Parallelization: Sieve of Eratosthenes

By Aaron Weeden, Shodor Education Foundation, Inc.

Module Document

Overview

This module presents the sieve of Eratosthenes, a method for finding the prime numbers below a certain integer. One can model the sieve for small integers by hand. For bigger integers, it becomes necessary to use a coded implementation. This code can be either serial (sequential) or parallel. Students will explore the various forms of parallelism (shared memory, distributed memory, and hybrid) as well as the scaling of the algorithm on multiple cores in its various forms, observing the relationship between run time of the program and number of cores devoted to the program. Two assessment rubrics, two exercises, and two student project ideas allow the student to consolidate her/his understanding of the material presented in the module.

Model

A **positive integer** is a number greater than 0 that can be written without a fractional or decimal component. 1, 2, 3, 4, etc. are all examples.

A **positive divisor** is a positive integer that divides another integer without leaving a remainder. For example, 2 is a positive divisor of 6, since 6 / 2 = 3 remainder 0.

A **natural number** is a number used for counting or ordering. 0, 1, 2, 3, etc. are all examples.

A **prime number** (or **prime**) is a natural number whose only positive divisors are 1 and itself. The prime numbers below 12 are 2, 3, 5, 7, and 11.

A **composite number** (or **composite**) is a natural number that is not prime, i.e. one that has at least 1 divisor other than 1 and itself. Examples are 4 (2 is a divisor), 6 (2 and 3 are divisors), 8 (2 and 4 are divisors), and 12 (2, 3, 4, and 6 are divisors).

In this module we are interested in the question, "How do we find all the primes under a certain integer?" One method is to use the **sieve of Eratosthenes**. To see how the sieve works, we can follow the steps below, using the example of finding the prime numbers under 16.

Quick Review Questions

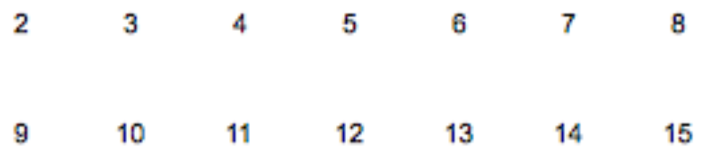
1. What are the positive divisors of 12?
2. What are the primes below 10?

Algorithm Example

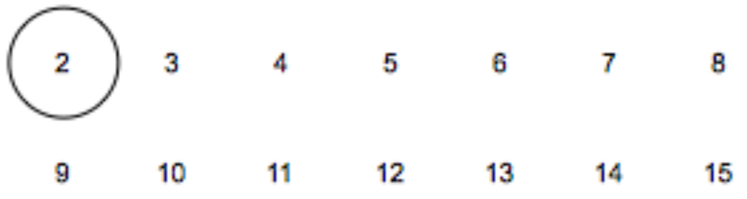
1. Write out the numbers from 2 to 15.
2. Circle the smallest unmarked, uncircled number in the list.
3. For each number bigger than the biggest circled number, mark the number if it is a multiple of the biggest circled number.
4. Repeat steps 2-4 until all numbers have been circled or marked. The circled numbers will be the primes; the marked numbers will be the composites.

Below is a more detailed description of what happens in pictures.

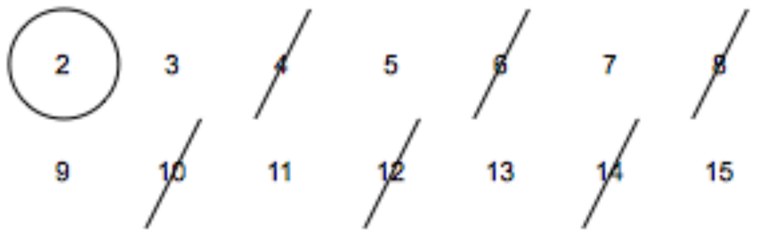
1. Write out the numbers 2 to 15.



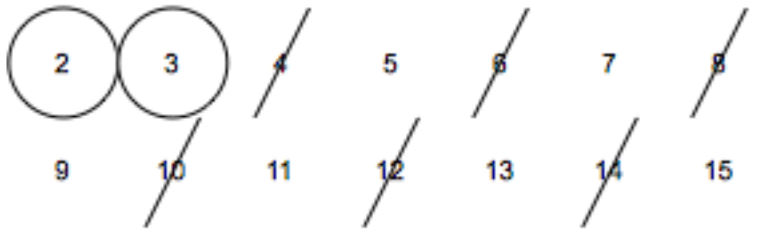
2. Circle the smallest unmarked, uncircled number, which in this case is 2.



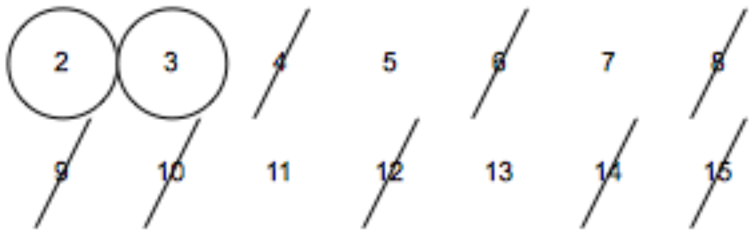
3. For each number bigger than 2, if the number is a multiple of 2, mark it.



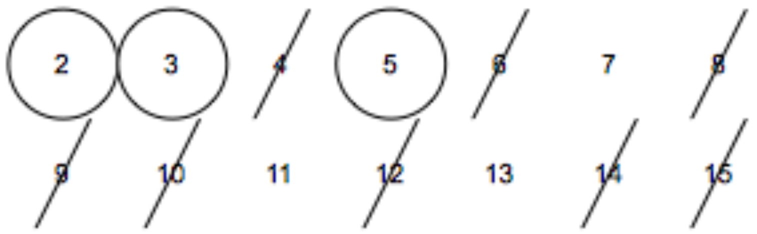
4. Circle the smallest unmarked, uncircled number, which in this case is 3.



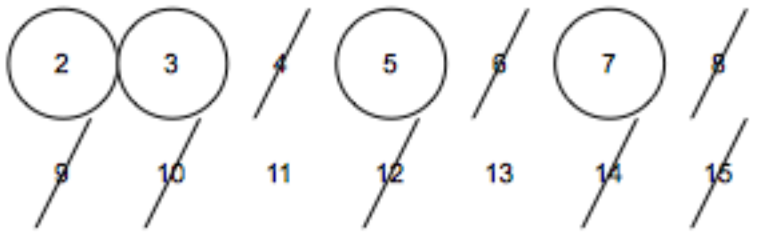
5. For each number bigger than 3, if the number is a multiple of 3, mark it.



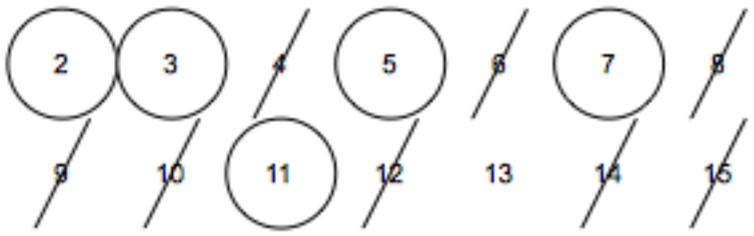
6. Circle the smallest unmarked, uncircled number, which in this case is 5.



