MA/CSSE Homework 8
Due 5/19

Directions

- **Each problem must be self-contained in a file**, named properly, and **must compile using the command** `mpicc filename.c`
- Turn in each `.c` file to the dropbox. **Do not create a `.zip` file**

Problems

Level: Easy

1. The *Sieve of Eratosthenes* is a classic method for determining all the prime numbers between 2 and k. It, and its variants, are still reasonable methods for finding moderately sized prime numbers today.

   The sieving method works by striking out multiples of all known prime numbers. We start with a list, L that contains all integers 2 through k.

   $$L = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \ldots, k\}$$

   2 is the smallest number in the list, so we take it as prime. We now walk through the list in steps of 2, marking off all the multiples of 2.

   $$L = \left\{\boxed{2}, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15, \cancel{16}, 17, \ldots, k\right\}.$$

   The next smallest unmarked integer in the list is 3, which we now know is prime. We mark 3 as prime and strike out all the multiples of 3.

   $$L = \left\{\boxed{2}, \boxed{3}, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}, \cancel{16}, 17, \ldots, k\right\}.$$

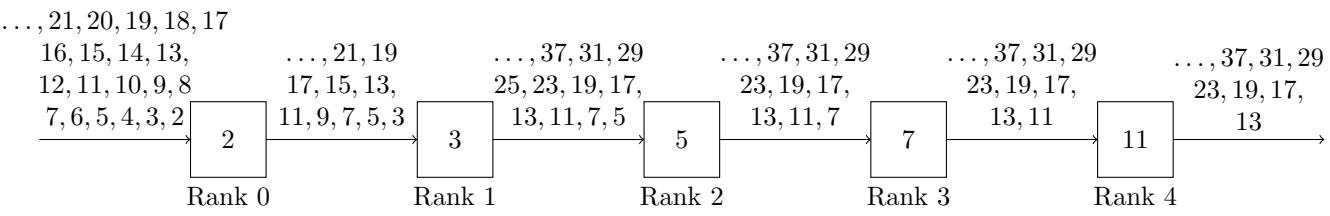
   The next smallest entry in the list is 5, which we mark as prime, and strike out all multiples of 5.

   $$L = \left\{\boxed{2}, \boxed{3}, \cancel{4}, \boxed{5}, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}, \cancel{16}, 17, \ldots, k\right\}.$$

   This process is repeated until we have marked out all the multiples of $j \leq \sqrt{k}$. At that point all unmarked entries in L may be safely marked as prime.

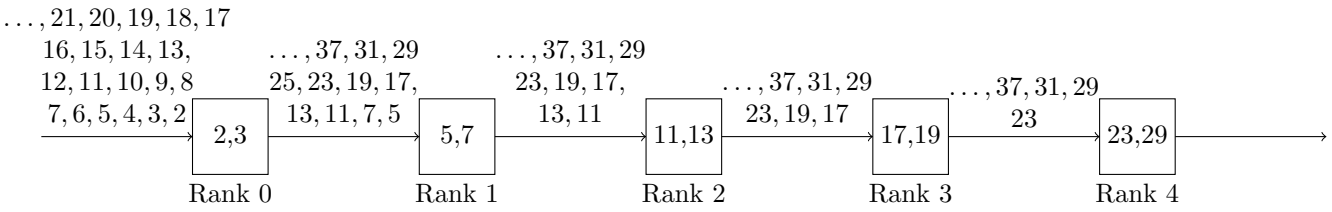
   We may think of the sieving method as a sequence of filters that are applied to L. First is the filter that discards all multiples of 2. The first element to survive that filter is 3, and so we run a filter that discards all multiples of 3 on the output of the first filter.

   A pipelined version of this algorithm might work as follows: Processor 0 acts as a list-generator and 2−filter. It checks every number x between 0 and k to see if x is a multiple of 2. Every number that is not a multiple of two is passed to the next processor. Processor 1 first receives 3 from rank 0, and so becomes a 3−filter. Processor 2 becomes a 5−filter, processor 3 a 7− filter, processor 4 a 11−filter, and so on. See the figure below:



We can see that all the "filters" are prime numbers, and that any number exiting rank 4 is not divisible by $2, 3, 5, 7$, or 11. Thus, any number making it past rank 4 that is less than 121 must be prime.

This is an effecitive method to determine the list of primes that are less than p^2, where p is the number of processors available. To allow us to generate bigger lists, we can have each processor be a filter for N/p primes, rather than just for 1 prime. See the figure.



