Homework 2

GROUP 7: Samantha Cheng, Pratik Mistry, Aditya Singh Thakur, Vikhyat Dhamija

Problem 1B

Part i:

I think Footprint version 2 of the Grid-Solver is correct because it correctly trades off footprints for the barriers of version 1. Because, in version 1 we had only one diff variable to store the state while multiple barriers so that all the threads work collaboratively on the same same state at any given time. But, this leads to slowness of the entire program because fast running threads will have to wait for slow running threads each time it encounters the barrier. Moreover, the principle of parallelism gets destroyed when the wait time is significantly huge.

Thus, adding one barrier and three different states via diff[] would solve the problem to far greater extent. Adding one barrier before the convergence check happens is the perfect place where all threads can come together after updating their respective states and performing computations and then start over again.

Now, variable mydiff is local to all the threads and doesn't collide with any threads. Thus, it will be a different copy on which each thread will use to get to the new states. Now, talking about maintaining the states, we have 3 positions/indexes which we can use for updating states in diff[]. The number 3 is very sufficient for storing the states in diff variable and we will see the consequences if the states/indexes are lower or more states in part ii of this problem.

For the time being, consider we have threads 1,2,3,4 where 1 is a slow running thread, 2 and 3 is a bit faster than 1 and 4 is the fastest thread. Now, while they start running the program, upto 1st encounter of the barrier every thread will be together waiting for checking the convergence. Further, as thread 4 is fast, it will finish the remaining statements and update the index to 2 while threads 1,2 and 3 still work on checking the convergence for index 1. Furthermore, say thread 4 performs the computation to accumulate their mydiff locally, and variable 2 and 3 finish checking convergence and update their index to 2 while thread 1 still checks the convergence for index 1. Thus, as soon as thread 4 adds own mydiff state into its current index i.e. 2 and initializes diff variable for index 3 while thread 1 still works on checking convergence for index 1, and thread 2 and 3 working on state index 2. Thus, after thread 4 initializes the 3rd state to 0, it waits at the barrier for other threads. This is how, number 3 is very useful as a fast running thread won't overwrite the state of a slow running thread and neither will the memory be wasted.

Part ii:

Exactly 3 states would be very sufficient for storing the states in diff variable because of the nature of multithreading programming that we have used for solving the Grid-Solver problem.

If we have fewer states to be maintained then there is a very high possibility that diff variable of a particular index used by very slow running threads could be overwritten or initialized to 0 by the fast running threads. This is severely affected because we initialize state to 0 as seen before the barrier and fast running threads often overwrite or initialize its next state to 0 while that state would still be used by slower running threads.

To practically explain the above point 2., consider we have threads 1, 2, 3, 4, 5, 6 and we have only 2 indexes to maintain 2 states inside the diff variable. If say threads 5 and 6 are the fastest among all, threads 2,3 and 4 are moderate speed and thread 1 is the slowest. Now, while they start running the program, upto 1st encounter of the barrier every thread will be together waiting for checking the convergence. Further, as 5 and 6 are fast, they will finish the remaining statements and update the index to 2 while threads 1,2,3,4 still work on checking the convergence for index 1. Furthermore, say 5 and 6 perform the computation to accumulate their mydiff locally, and variable 2,3 and 4 finish checking convergence and update their index to 2 while thread 1 still checks the convergence for index 1. Thus, as soon as thread 5 and 6 adds own mydiff state into its current index i.e. 2 and initializes diff variable for index 1 while thread 1 still works on checking convergence for index 1, then this is a huge problem. Thread 1 will now calculate wrong convergence because the diff variable for index 1 is initialized to 0 by thread 5 and 6. Thus, if we could have a third state i.e. index 3 (diff[3]) then thread 5 and 6 could have initialized the 3rd state and then wait at the barrier until all other threads finish their task and come at the same barrier.

Similarly, we could draw the same analogy for more states, for example 4 states. We know that if we have three states then the fast running thread could initialize the 3rd state and wait for other threads to finish their respective states at the barrier. Thus, the process repeats again after all threads start over again from the barrier but for the entire lifecycle only 3 states could be used at max and 1 state could be idle/wasted. Thus, the memory resources will waste in the end.

Part iii.
This is the most important point in the entire Grid-Solver problem because the barrier is a place where all threads start over again. Thus, irrespective of how fast the thread is, it will eventually have to wait at the barrier for slow running threads.