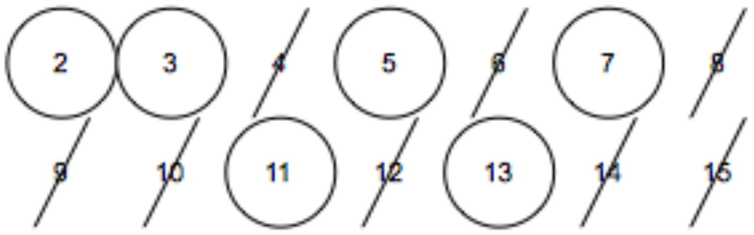
7. For each number bigger than 5, if the number is a multiple of 5, mark it. Notice that all multiples of 5 have already been marked. Circle the smallest unmarked, uncircled number, which in this case is 7.



7. For each number bigger than 7, if the number is a multiple of 7, mark it. Notice that all multiples of 7 have already been marked. Circle the smallest unmarked, uncircled number, which in this case is 11.



8. For each number bigger than 11, if the number is a multiple of 11, mark it. Notice that all multiples of 11 have already been marked. Circle the smallest unmarked, uncircled number, which in this case is 13.



9. For each number bigger than 13, if the number is a multiple of 13, mark it. Notice that all multiples of 13 have already been marked.

All numbers have now been circled or marked, so we have finished the algorithm. The prime numbers less than 16 are the circled numbers: 2, 3, 5, 7, 11, and 13. The composite numbers less than 16 are the marked numbers: 4, 6, 8, 9, 10, 12, 14, and 15.

We can model the sieve by hand for small numbers like 16. For much larger numbers, it becomes necessary to find a more streamlined way of doing so, such as through automation using a computer. The next few sections present a way to code the sieve so that a computer simulation is possible. This code is **serial**, i.e. non-parallel. The next sections after that will explore a parallel code to accomplish the same task.

Serial Code

This section explains the serial code that is attached to this module, `sieve.serial.c` , found in the `sieve` directory that is created by extracting the `sieve.zip` file. The section presents snippets of code followed by explanations of the snippets.

```
/* Declare variables */
int N = 16; /* The positive integer under which we are
               finding primes */
int sqrtN = 4; /* The square root of N, which is stored
                  in a variable to avoid making
                  excessive calls to sqrt(N) */
int c = 2; /* Used to check the next number to be
              circled */
int m = 3; /* Used to check the next number to be
              marked */
int *list; /* The list of numbers – if list[x] equals
              1, then x is marked.  If list[x] equals
              0, then x is unmarked. */
char next_option = ` `; /* Used for parsing command
                           line arguments */
```

We initialize **`N`** with the same arbitrary example value that we used in the Algorithm Example section above, `16` . As we'll see below, the user can change this value at runtime by passing a command line argument to the program.

The values of **`sqrtN`** , **`c`** , and **`m`** will be overwritten later, but best practice is to initialize all variables.

`list` will be an array, but we do not yet know how big it will be (again because **`N`** 's value can be changed below). Since we do not know what size it will be, but we do know what type of elements it will contain, we declare it as a pointer to `int` .

`next_option` is used in the code snippet below.

```
/* Parse command line arguments -- enter 'man 3
   getopt' on a shell for more information */
while((next_option = getopt(argc, argv, "n:"))
         != -1) {
```

If the user does not wish to use the default value of **N**, she/he can choose a different value by passing it to the program on the command line using the `-n` option. For example, running `./sieve.serial -n 100` would find the primes under `100`. This is achieved through the **getopt()** function.

getopt() takes three arguments: the first two arguments to main (**argc**, which is the count of arguments, and **argv**, which is the list of arguments) and a string of possible options (**"n:"** in this case). The string **n:** means that the user can pass `-n` followed by an argument.

getopt() scans each argument in **argv**, one at a time, and returns it. In our code, we catch this return value and store it in the **next_option** variable so that we can use it in the **switch**/**case** statement.

getopt() returns **-1** if it has run out of options to scan in **argv**. Our **while** loop will run until **getopt()** returns **-1**, i.e. until there are no more options to scan in **argv**.

```
switch(next_option) {
        case 'n':
            N = atoi(optarg);
            break;
```

The value we stored in **next_option** is used in the **switch**/**case** statement. If **next_option** is **'n'**, then we set **N** to **optarg**, which stores the value that followed `-n` on the command line. This must be converted from a **string** to an **int**, so we use **atoi(optarg)**, which converts **optarg** to an **int**.

```
default:
            fprintf(stderr, "Usage: %s [-n N]\n",
                    argv[0]);
            exit(-1);
    }
}
```

`-n` is the only option we want to make available. If the user enters any other option (i.e. in the **default** case), we print a usage message to the user, which tells her/him the correct way to run the program. We then exit the program with an error using **exit(-1)**.

Quick Review Questions

3. What would the command look like for running the program to find the primes under 5000?
4. What would happen if the user entered "`./sieve.serial -p`" on the command line?

```
/* Calculate sqrtN */
sqrtN = (int)sqrt(N);
```

To calculate the square root of **`N`**, we use the **`sqrt()`** function, which returns a value with type **`double`**. We cast this value as type **`int`** instead and assign it to the **`sqrtN`** variable.

```
/* Allocate memory for list */
list = (int*)malloc(N * sizeof(int));
```

`list` is declared as a pointer. To make it useable as an array, we must allocate memory for it. This can be achieved using the **`malloc()`** function. We want to allocate enough memory for **`N`** positive integers, each of which has a certain size, determined by the **`sizeof`** function. So, we pass **`N * sizeof(int)`** to **`malloc()`**. **`malloc()`** allocates that memory and returns a pointer to it. We cast this pointer (i.e. change its data type) to **`int`** so that it matches the data type of **`list`**, and we assign this new pointer to **`list`**.

```
 /* Exit if malloc failed */
 if(list == NULL) {
     fprintf(stderr, "Error: Failed to allocate memory
for list.\n");
     exit(-1);
 }
```

`malloc()` should be able to allocate this memory, but it will occasionally fail, especially if there is not enough memory available. If **`malloc()`** does fail, we want to halt the execution of our program (using **`exit(-1)`**) and notify the user that something went wrong.

```
/* Run through each number in the list */
for(c = 2; c <= N-1; c++) {

    /* Set each number as unmarked */
    list[c] = 0;
}
```

We set a number as unmarked by putting a `0` in its place in the `list` array. To set the number represented by `c` as unmarked, we use `list[c] = 0` .

To set `c` unmarked for all values of `c` , we use a `for` loop with `c` ranging from `2` to `N-1` .

Quick Review Question

5. At this point in the program, if the value of **N** is 8, what will be the value of `list[7]`?

```
/* Run through each number up through the square root
   of N */
for(c = 2; c <= sqrtN; c++) {

    /* If the number is unmarked */
    if(list[c] == 0) {
```

If we want to check each number in the list to see if it prime, we can loop from `2` to `N-1` with a different value of `c` each time. However, we can save time by only going through `sqrt(N)` . Recall that in the Algorithm Example we were looking for primes under 16. We started out by circling 2 and marking all its multiples, then circling 3 and marking all its multiples, then circling 5, etc. At some point, we came across a number that had no multiples less than 16 that were not already marked, namely 5. If we stopped at this point and just circled all of the remaining unmarked numbers, we would still have the correct list of primes as a result. The reason for this is that if the square root of `N` is multiplied by itself, we get `N` , which is already bigger than `N-1` . We do not need to multiply the square root of `N` by anything smaller, because these numbers have already been circled and their multiples found. We also do not need to multiply the square root of `N` by anything bigger, because these multiples will all be bigger than `N` , therefore bigger than `N-1` , i.e. bigger than any number that we need to check to be a multiple of `c` . Thus, we only need to loop `c` from `2` to `sqrtN` .

We know a number is prime if it is unmarked. Recall that we indicate a number is unmarked by setting it as such: `list[c] = 0` . We can check if it is unmarked through the `if(list[c] == 0)` statement.