Write a program in `eratosthenes.c` that calculates the first N primes using the Sieve of Eratosthenes. Each processor should act as a filter for at most $\lceil N/p \rceil + 1$ primes.

Note that rank 0 does not know ahead of time how many number to put into the pipeline, that must be detected as we go. Rank p is the only processor than can detect when we have generated N primes. When the N^{th} prime is found, have rank p send a special message to rank 0, telling it to stop processing. When rank 0 stop processing, have it send a message to rank 1, then from rank 1 to 2, and so on.

Rank 0 must periodically check to see if a message has arrived from rank p, telling rank 0 to stop processing. Use the command

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
```

on the root to check to see if there is a message from rank p. `MPI_Iprobe` checks to see if there is a message from `source` with the given `tag` on the communicator `comm`. If there is, then `flag` is set to true and the `status` variable is filled up with all the standard information about the message. For instance, you could check to see how large the message is, before receiving it. If there is no message available that matches the criteria, `flag` is set to false. In either case
ttMPI_Iprobe **will not block**, which makes it perfect for this use case.

Once the first N primes are found,use `MPI_Gatherv` to send them all back to the master process, and have the root print them if requested, and write a file if requested.

The standard implementation may be compiled with
`$ mpicc objs/eratosthenes_standard.<arch>.o -o eratosthenes_standard`

The file `eratosthenes_start.c` and `eratosthenes_helpers.h` give you some framework.

In addition to passing `test_eratosthenes.py`, hand in a file that contains the first 10^6 prime numbers named `lots_o_primes.txt`.

Level: Difficult

1. Create a program in `linear_solve.c` that uses the pipelined method of Chapter 11 (p355) with striped partitioning to solve linear systems of equations.

   In particular, suppose we want to solve a linear system of equations, $Ax = b$. Partition A and b so that each processor is responsible for roughly n/p contiguous rows of A and the same roughly n/p rows of b. Rank 0 begins the algorithm by passing row 0 of A and b to rank 1. Rank 1 forwards Row 0 to rank 2, which forward to rank 3, and so on. Once each processor, j, has forwarded row 0 to rank $j+1$, rank j performs elimination in column 0 on the rows for which it is responsible.

   Once rank 0 is done with elimination in column 0, it forwards row 1 to processor 1, and then begins elimination in column 1. The method continues until the matrix is in row-echelon form. Once the matrix is in row-echelon form, use back substitution, also pipelined, to find the solution vector x. Collect x on the root node, produce an output file or print x out if requested. See the attached pages for another description.

   The standard code may be compiled with
   `$ mpicc objs/linear_solve_standard.<arch>.o -o linear_solve_standard`

   You need to respect the options provided in the `options` structure, the file `linear_solve_start.c` provides a good framework.

   The root node should be the only matrix that reads in A and b. You should use `MPI_Scatterv` to distribute the proper partitions of A and b to the other nodes. There is no need to call a collective operation to gather x – back substitution will naturally lead to rank 0 having a full copy of x.

   The function `residual` calculates $\sum_i |(Ax)_i - b_i|$. You can use it to check to see if you have computed the correct solution. If `residual(A,x,b)` is very small (like 10^{-10}) then x is very very close to the actual solution to $Ax = b$.

   You may assume that A is non-singular, and that pivoting is not needed in order to get a good solution to $Ax = b$. You can generate random matrices for testing your code with `rand_inv_matrix n` and you can generate a random matrix (good for making b) with `rand_matrix m n`.