Now, if we don't have one barrier at all in our program then there is no effect on increasing the states i.e. diff[]. Because the fast running threads at some point will eventually overwrite/initialize the states in diff[] of slower running threads while they are still working on convergence. Thus, a lot of threads could lose important information of its own state and will eventually calculate the convergence for a different or wrong state. This will lead to bad behavior of the program and we will surely get incorrect results.

Problem 2

Part 1:

1. In the first case i..e blocked assignment, the arithmetic intensity per core will be ratio of number of computed elements to the number of communicated elements i.e.
   Arithmetic Intensity = (N*N/P) / (2N) = N / P.
   Now since we are generalizing it to having P processors, the total arithmetic intensity would be:
   Arithmetic Intensity = (N/P)*P = N

2. In second case i.e. interleaved assignment, the arithmetic intensity per core will be:
   Arithmetic Intensity = (N*N/P) / (2N*N/P) = ½.
   Now since we are generalizing it to having P processors, the total arithmetic intensity would be:
   Arithmetic Intensity = P/2.

3. For the 3rd case i.e. tile assignment, the arithmetic intensity per core:
   Arithmetic Intensity = (N / sqrt(P))^2 divided by 4N / sqrt(P) = N / (4 * sqrt(P)).
   Now since we are generalizing it to having P processors, the total arithmetic intensity would be:
   Arithmetic Intensity = (N * sqrt(P)) / 4.

4. The value of AI depends completely on N and P. If N is smaller than P / 2 i.e. AI of case 2 and (N*sqrt(P))/ 4 i.e. AI of case 3, then the AI of 1st blocked assignment would have the lowest AI because the first case only depends on N. But if we assume that interleaved assignment would typically have the lowest AI as the number of processors would normally be much smaller than the length of the grid and the 1st blocked assignment would typically be the largest as we need 16 processors in the tile assignment to even match the AI of the 1st block assignment

Part 2:

1. To start with we can take a system of linear equations such that it is equal to a different variable, such as variable 1, variable 2, variable 3, and so on and lets initialize all the variables equal to 0. Now after we compute the value of variable 1, we can go ahead to compute the value of variable 2 using the new value that was computed for variable 1. We keep on continuing this chain for subsequent variables i.e. variables 3,4,5,... etc.. We keep on doing this until we don't meet the convergence similar to Grid Solver using the Footprint version that we have seen. Once the convergence meets, we get the final result.
2. Now as we need previously updated values to compute the new values of the variables in each variable, we don't think there exists any other sequential method to achieve our goal.

Part 2B:

1. Both the variations of computing primality can be executed properly, however for the second one, any primality cannot be found. One way that we can consider is by considering the cache, as the cache can only hold 4 elements at a time (for example), we have each cache in line with the 4 grid elements and traverse down the grid to compute the primalities and average result, each processor handling a separate section of the grid, so we can maximize effective usage of the cache. This is very similar to what we had seen in the lectures where we had studied concepts like cache miss, cache conflicts and inherent and artifactual communication issues and how to overcome those via different strategies.
2. We can also achieve this via interleaved assignment, but we might waste some information from the edges, if N isn't completely divisible by 4. Thus, we won't get 100% accurate results and we can see that the former strategy would prove to be better while also dealing with caching problems - 4C's - cold hits, cache miss, cache conflicts and inherent and artifactual communications.

Part 3:

I. <u>Each of the processors P is dual core, hyper-threaded, with 4-way SIMD</u>.

Now since we have P processors which are dual core and hyperthreaded i.e. we have a total 4 threads that we can run in parallel each with 4-way SIMD capabilities. Thus at a time 16 independent work items can be processed.

A. Case 1: Blocked Assignment
    1. Suppose we have N*N cells in total, and we know we can only process (N*N/2*2) because we compute either red or black elements in one clock cycle among all the processors and only two threads can be running at a time in two cores. But since we have 4-way SIMD capability, the total elements that would be processed in a single clock cycle would be: 4*(N*N/4) = **N*N**.

B. Case 2: Interleaved Assignment
    1. Suppose we have N*N cells in total, and we know we can only process (N*N/2*2) because we compute either red or black elements in one clock cycle among all the processors and only two threads can be running at a time in two cores. But since we have 4-way SIMD capability, the total elements that would be processed in a single clock cycle would be: 4*(N*N/4) = **N*N**.
    2. This is the same as case 1 because we are not considering the communication costs and not computing arithmetic intensity here and it's just about how many elements are processed.