Here is where we would circle the number, but we can indicate that a number is circled by simply keeping it unmarked in the array. At the end of the program, the numbers that are unmarked will be the same as the ones that would be circled if we performed the algorithm by hand. The circles help us when we are working by hand to keep track of which number we should use for the value of `c` . Since `c` is always increasing in the `for` loop, we do not have to worry about the computer losing track of this, and we do not need to do anything to "circle" the number.

```
                    /* Run through each number bigger than c
                     */
                    for(m = c+1; m <= N-1; m++) {

                        /* If m is a multiple of c */
                        if(m%c == 0) {

                            /* Mark m */
                            list[m] = 1;
                        }
                    }
                }
            }
```

To mark a number, we put a `1` in its place in `list` . `list[m] = 1` will mark number `m` .

We only want to mark `m` if it is a multiple of the number that is currently circled, `c` . To check if this is the case, we can use the **modulo operator**, `%` . The modulo operator returns the remainder of a division of integers. For example, `7%3` would return `1` , since `7/3` is `2` remainder `1` . If the remainder is `0` , then the division was even. For example, `6%2` is `3` remainder `0` . So, `m%c == 0` is true if `m` is a multiple of `c` .

We want to run through each number bigger than `c` to find multiples. The smallest number bigger than `c` is `c+1` , and the biggest in the sieve is `N-1` . We use a `for` loop from `c+1` to `N-1` to cover all values of `m` .

```
/* Run through each number in the list */
    for (c = 2; c <= N-1; c++) {

        /* If the number is unmarked */
        if(list[c] == 0) {

            /* The number is prime, print it */
            printf("%d ", c);

        }
    }
    printf("\n");
```

To print a number, we use the **printf()** function. This takes a string followed by the arguments that are referenced within this string. In this example, **%d** references **c** . We print all the numbers that are unmarked; this is our list of primes.

When we are finished printing numbers, we print a newline with **printf("\n")** so that the shell prompt will start on a new line when the program finishes running.

```
    /* Deallocate memory for list */
    free(list);

    return 0;
```

Just as we allocated memory for **list** , we must also deallocate it by calling the **free()** function, which allows other processes to use the memory that was being used by **list** . This follows the general rule that every **malloc()** should be matched with a **free()** .

When the program finishes, we **return** from the **main()** function with **0** to indicate that the program finished successfully.

Running the Serial Code

The serial code can be compiled with the GNU compiler by entering `gcc -o sieve.serial sieve.serial.c -lm` in a shell. This creates an executable called `sieve.serial`, which can be run in the shell by entering `./sieve.serial`. Confirm that the program works for a few values of **N** by using `./sieve.serial -n` followed by your choice of **N**. An example of this is shown below.

```
$ gcc -o sieve.serial sieve.serial.c -lm
$ ./sieve.serial -n 10
2 3 5 7
$ ./sieve.serial -n 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
$ ./sieve.serial -n 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97
$ ./sieve.serial -n 1000
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97 101 103 107 109 113 127 131 137 139
149 151 157 163 167 173 179 181 191 193 197 199 211
223 227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349 353 359 367
373 379 383 389 397 401 409 419 421 431 433 439 443
449 457 461 463 467 479 487 491 499 503 509 521 523
541 547 557 563 569 571 577 587 593 599 601 607 613
617 619 631 641 643 647 653 659 661 673 677 683 691
701 709 719 727 733 739 743 751 757 761 769 773 787
797 809 811 821 823 827 829 839 853 857 859 863 877
881 883 887 907 911 919 929 937 941 947 953 967 971
977 983 991 997
$
```

The next few sections provide an introduction to parallelism, parallel hardware, and the motivations for parallelism, followed by discussions of the parallel codes for sieve of Eratosthenes.

Introduction to Parallelism

In parallel processing, rather than having a single program execute tasks in a sequence (like the tasks of the algorithm above), parts of the program are instead split such that the program is executed **concurrently** (i.e. at the same time), by multiple entities.

The entities that execute the program can be called either **threads** or **processes** depending on how memory is mapped to them.

In **shared memory** parallelism, **threads** share a memory space among them. Threads are able to read and write to and from the memory of other threads. The standard for shared memory considered in this module is OpenMP, which uses a series of **pragmas**, or directives for specifying parallel regions of code in C, C++ or Fortran to be executed by threads.

In contrast to shared memory parallelism, in **distributed memory** parallelism, **processes** each keep their own private memories, separate from the memories of other processes. In order for one process to access data from the memory of another process, the data must be communicated, commonly by a technique known as **message passing**, in which the data is packaged up and sent over a network. One standard of message passing is the **Message Passing Interface (MPI)**, which defines a set of functions that can be used inside of C, C++ or Fortran codes for passing messages.

A third type of parallelism is known as **hybrid**, in which both shared and distributed memory are utilized. In hybrid parallelism, the problem is broken into tasks that each process executes in parallel. The tasks are then broken further into subtasks that each of the threads execute in parallel. After the threads have executed their sub-tasks, the processes use the shared memory to gather the results from the threads. The processes use message passing to gather the results from other processes.

In the next section we explore the type of hardware on which processes and threads can run.

Quick Review Questions:

6. What is the name for entities that share memory? For those with distributed memory?
7. What is **message passing** and when is it needed?

Parallel Hardware

In order to use parallelism, the underlying hardware needs to support it. The classic model of the computer, first established by John von Neumann in the 20th century, has a single CPU connected to memory. Such an architecture does not support parallelism because there is only one CPU to run a stream of instructions. In order for parallelism to occur, there must be multiple processing units running multiple streams of instructions. **Multi-core** technology allows for parallelism by splitting the CPU into multiple compute units called cores. This model works well for shared memory parallelism because the cores will often share RAM. Parallelism can also exist between multiple **compute nodes**, which are computers connected by a network. This requires distributed memory parallelism, since each compute node has its own RAM. Compute nodes may themselves have multi-core CPUs, which allow for hybrid parallelism: shared memory among the cores and message passing between the compute nodes.

Quick Review Questions:

8. Why is parallelism impossible on a von Neumann computer?
9. What is the difference between a **core** and a **compute node**?

Motivation for Parallelism

We now know what parallelism is, but why should we use it? The three motivations we will discuss here are **speedup**, **accuracy**, and **weak scaling**. These are all compelling advantages for using parallelism, but some also exhibit certain limitations that will also be discussed.

Speedup is the idea that a program will run faster if it is parallelized as opposed to executed serially. The advantage of speedup is that it allows a problem to be solved faster. If multiple processes or threads are able to work at the same time, the work will theoretically be finished in less time than it would take a single instruction stream.

Accuracy is the idea of forming a better model of a problem. If more processes or threads are assigned to a task, they can spend more time doing error checks or other forms of diagnostics to ensure that the final result is a better approximation of the problem that is being solved. In order to make a program more accurate, speedup may need to be sacrificed.

Weak scaling is perhaps the most promising of the three. Weak scaling says that more processes and threads can be used to solve a bigger problem in the same amount of time it would take fewer processes and threads to solve a smaller problem. A common analogy to this is that one person in one boat in one hour can catch much fewer fish than ten people in ten boats in one hour.

There are issues that limit the advantages of parallelism; we will address two in particular. The first, **communication overhead**, refers to the time that is lost waiting for communications to take place in between calculations. During this time, valuable data is being communicated, but no progress is being made on executing the algorithm. The communication overhead of a program can quickly overwhelm the total time spent solving the problem, sometimes even to the point of making the program less efficient than its serial counterpart. Communication overhead can thus mitigate the advantages of parallelism.

