheme for message passing between ranks). Our implementation does not yield such speedup on at least two different operating systems, indicating that our algorithm for prime computation and aggregation of results may not be optimal for improving performance. On the Mac operating system, incrasing the number of ranks leads to a steady decrease in the number of primes found, wheras the Ubuntu speedup shows a sharp drop in efficiency and then a moderate increase that is roughly equivalent to the 4 rank computation on the opposing system (an increase may not necessarily indicate speedup due to how close it is to the 4 rank result on the other machine, indicating that our parallel solution does no better than its predecessor to some degree). From these results, we can definitively conclude that any increased amount of ranks will fare worse than serial implementation, which is unexpected. After consulting with the course instructor, our group determined that our lack of performance improvement was due to the use of synchronous calls during each iteration of prime computation per rank. Our implementation works as follows:

-Have the n value for a particular rank start from its rank number + 2 (e.g. Rank 0 would start at 2, rank 1 would start at 3, Rank 2 would start at 4, and rank 3 would start at 5).

-For the n value, perform the prime computation loop (i.e divide n one at a time by the numbers ranging from 2 to sqrt(n) until you either determine that n is prime or composite).

-Perform 2 MPI_Gather, one after the other, calls to make 2 arrays whose size are equal to the total number of ranks. The first MPI_Gather call collects an array of the values of n for each rank in order (such as [2,3,4,5]), and the second array contains a series of 1s and 0s indicating whether or not the numbers in the previous array are prime or not. Traverse both arrays to compute the number of primes found this round and what number we are currently up to for n. If you hit the limit, which is a power of 10, while going through the arrays, print the aggregate results obtained up to that limit.

-Perform an AllReduce() call after computing the necessary information from gather to check if any process got the signal. If any process got the signal, exit the prime loop and return the final results.

-If we keep going, increase the value of n for the current rank by the total number of ranks (so an n value of 2 with 4 ranks means that the next number we check would be 6, then 10, then 14, etc.). Since all ranks start at a different number and the n value is a constant offset, we can guarantee that no two ranks will ever work on the same number (i.e. starting from n, n+1, n+2, and n+3 for four ranks, increasing each by the same constant will still result in a different number).

Based on our algorithm, it is clear to see that our performance is slowed substantially by the two MPI_Gather calls and the MPI_Allreduce call for loop termination. All of these calls are synchronous, meaning that all ranks must hit these calls before they can continue prime computation. In practice, this means that when any given rank finishes determining whether or not a number is prime, it must wait for all other primes to make their determinations, which ultimately results in each iteration taking as long as the slowest prime computation for that round. When doing these computations in serial, there is no holdup for computing the next number since the synchronous calls are made with only one rank, meaning that the serial can immediately continue when reaching these calls. With serial, the primes computations can seamlessly continue from one value of n to the next, while multiple ranks must wait up for each other. In this case, if we reach a round where we are computing a number that has a low factor indicating that it is composite and a number that is a very large prime at the same time, the composite number will essentially be hanging with a great deal of downtime while another rank determines that it is dealing with the large prime. This massive amount of downtime eliminates any potential speedup that would be gained using subsequent ranks. Additionally, since we are choosing the starting numbers of some ranks to be even, they will immediately be determined to be composite numbers, meaning that we will run into situations where at least half the ranks have finished very quickly while other ranks are dealing with large primes in their computation loop.

Although the barriers associated with the synchronous calls eliminate speedup, our algorithm does have the unique benefit of guaranteeing correctness when outputting results as well as guaranteeing that the ranks are working on n values that are relatively close together. Since we chose the offset to be the number of ranks, the difference between the minimum n value and the maximum value for every iteration is simply the number of ranks, which in the worst case will be 4 for this

assignment. This is a very acceptable boundary that ensures certain ranks do not compute a large number of primes very quickly while other ranks operate slowly, skewing the results as to how many primes below n were calculated. Since our synchronous calls keep the ranks in sync with one another with a very low offset, we can guarantee a higher degree of accuracy as to what value of n we reached and how many primes were exactly computed below that n value. Furthermore, because of how we staggered the numbers, we can guarantee that each rank is doing a unique computation per batch. Essentially, this means when we output a result once we receive a signal, we can guarantee that EVERY number below that value of n was checked and that our prime count is accurate and not the result of skewed rank progress. What our algorithm lacks in speed, it makes up for in ensuring accuracy and synchronization. Potential improvements to this code could include limiting the number of iterations where we make this synchronous call (such as running the gather and reduction calls every 5 iterations or so, as opposed to every iteration). When attempting to implement this, however, we found that attempting to synchronize the ranks and make sure that they all hit the barrier without the side effect of a rank exiting the main loop and finishing was fairly difficult and left us with segmentation faults and non-terminating processes. Ultimately, we decided to focus on accuracy and guaranteed termination at the expense of performance, which is a tradeoff that sometimes needs to be made when doing parallel programming.