   Note that, with the exception of the initial `Scatterv`, each rank j only needs to communicate with rank $j-1$ during elimination, and with rank $j+1$ during back substitution. Your code must produce the correct results, respect all command line options, and must run within $2x$ time of the standard code.

   A note on sending rows: The matrix data structure is storing rows contiguously. So, if you want to send rows 3 and 4 of matrix A, and matrix A is 20 wide, you can use just on `MPI_Send` like this
   `MPI_Send(A.data[3],40,MPI_DOUBLE,...)`

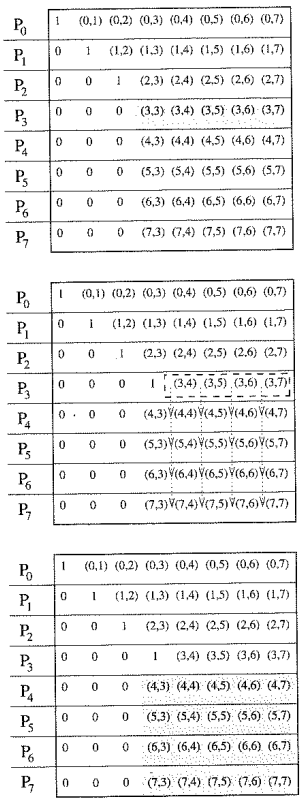


Figure 8.6 Gaussian elimination steps during the iteration corresponding to $k = 3$ for an 8×8 matrix partitioned rowwise among eight processes.

belong to the same process. So this step does not require any communication. In the second computation step of the algorithm (the elimination step of line 12), the modified (after division) elements of the kth row are used by all other rows of the active part of the matrix. As Figure 8.6(b) shows, this requires a one-to-all broadcast of the active part of the kth row to the processes storing rows $k + 1$ to $n - 1$. Finally, the computation $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$ takes place in the remaining active portion of the matrix, which is shown shaded in Figure 8.6(c).

The computation step corresponding to Figure 8.6(a) in the kth iteration requires $n - k - 1$ divisions at process P_k. Similarly, the computation step of Figure 8.6(c) involves $n - k - 1$ multiplications and subtractions in the kth iteration at all processes P_i, such that $k < i < n$. Assuming a single arithmetic operation takes unit time, the total time spent in computation in the kth iteration is $3(n - k - 1)$. Note that when P_k is performing the divisions, the remaining $p - 1$ processes are idle, and while processes $P_{k+1}, \ldots, P_{n-1}$ are performing the elimination step, processes $P_0, \ldots, P_k$ are idle. Thus, the total time spent during the computation steps shown in Figures 8.6(a) and (c) in this parallel implementation of Gaussian elimination is $3\sum_{k=0}^{n-1}(n - k - 1)$, which is equal to $3n(n - 1)/2$.

The communication step of Figure 8.6(b) takes time $(t_s + t_w(n-k-1)) \log n$ (Table 4.1). Hence, the total communication time over all iterations is $\sum_{k=0}^{n-1}(t_s + t_w(n - k - 1)) \log n$, which is equal to $t_s n \log n + t_w(n(n - 1)/2) \log n$. The overall parallel run time of this algorithm is

$$T_P = \frac{3}{2}n(n - 1) + t_s n \log n + \frac{1}{2}t_w n(n - 1) \log n. \tag{8.18}$$

Since the number of processes is n, the cost, or the process-time product, is $\Theta(n^3 \log n)$ due to the term associated with t_w in Equation 8.18. This cost is asymptotically higher than the sequential run time of this algorithm (Equation 8.17). Hence, this parallel implementation is not cost-optimal.

Pipelined Communication and Computation We now present a parallel implementation of Gaussian elimination that is cost-optimal on n processes.

In the parallel Gaussian elimination algorithm just presented, the n iterations of the outer loop of Algorithm 8.4 execute sequentially. At any given time, all processes work on the same iteration. The $(k + 1)$th iteration starts only after all the computation and communication for the kth iteration is complete. The performance of the algorithm can be improved substantially if the processes work asynchronously; that is, no process waits for the others to finish an iteration before starting the next one. We call this the ***asynchronous*** or ***pipelined*** version of Gaussian elimination. Figure 8.7 illustrates the pipelined Algorithm 8.4 for a 5×5 matrix partitioned along the rows onto a logical linear array of five processes.