A second issue is described in an observation put forth by Gene Amdahl and is commonly referred to as **Amdahl's Law**. Amdahl's Law says that the speedup of a parallel program will be limited by its **serial regions**, or the parts of the algorithm that cannot be executed in parallel. Amdahl's Law posits that as the number of processors devoted to the problem increases, the advantages of parallelism diminish as the serial regions become the only part of the code that take significant time to execute. In other words, a parallel program can only execute as fast as its serial regions. Amdahl's Law is represented as an equation below.

$$\text{Speedup} = \frac{1}{(1-P)+\frac{P}{N}}\text{, where}$$

P = the time it takes to execute the parallel regions
1 – P = the time it takes to execute the serial regions
N = the number of processors

Amdahl's Law shows us that a program will have diminishing returns in terms of speedup as the number of processors is increased. However, it does not place a limit on the weak scaling that can be achieved by the program, as the program may allow for bigger classes of problems to be solved as more processors become available. The advantages of parallelism for weak scaling are summarized by John Gustafson in Gustafson's Law, which says that bigger problems can be solved in the same amount of time as smaller problems if the processor count is increased. Gustafson's Law is represented as an equation below.

$$\text{Speedup}(N) = N - (1 - P) * (N - 1)$$

where
N = the number of processors
(1 – P) = the time it takes to execute the serial regions

Amdahl's Law reveals the limitations of what is known as **strong scaling**, in which the problem size remains constant as the number of processors is increased. This is opposed to **weak scaling**, in which the problem size per processor remains constant as the number of processors increases, but the overall problem size increases as more processors are added. These concepts will be explored further in an exercise.

After reading about parallelism and its motivations, students should be ready to do Exercise 1, which takes the student through logging into a cluster, writing a small piece of parallel code, and submitting it to a **scheduler**, which manages the programs that are being run on the cluster. Exercise 1 is included as an attachment to this module.

After completing the exercise, students can explore the parallel algorithms of the sieve of Eratosthenes. We start with the OpenMP algorithm.

Quick Review Questions:

10. What is Amdahl's Law? What is Gustafson's Law?
11. What is the difference between **strong scaling** and **weak scaling**?

OpenMP Algorithm

In a parallel algorithm using shared memory, we want multiple threads to work simultaneously. It is not enough to have one thread executing tasks in order. We must identify procedures that are able to take place at the same time.

Let us first look back at the Algorithm Example we used earlier:

1. Write out the numbers from 2 to 15.
2. Circle the smallest unmarked, uncircled number in the list.
3. For each number bigger than the biggest circled number, mark the number if it is a multiple of the biggest circled number.
4. Repeat steps 2-4 until all numbers have been circled or marked. The circled numbers will be the primes; the marked numbers will be the composites.

Which of these steps can be made parallel? We can answer this question by looking at the steps and thinking about their dependencies. For example, step 1 can be made parallel; one thread writing a number does not depend on another thread writing a different number. We can rewrite the step this way:

1. In parallel, threads write out the numbers from 2 to 15.

This rephrases the step in language that supports shared memory parallelism. Numbers can be written to memory without worry of overwriting other numbers, since each number is assigned its own chunk of memory.

Can step 2 be made parallel? No. There is only one smallest unmarked, uncircled number in the list at any given time, and only one thread needs to circle it at a time. This part is in a serial region of the algorithm; it cannot be made parallel.

What about step 3? This can be made parallel. One thread marking one multiple of a number does not depend on another thread marking another multiple of the number. We can rewrite the step this way:

3. In parallel, threads mark the multiples of the biggest circled number.

Can step 4 be made parallel? No, because it contains a section that must be serial, namely step 2. Thus, we can conclude that only steps 1 and 3 can be made parallel.

How does this look in the code? It is a simple addition to the serial code; two lines are added, both of which are `#pragma omp parallel for` . If this line is placed above a `for` loop, the loop will be split up among the threads, and each thread will execute a different iteration of the loop. For example:

```
    /* Run through each number in the list */
#pragma omp parallel for
    for(c = 2; c <= N-1; c++) {

        /* Set each number as unmarked */
        list[c] = 0;
    }
```

In this loop, each thread is assigned a different value of `c` , which the thread uses to complete an iteration of the loop. Once the thread finishes, if there are any more iterations of the loop to complete, the thread is assigned another value of `c` . This continues until all iterations of the loop have been finished.

This code can be compiled using `gcc -fopenmp -o sieve.openmp sieve.openmp.c -lm` . Note the `-fopenmp` option; this tells the compiler to enable OpenMP. The number of OpenMP threads used by the program can be adjusted by changing the `OMP_NUM_THREADS` environment variable (for example by entering `export OMP_NUM_THREADS=2` in a BASH shell to specify two threads). The OpenMP program is run the same way as the serial program using `./sieve.openmp` . Confirm that the program works for a few values of **N** by using `./sieve.openmp -n` followed by your choice of **N** . An example of this is shown below.

```
$ gcc -fopenmp -o sieve.openmp sieve.openmp.c -lm
$ ./sieve.openmp -n 10
2 3 5 7
$ ./sieve.openmp -n 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
$ ./sieve.openmp -n 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97
$ ./sieve.openmp -n 1000
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97 101 103 107 109 113 127 131 137 139
149 151 157 163 167 173 179 181 191 193 197 199 211
223 227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349 353 359 367
373 379 383 389 397 401 409 419 421 431 433 439 443
449 457 461 463 467 479 487 491 499 503 509 521 523
541 547 557 563 569 571 577 587 593 599 601 607 613
```

```
617 619 631 641 643 647 653 659 661 673 677 683 691
701 709 719 727 733 739 743 751 757 761 769 773 787
797 809 811 821 823 827 829 839 853 857 859 863 877
881 883 887 907 911 919 929 937 941 947 953 967 971
977 983 991 997
```

MPI Code

An MPI version of the code is similar to OpenMP, but it uses processes and distributed memory instead of threads and shared memory. Distributed memory requires a few more considerations than shared memory when programming. These are discussed below.

One consideration is that each process will receive a copy of the same program. The compiled program will be executed in its entirety by each process. This differs from shared memory parallelism, in which threads are spawned only at certain times during the program, followed by sections in which only one thread is executing.

All threads are able to modify the `list` array in parallel in this code snippet that uses shared memory:

```
/* Run through each number in the list */
#pragma omp parallel for
    for(c = 2; c <= N-1; c++) {

        /* Set each number as unmarked */
        list[c] = 0;
    }
```

With distributed memory, the `list` array of one process cannot be accessed by another process. Each process will need to work on its own `list` array.

All processes must find all the primes in the numbers `2` through `sqrtN` . After this, they can split up the range of numbers above `sqrtN` to find the rest of the primes. In our distributed memory code, we use two different lists to handle these two different tasks. `list1` is designated for the numbers `2` through `sqrtN` . `list2` contains a section of the remaining numbers.

The size of a process's `list2` array will depend on how we split up the remaining numbers among the processes. If we wish to make as even a split as possible, we can simply divide the count of numbers by the count of processes. We determine the count of numbers by subtracting the smallest number from the largest number and adding 1 (for example, in the range 2 to 4, there are 3 numbers, which we can obtain by 4-2+1). In our case this count will be `(N-1)-(sqrtN+1)+1`, which simplifies to `N-(sqrtN+1)`. We then divide this number by the number of processes, `p`, to obtain `(N-(sqrtN+1))/p`. We can call this number `S`, for "split size".

If things do not divide evenly, some remainder will result. This remainder can be obtained with the modulus operator (`%`). The remainder of the split will be `(N-(sqrtN+1))%p`. We can call this number `R`, for "remainder". What do we do with this remainder? It must be assigned to one of the processes. As we'll see below, it can be advantageous to assign it to the last process so the arithmetic is a little easier to work with.