C. Case 3: Tile Assignment
    1. Suppose we have N*N cells in total, and we know we can only process (N*N/2*2) because we compute either red or black elements in one clock cycle among all the processors and only two threads can be running at a time in two cores. But since we have 4-way SIMD capability, the total elements that would be processed in a single clock cycle would be: 4*(N*N/4) = **N*N**.
    2. This is again the same as previous cases because we just want to find elements processed in one clock cycle rather among all the processors.

II. Each of the processors is single core, single-threaded, with no SIMD capabilities.
Now we have P processors which are single core and single-threaded and no SIMD capabilities.

The overall solution would be same for all the cases where number of elements processed at one clock cycle will always be **(N*N/2)** because we compute either red or black elements in one clock cycle among all the processors.

Problem 3

In order to compare the three machines for the specific workload , we will calculate how many groups of 16 elements each machine can process.

a) 4 Ghz single core CPU:
A group of 16 elements will take 376 (15x8+256) cycles on the single core system. Given the 4 GHz clock rate this machine can process 4/376 x 109 groups of 16 elements per second.

b) 1 GHz quad-core 4-wide SIMD CPU:
In this case, 16 elements will be scheduled in 4 sets of 4 elements in the cores. The first group requires 256 iterations through the loop (256 cycles), because of the divergent execution (3 of the 4 lanes only require 8 iterations). The last three groups suffer no divergence, and require 8 iterations (8 cycles) through the inner loop. Therefore each group of 16 elements requires (3*8 + 256) cycles. There are a total of four cores operating simultaneously at 1 GHz, so the machine can process (4/280) x 109 groups of 16 elements per second.

c) 1 GHz dual-core 16-wide SIMD CPU:
16 elements will take 256 cycles to execute on each core (every group must have to wait for the long-running element, so the 16-wide machine suffers from significant divergence). There are two 1GHz cores, so the machine can process (2/256) x 109 groups of 16 elements per second.

Thus , according to these results, the best performing core for this workload is the 1 GHz quad-core 4-wide SIMD CPU and the 1 GHz dual-core CPU will provide the lowest performance.

Problem 4

Part 1:
One main difference is in *calling*. Pthread_exit is called from the thread itself to terminate its execution early. Pthread_join is called from the thread that created it, which may be the main function, to wait for it to terminate and obtain its result.

Another main difference is in *returning*. Pthread_exit cannot return to its caller. Pthread_join returns 0 on success and an error number on error. Both return a result from the terminated thread via pointers.

Part 2:
a) We saw the total task count adding up to 3 more than the cap because of the way we allowed threads to access thread_data.next_task. We designated next_task as an atomic variable, which allows threads to have exclusive read/write access. However, in the while loop, we read the next_task, then give up the access, then take the access back when incrementing next_task. In the time it took between checking the value in the while condition and actually incrementing the variable, other threads were able to check the condition using the previous value. The 3 is not coincidental; while one thread was incrementing the next_task to the total task count, 3 other threads were also accessing the shared variable and entered their while loop.

b) Yes, thread_data.next_task plays a role in the situation beforehand (as described in part a). It is a shared variable, and has a synchronization primitive associated with it because it was defined as an atomic variable.

Assuming it is an atomic variable, then only one thread can access it at a time. If thread 1 is incrementing thread_data.next_task, and thread 2 tries to access it at the same time, thread 2 must wait for thread 1 to finish its incrementation.

Assuming it is not an atomic variable, then we may see race conditions if multiple threads attempt to access it at the same time. If thread 1 is incrementing thread_data.next_task, and thread 2 tries to access it at the same time, then thread 2 may read the original value before thread 1 finishes incrementing. Or, thread 2 reads the value after thread 1 finishes incrementing. The behavior is not well-defined due to these race conditions.

Problem 5

Q1

The dependencies in this problem are that in each iteration of the program, each row element depends on the element to the left, and each row depends on the previous row. However, there is independent work among the diagonals. Here are possible ways to parallelize our program:

- Partition the grid along the diagonal, parallelizing the tasks among diagonal grid cells before moving on to the next diagonal
- Update every other grid cell in parallel per row. We can do an interleaved assignment so that each processor works on every few rows.
- Again, update every other grid cell in parallel (imagine a checkerboard: update all the red cells in parallel, then all the black cells in parallel). We can do this in a blocked assignment, such as a few rows at a time. There is opportunity for exploiting locality here.

Q2

Maximum speedup = **1/ (s + (1-s)/p)** where s = fraction of total work that is inherently sequential and p = the number of processors.