During the kth iteration of Algorithm 8.4, process P_k broadcasts part of the kth row of the matrix to processes $P_{k+1}, \ldots, P_{n-1}$ (Figure 8.6(b)). Assuming that the processes form a logical linear array, and P_{k+1} is the first process to receive the kth row from process P_k. Then process P_{k+1} must forward this data to P_{k+2}. However, after forwarding the kth row

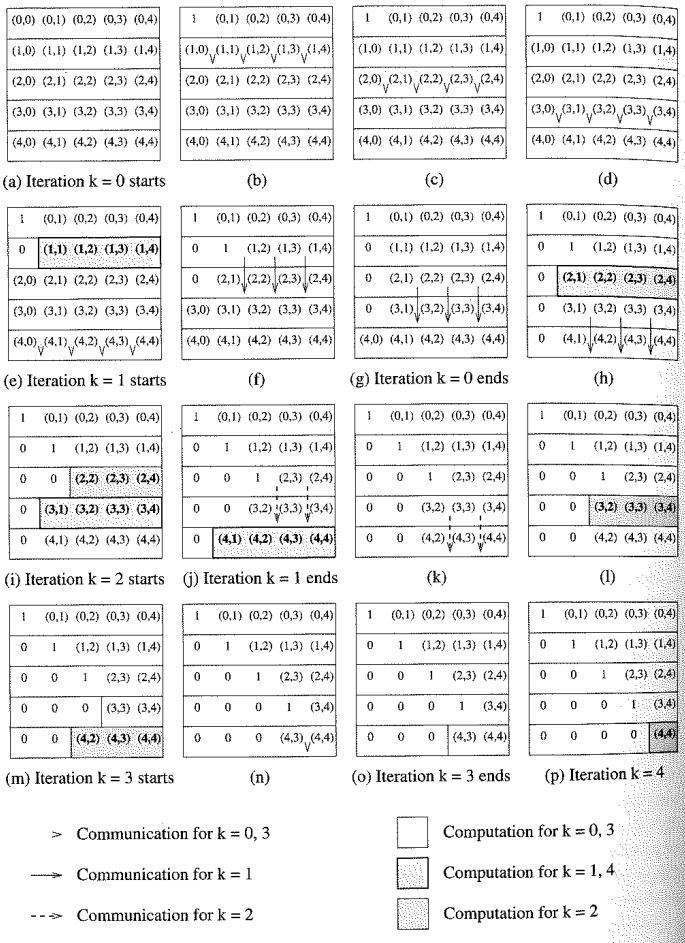


Figure 8.7 Pipelined Gaussian elimination on a 5 × 5 matrix partitioned with one row per process.

to P_{k+2}, process P_{k+1} need not wait to perform the elimination step (line 12) until all the processes up to P_{n-1} have received the kth row. Similarly, P_{k+2} can start its computation as soon as it has forwarded the kth row to P_{k+3}, and so on. Meanwhile, after completing the computation for the kth iteration, P_{k+1} can perform the division step (line 6), and start the broadcast of the $(k+1)$th row by sending it to P_{k+2}.

In pipelined Gaussian elimination, each process independently performs the following sequence of actions repeatedly until all n iterations are complete. For the sake of simplicity, we assume that steps (1) and (2) take the same amount of time (this assumption does not affect the analysis):

1. If a process has any data destined for other processes, it sends those data to the appropriate process.
2. If the process can perform some computation using the data it has, it does so.
3. Otherwise, the process waits to receive data to be used for one of the above actions.

Figure 8.7 shows the 16 steps in the pipelined parallel execution of Gaussian elimination for a 5 × 5 matrix partitioned along the rows among five processes. As Figure 8.7(a) shows, the first step is to perform the division on row 0 at process P_0. The modified row 0 is then sent to P_1 (Figure 8.7(b)), which forwards it to P_2 (Figure 8.7(c)). Now P_1 is free to perform the elimination step using row 0 (Figure 8.7(d)). In the next step (Figure 8.7(e)), P_2 performs the elimination step using row 0. In the same step, P_1, having finished its computation for iteration 0, starts the division step of iteration 1. At any given time, different stages of the same iteration can be active on different processes. For instance, in Figure 8.7(h), process P_2 performs the elimination step of iteration 1 while processes P_3 and P_4 are engaged in communication for the same iteration. Furthermore, more than one iteration may be active simultaneously on different processes. For instance, in Figure 8.7(i), process P_2 is performing the division step of iteration 2 while process P_3 is performing the elimination step of iteration 1.

We now show that, unlike the synchronous algorithm in which all processes work on the same iteration at a time, the pipelined or the asynchronous version of Gaussian elimination is cost-optimal. As Figure 8.7 shows, the initiation of consecutive iterations of the outer loop of Algorithm 8.4 is separated by a constant number of steps. A total of n such iterations are initiated. The last iteration modifies only the bottom-right corner element of the coefficient matrix; hence, it completes in a constant time after its initiation. Thus, the total number of steps in the entire pipelined procedure is $\Theta(n)$ (Problem 8.7). In any step, either $O(n)$ elements are communicated between directly-connected processes, or a division step is performed on $O(n)$ elements of a row, or an elimination step is performed on $O(n)$ elements of a row. Each of these operations take $O(n)$ time. Hence, the entire procedure consists of $\Theta(n)$ steps of $O(n)$ complexity each, and its parallel run time is $O(n^2)$. Since n processes are used, the cost is $O(n^3)$, which is of the same order as the sequential complexity of Gaussian elimination. Hence, the pipelined version of parallel Gaussian elimination with 1-D partitioning of the coefficient matrix is cost-optimal.