Let's try our splitting up of numbers on an example. Assume we want to find the primes under 100 and we have 3 processes. Each process is assigned a split size of `(100-(sqrt(100)+1))/3`, which simplifies to `29`. Rank 2 is the last process, so it would be assigned a remainder of `(100-(sqrt(100)+1))%3`, which simplifies to `2`.

What are the bounds of `list2`? Each process will have a lowest and highest number in its split. We can call these values `L` and `H`, respectively. We know that `L` must be at least as big as `sqrtN+1`. This will be the `L` of Rank 0, since Rank 0 is responsible for the first split. Rank 0 is responsible for `S` numbers, so its `H` will be `L+S-1`, or `sqrtN+1+S-1`, or just `sqrtN+S`. Rank 1 is responsible for the next number after this, so its `L` will be `sqrtN+S+1`. It is also responsible for `S` numbers, so its `H` will be `L+S-1`, or `sqrtN+S+1+S-1`, or just `sqrtN+2*S`. Rank 2 is responsible for the next number after this, so its `L` will be `sqrtN+2*S+1`. Its `H` will be `L+S-1`, or `sqrtN+2*S+1+S-1`, or just `sqrtN+3*S`. The pattern in general is that Rank `r`'s `L` will be `sqrtN + r*S + 1`, and its `H` will be `L+S-1`. Finally, since the last process (which has rank `p-1`) is responsible for the remainder (`R`), we add it to its `H` using `H += R`.

One last consideration to make is that only Rank 0 will be printing values. In order for Rank 0 to print the values in each `list2` of the other processes, those processes will each need to send Rank 0 a message containing its `list2`. This happens at the end after all multiples have been marked.

With all these considerations in mind, let us look how the distributed memory code, `sieve.mpi.c`, differs from the serial and OpenMP versions.

Quick Review Questions

For the following 2 questions, assume there are 4 processes and we are trying to find the primes under 16 using the MPI version of the program.

12. What will be the biggest number contained in `list1` ?
13. What will be the values for `S` and `R` ?
14. What will be the values of `L` and `H` for each of the processes?

```
int *list1; /* The list of numbers <= sqrtN -- if
               list1[x] equals 1, then x is
               marked.  If list1[x] equals 0, then x is
               unmarked. */
int *list2; /* The list of numbers > sqrtN – if
               list2[x-L] equals 1, then x is marked.
               If list2[x-L] equals 0, then x is
               unmarked. */
int S = 0; /* A near-as-possible even split of the
              count of numbers above sqrtN */
int R = 0; /* The remainder of the near-as-possible
              even split */
int L = 0; /* The lowest number in the current
              process's split */
int H = 0; /* The highest number in the current
              process's split */
int r = 0; /* The rank of the current process */
int p = 0; /* The total number of processes */
```

These are the new variables that have been added to the MPI version of the code. You can review what they represent by reading back over the previous few paragraphs of this document.

```
/* Initialize the MPI Environment */
MPI_Init(&argc, &argv);
```

Before we can do anything with MPI, we must first initialize the environment. **`MPI_Init()`** must be called before any other MPI functions can be called. We pass it **`argc`** and **`argv`** so that it can strip out command line arguments that are relevant to it. We must pass these by address using the ampersand (**&**) operator.

```
    /* Determine the rank of the current process and
           the number of processes */
        MPI_Comm_rank(MPI_COMM_WORLD, &r);
        MPI_Comm_size(MPI_COMM_WORLD, &p);
```

After we have initialized MPI, we can call our first MPI functions, **`MPI_Comm_rank()`** and **`MPI_Comm_size()`** , which will help us determine the values of the **`r`** and **`p`** variables, respectively. The first argument of each of these, **`MPI_COMM_WORLD`** , specifies the **communicator** of the MPI processes. This is the group of processes among which the current process is able to send and receive messages. **`MPI_COMM_WORLD`** is the default value and refers to all the processes.

Once these two functions have been called, we will have values for the rank of the current process and the total number of processes that we can use later.

```
/* Calculate S, R, L, and H */
S = (N-(sqrtN+1)) / p;
R = (N-(sqrtN+1)) % p;
L = sqrtN + r*S + 1;
H = L+S-1;
if(r == p-1) {
    H += R;
}
```

`S` , **`R`** , **`L`** , **and `H`** are calculated as explained above.

```
/* Allocate memory for lists */
list1 = (int*)malloc((sqrtN+1) * sizeof(int));
list2 = (int*)malloc((H-L+1) * sizeof(int));
```

`list1` will contain all the numbers less than **`sqrtN+1`** . **`list2`** will contain the process's split of the rest of the numbers, the size of which we can now represent as **`H-L+1`** .

```
/* Run through each number in list1 */
    for(c = 2; c <= sqrtN; c++) {

        /* Set each number as unmarked */
        list1[c] = 0;
    }

    /* Run through each number in list2 */
    for(c = L; c <= H; c++) {

        /* Set each number as unmarked */
        list2[c-L] = 0;
    }
```

We run through each number in each list and set it as unmarked. Note that we set number **`c`** in **`list2`** by setting element **`c-L`** . Why do we do this? Consider Rank 0. If it is responsible for marking number **`sqrtN+1`** first, it will want to do so in element **`0`** of the array, not in element **`sqrtN+1`** . If we subtract **`L`** from **`sqrtN+1`** , we get the value we want, **`0`** . In general, numbers **`L`** through **`H`** are stored in elements **`0`** through **`H-L`** of **`list2`** , and number **`c`** will be stored in element **`c-L`** of **`list2`** .

Quick Review Questions

For the following 2 questions, assume there is 1 process and we are finding the primes under 16 using the MPI version of the program.

15. In which list will the number 3 be stored? At which position?
16. In which list will the number 8 be stored? At which position?

```
/* Run through each number in list1 */
  for(c = 2; c <= sqrtN; c++) {

      /* If the number is unmarked */
      if(list1[c] == 0) {

          /* Run through each number bigger than c in
             list1 */
          for(m = c+1; m <= sqrtN; m++) {

              /* If m is a multiple of c */
              if(m%c == 0) {

                  /* Mark m */
                  list1[m] = 1;
              }
          }

          /* Run through each number bigger than c in
             list2 */
          for(m = L; m <= H; m++)
          {
              /* If m is a multiple of C */
              if(m%c == 0)
              {
                  /* Mark m */
                  list2[m-L] = 1;
              }
          }
      }
  }
```

As in the serial and OpenMP code, we start by running through each number less than **sqrtN** . The difference here is that we have two lists in which to mark multiples instead of just one. We mark all the multiples less than or equal to **sqrtN** in **list1** and all the multiples greater than **sqrtN** in **list2** .

```
/* If Rank 0 is the current process */
   if(r == 0) {

       /* Run through each of the numbers in list1 */
       for(c = 2; c <= sqrtN; c++) {

           /* If the number is unmarked */
           if(list1[c] == 0) {

               /* The number is prime, print it */
               printf("%lu ", c);
           }
       }
```

We are now splitting the code such that certain sections will be executed by Rank 0 and other sections will be executed by the other processes. We do this by surrounding the sections of code with `if(r == 0)` and `else` statements. When the process encounters these statements, it will only execute the code that matches its rank; so Rank 0 will execute the `if(r == 0)` section (since its rank, `r`, is `0`), and the others will execute the `else` section.

Rank 0 is responsible for printing the prime numbers. The code above shows how it prints the primes in `list1`.

```
       /* Run through each of the numbers in list2 */
       for(c = L; c <= H; c++) {

           /* If the number is unmarked */
           if(list2[c-L] == 0) {

               /* The number is prime, print it */
               printf("%lu ", c);
           }
       }
```

The code above shows how Rank 0 prints the primes from its own `list2`.

```
/* Run through each of the other processes */
      for(r = 1; r <= p-1; r++) {

          /* Calculate L and H for r */
          L = sqrtN + r*S + 1;
          H = L+S-1;
          if(r == p-1) {
              H += R;
          }

          /* Receive list2 from the process */
          MPI_Recv(list2, H-L+1, MPI_INT, r, 0,
                   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Rank 0 is now ready to receive each `list2` from the other processes. To do so, it runs through each of their ranks. Based on the rank, it calculates the `L` and `H` of the process just as the process calculated its own `L` and `H`. It then uses the `MPI_Recv()` function to receive `list2` from the process. Let us examine each of the arguments to this function, in order.

`list2`, the first argument, is known as the **receive buffer**. It is the area of memory into which the message should be stored. This argument must be a pointer to memory. This works for `list2` because we declared `list2` as a pointer to `int`.

The second argument is the size of the message being sent. In this case, we want to send the entirety of `list2`, whose size is `H-L+1` (this is the first reason we needed to calculate `L` and `H` for the sending process).

The third argument is the MPI data type of the elements of the receiving buffer. The elements of `list2` are of type `int`, so we use the MPI data type `MPI_INT`.

The fourth argument is the rank of the sending process. In our case, that rank is `r`.

The fifth argument is the **tag** of the message. In MPI programs that use multiple types of messages, it can be helpful to distinguish them using a tag, a unique value that indicates what type of message is being received. We only have one type of message in this program, so we just use the arbitrary value `0` for the tag.

The sixth argument is the communicator of the processes that are sending and receiving. In our case this is `MPI_COMM_WORLD`. Recall that this is also the value we used for the `MPI_Comm_rank()` and `MPI_Comm_size()` functions.

The seventh argument is the **status** of the message. This reports whether the message was received in error. It can be helpful to catch this value and take appropriate action, but the value can also be ignored, as it is in this case, by using `MPI_STATUS_IGNORE`.

After this function has been called, Rank 0's **`list2`** will be replaced by the **`list2`** that was just received from rank **`r`** .

Quick Review Question

17. Assume we are Rank 3 and trying to receive a message in a custom communicator called **`MPI_OUR_COMM`** from Rank 4. We want to receive **`3`** elements in an array called **`list`**. The elements should be of type **`MPI_FLOAT`**. We don't care about the status of the message we receive, and the tag of the message should be **`5`**. What would the call to **`MPI_Recv()`** look like?

```
        /* Run through the list2 that was just
           received */
        for(c = L; c <= H; c++) {

            /* If the number is unmarked */
            if(list2[c-L] == 0) {

                /* The number is prime, print it */
                printf("%lu ", c);
            }
        }
```

Rank 0 can now print the prime values it just received from rank **`r`** . After looping over all the processes, all of the prime numbers will have been received and printed.

```
    /* If the process is not Rank 0 */
    } else {

        /* Send list2 to Rank 0 */
        MPI_Send(list2, H-L+1, MPI_INT, 0, 0,
                 MPI_COMM_WORLD);
    }
```

The processes that are not Rank 0 simply send their **`list2`** s to Rank 0 using **`MPI_Send()`** . The arguments to this function are essentially the same as in **`MPI_Recv()`** . One notable difference is that the fourth argument of **`MPI_Send()`** is the rank of the receiving process, not the sending process. It is also worth noting that **`MPI_Send()`** does not have a status argument.

The tags, MPI data types, and communicators of the **`MPI_Send()`** and **`MPI_Recv()`** functions must match in order for the message to go through from the sender to the receiver. The **`MPI_Send()`** function is a **non-blocking** call, which means that the program will continue once the message has been sent. By contrast, **`MPI_Recv()`** is a blocking call, so the program will not continue until the message is received. Each **`MPI_Recv()`** must have a matching **`MPI_Send()`** , otherwise the program will hang.

```
/* Deallocate memory for list */
free(list2);
free(list1);
```

Since we allocated both **`list1`** and **`list2`** , we need to deallocate them using **`free()`** . It is a good idea to deallocate memory in the reverse order it was allocated to avoid freeing bytes of memory multiple times, which would cause an error in the program.

```
/* Finalize the MPI environment */
MPI_Finalize();
```

The last step before exiting is to finalize MPI. Just as we initialized MPI at the beginning of the program using **`MPI_Init()`** , we finalize MPI at the end of the program using **`MPI_Finalize()`** . No more MPI functions are allowed to be called after **`MPI_Finalize()`** has been called.

The MPI code can be compiled using `mpicc -o sieve.mpi sieve.mpi.c -lm` . Note that we are using `mpicc` in this case instead of `gcc` . `mpicc` is a wrapper around `gcc` that includes the MPI libraries and linkers.

To run the MPI program, we use the `mpirun` command. We pass this command the number of processes with which we wish to run using the `-np` argument. For example, if we wished to run the program with 2 processes, we would use the command `mpirun -np 2 ./sieve.mpi` . We can still pass `sieve.mpi` the `-n` argument in conjunction with `mpirun` . Confirm that the program works for a few values of **N** and for multiple processes by using `mpirun -np <something> ./sieve.mpi -n` followed by your choice of **N** . An example of this is shown below.

```
$ mpicc -o sieve.mpi sieve.mpi.c -lm
$ mpirun -np 2 ./sieve.mpi -n 10
2 3 5 7
$ mpirun -np 4 ./sieve.mpi -n 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

```
$ mpirun -np 10 ./sieve.mpi -n 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97
$ mpirun -np 1 ./sieve.mpi -n 1000
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97 101 103 107 109 113 127 131 137 139
149 151 157 163 167 173 179 181 191 193 197 199 211
223 227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349 353 359 367
373 379 383 389 397 401 409 419 421 431 433 439 443
449 457 461 463 467 479 487 491 499 503 509 521 523
541 547 557 563 569 571 577 587 593 599 601 607 613
617 619 631 641 643 647 653 659 661 673 677 683 691
701 709 719 727 733 739 743 751 757 761 769 773 787
797 809 811 821 823 827 829 839 853 857 859 863 877
881 883 887 907 911 919 929 937 941 947 953 967 971
977 983 991 997
```

Hybrid Code

The Hybrid code is written by making the same small changes to the MPI code that were made to the serial code to write the OpenMP code. In particular, we parallelize the loops that can be run in parallel.

You can find these loops by searching the `sieve.hybrid.c` source code for **`#pragma omp parallel for`** .

The hybrid code can be compiled using `mpicc -fopenmp -o sieve.hybrid sieve.hybrid.c -lm` . Note that we are again using `mpicc` , and that the `-fopenmp` argument has been added.

The number of OpenMP threads used by the program can once again be adjusted by changing the `OMP_NUM_THREADS` environment variable. We run the hybrid program as we do the MPI program, using `mpirun` . Confirm that the program works for a few values of **N** and for multiple processes by using `mpirun -np <number of processes> ./sieve.hybrid -n` followed by your choice of **N** . An example of this is shown below.

```
$ mpicc -fopenmp -o sieve.hybrid sieve.hybrid.c -lm
$ mpirun -np 2 ./sieve.hybrid -n 10
2 3 5 7
$ mpirun -np 4 ./sieve.hybrid -n 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
$ mpirun -np 10 ./sieve.hybrid -n 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97
$ mpirun -np 1 ./sieve.hybrid -n 1000
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
71 73 79 83 89 97 101 103 107 109 113 127 131 137 139
149 151 157 163 167 173 179 181 191 193 197 199 211
223 227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349 353 359 367
373 379 383 389 397 401 409 419 421 431 433 439 443
449 457 461 463 467 479 487 491 499 503 509 521 523
541 547 557 563 569 571 577 587 593 599 601 607 613
617 619 631 641 643 647 653 659 661 673 677 683 691
701 709 719 727 733 739 743 751 757 761 769 773 787
797 809 811 821 823 827 829 839 853 857 859 863 877
881 883 887 907 911 919 929 937 941 947 953 967 971
977 983 991 997
```

Now that we have explored our source code we can determine how it will scale. Students should complete Exercise 2, an attachment to this module.

Scaling Discussion

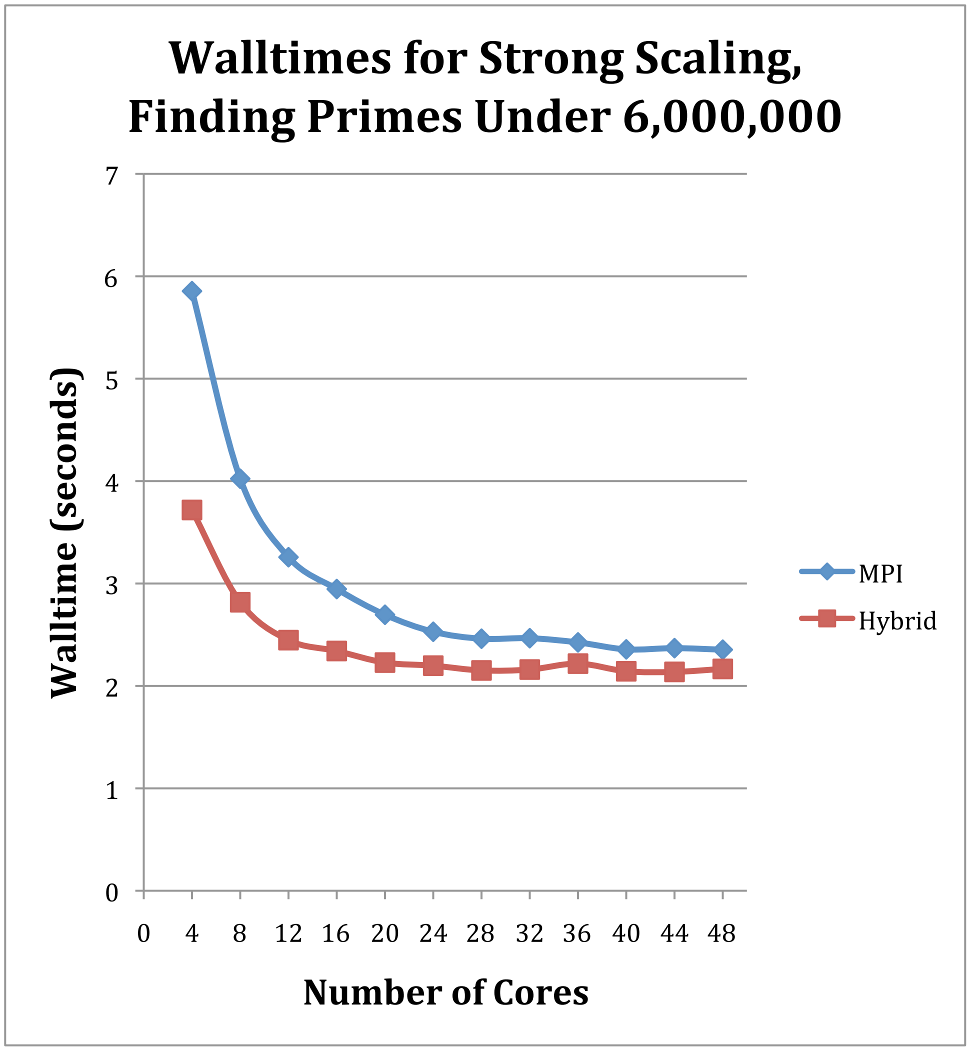
Exercise 2 is likely to produce tables that look something like the following. We have also included graphs of the data for the MPI and hybrid runs to show the trends that occur when more cores are added.

Walltimes for Strong Scaling, Finding Primes Under 6,000,000

# of nodes used	Total # of cores	Serial	OpenMP	MPI	Hybrid
1	4	0m21.919s	0m2.839s	0m5.855s	0m3.718s
2	8			0m4.023s	0m2.817s
3	12			0m3.257s	0m2.447s
4	16			0m2.946s	0m2.341s
5	20			0m2.695s	0m2.228s
6	24			0m2.529s	0m2.198s
7	28			0m2.461s	0m2.151s
8	32			0m2.466s	0m2.160s
9	36			0m2.425s	0m2.218s
10	40			0m2.357s	0m2.143s
11	44			0m2.369s	0m2.137s
12	48			0m2.354s	0m2.166s

There are a few things to notice from this data. First of all, the speedup is dramatic when moving from serial to parallel: from 21 seconds to about 2 seconds for about a 10x speedup. Note also that until about 20 cores, there is no advantage to using MPI over OpenMP to achieve speedup for this problem size; at 20 cores, the time to execute the MPI code drops below the time to execute the OpenMP code. The hybrid code becomes advantageous even sooner, at about 8 cores.

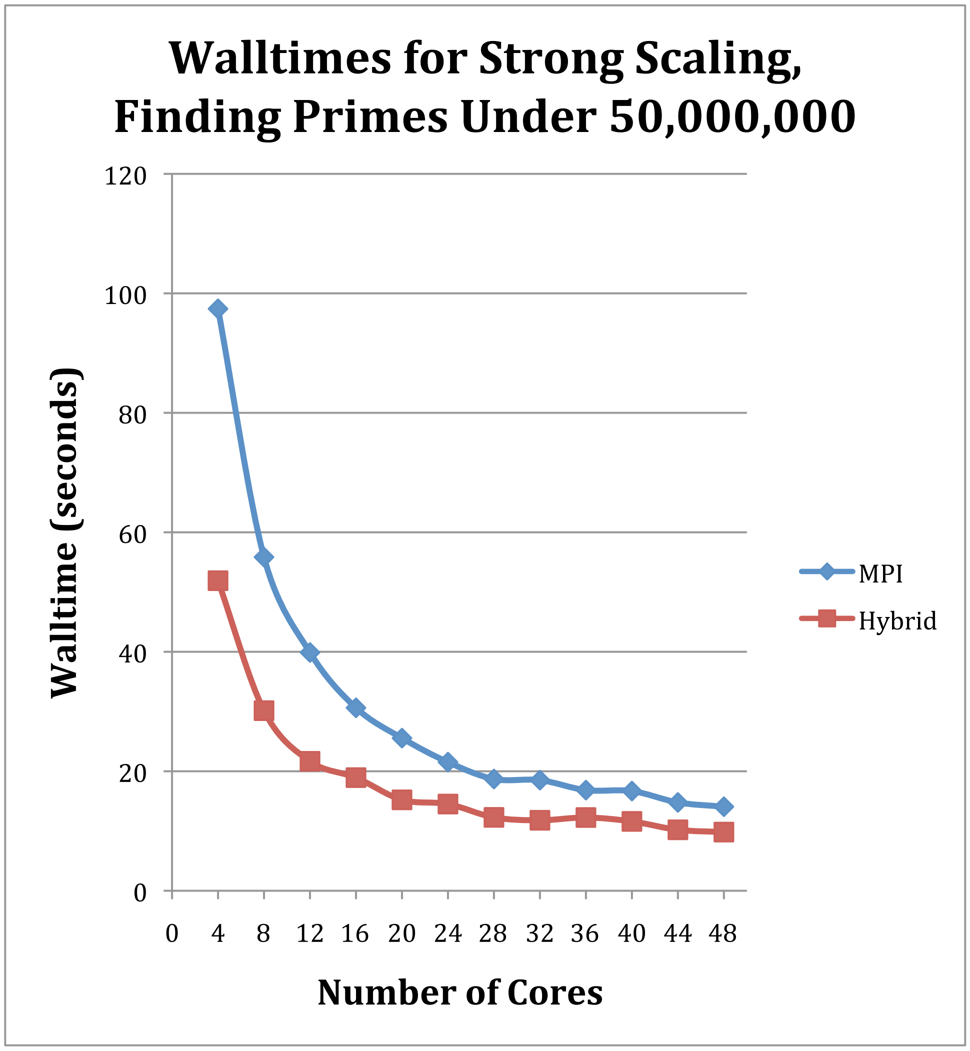
A graph of the data is shown below.



Note the trend of this graph: a dramatic decrease in walltime followed by a much more level decrease. This is indicative of Amdahl's Law. We are adding more cores to the problem, but we are not seeing dramatic increases in speedup. In fact, up to 48 cores, we have not yet seen the walltime dip below 2 seconds. This would suggest that for this problem size, the serial regions of the code require about 2 seconds to execute in total. We can see a similar trend for larger problem sizes, as indicated by the data and chart below.

Walltimes for Strong Scaling, Finding Primes Under 50,000,000

# of nodes used	Total # of cores	Serial	OpenMP	MPI	Hybrid
1	4	6m35.128s	0m56.825s	1m37.416s	0m51.900s
2	8			0m55.842s	0m30.146s
3	12			0m39.880s	0m21.666s
4	16			0m30.621s	0m18.937s
5	20			0m25.549s	0m15.216s
6	24			0m21.532s	0m14.512s
7	28			0m18.702s	0m12.279s
8	32			0m18.517s	0m11.798s
9	36			0m16.845s	0m12.258s
10	40			0m16.689s	0m11.605s
11	44			0m14.780s	0m10.202s
12	48			0m14.065s	0m9.820s

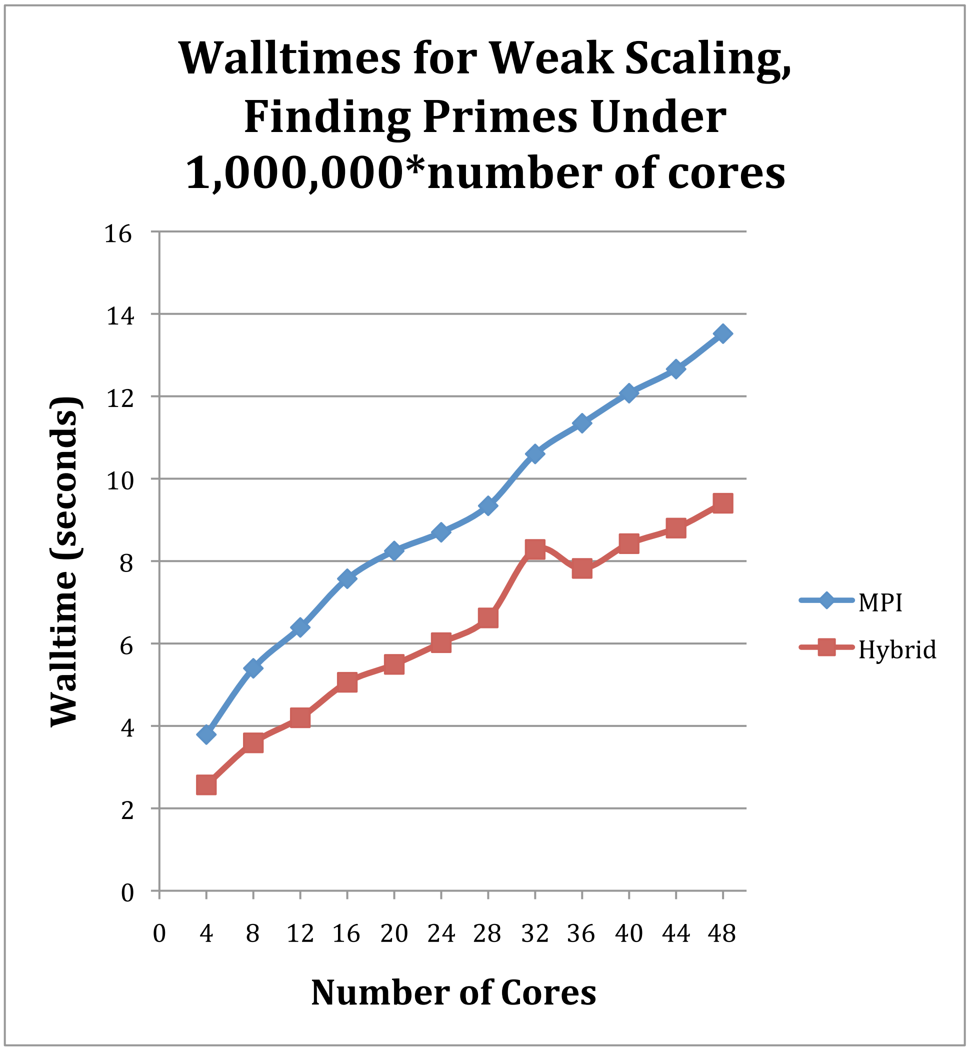


Below are example data and a graph for the weak scaling part of Exercise 2.

Walltimes for Weak Scaling, Finding Primes Under 1,000,000 * Number of Cores

# of nodes used	Total # of cores	Finding primes under	Serial	OpenMP	MPI	Hybrid
1	4	4,000,000	0m10.616s	0m1.598s	0m3.790s	0m2.569s
2	8	8,000,000			0m5.398s	0m3.589s
3	12	12,000,000			0m6.390s	0m4.198s
4	16	16,000,000			0m7.572s	0m5.059s
5	20	20,000,000			0m8.246s	0m5.493s
6	24	24,000,000			0m8.699s	0m6.020s
7	28	28,000,000			0m9.342s	0m6.621s
8	32	32,000,000			0m10.601s	0m8.283s
9	36	36,000,000			0m11.346s	0m7.822s
10	40	4,000,000			0m12.075s	0m8.423s
11	44	44,000,000			0m12.660s	0m8.801s
12	48	48,000,000			0m13.520s	0m9.404s

Keep in mind that with weak scaling we are increasing the problem size proportionally to the increase in the number of cores. Because of communication overhead between the processes, the walltime will gradually increase as we add more cores. This is visible in the slow upward trend of the graph below.



We can solve problems ranging from sizes of 4,000,000 to 48,000,000 using 4-48 cores in under 14 seconds for MPI and in under 10 seconds for Hybrid. If we draw the trendline for each of the curves, we see that it takes roughly an extra 0.2 seconds per core for MPI and roughly an extra 0.15 seconds per core for Hybrid. This gives us an idea of how the problem scales, and about the general trend of the MPI program versus the Hybrid program. Ideally, these lines would be flat, but communication overhead forces them to have a gradual trend upward.

Student Project Ideas

1. Explore strong scaling of the MPI and Hybrid codes for different problem sizes. Is it true for each problem size that at 20 cores MPI becomes more advantageous than OpenMP? That at 8 cores Hybrid becomes more advantageous than OpenMP? See if you can find the problem size for which it is most advantageous to use MPI over OpenMP (the one for which it requires the fewest cores for MPI to be faster than OpenMP) by adjusting the problem size and observing the results. Do the same for Hybrid over OpenMP.

2. Gather walltime data for the serial and OpenMP versions of the code for each of the problem sizes in the weak scaling exercise. Graph the data for serial, OpenMP, MPI, and Hybrid. Perform a regression analysis on each set of data to determine the slope of each trend line. How does this value differ for each line? What does this tell you about the scaling of each version of the code?