This is derived from Ahmadl law that we have studied in the lecture where p is number of processors running some part of the code in parallel and s amount of sequential code execution while running parallel program. Thus, the denominator comes up as addition of sequential code execution plus the remaining code that is executed in parallel by p processors.

Q3
In the shared address space model, we use mutex locks to protect shared variables, and barriers to express dependencies between phases of computation.

```
Initialize global diff array
Solve function {
    Initialize thread local variables
    While not done:
        Perform computation and store into local variable
        Take lock
        Add local variable to global diff at index
        Release lock
        Barrier → all processors catch up here
        Check for convergence
}
```

Q4
In the message passing model, there are no shared variables.

- Send and receive ghost rows to neighbor threads; even-numbered threads send, then receive. Odd-numbered threads receive, then send.
- Perform computation
- All threads send result of local computation to thread 0
- Thread 0 performs takes results from other threads, checks for convergence, and sends result back to the other threads

Problem 6a

<u>HOW TO RUN THE CODE:</u>
The code is found in the file called “**hw2_6.c**”.

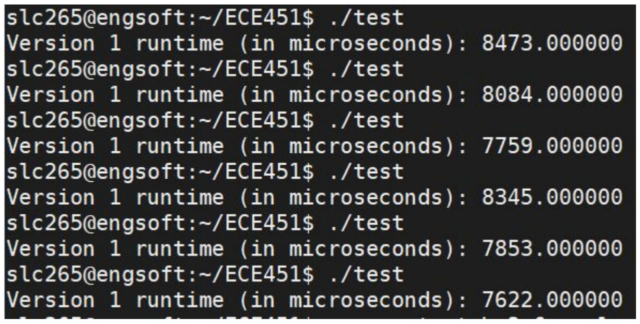
In main function:
Uncomment part1(), part2(), or part3() depending on which version of the problem you want to run. Only uncomment one at a time because they all use the same input array to read from and output array to write to, so the output runtime may be affected if you run multiple versions at once.

In terminal:
gcc -o test hw2_6.c -lm -lpthread
./test

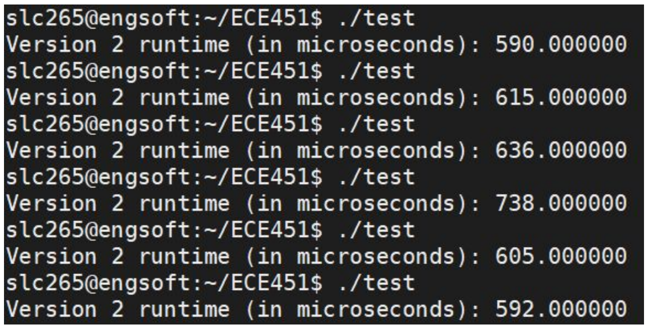
<u>Explanation of code:</u>
The function **oddRand()** populates the array with random odd numbers between 400 and 800. The output array is populated with 0 if the element from the random array is a composite number, 1 if it is a prime number, and 2 if it is a prime number and its reverse is also a prime number. This determination is done through the functions **isPrime, reversiblePrime,** and **getOutput**.

There are two mutex locks. One lock is used in all three versions of the problem in order to protect the shared **output** array. Another lock is used only in version 3 to protect a shared variable called **nextTask**, which determines which element a thread should perform its actions on next.

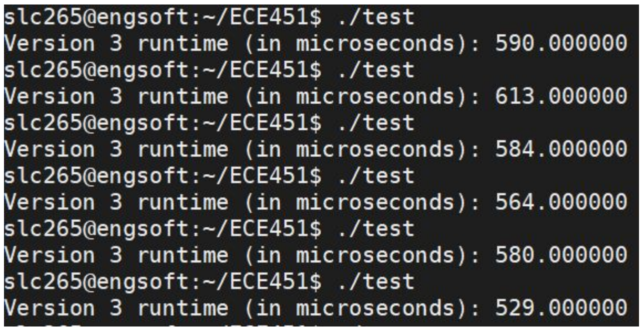
Q1: Versions 1 runs the slowest at around 8000 microseconds. Versions 2 and 3 are much faster, at around 500-600 microseconds. This makes sense because if each thread is only working on one array element, such as in version 1, there is a lot of context switching and the overhead associated with creating each pthread becomes very apparent. Using, for example, 4 threads makes more sense here because we can use each thread to take on multiple tasks, yet still use the multiple threads to execute in an "interleaved" fashion.



```
slc265@engsoft:~/ECE451$ ./test
Version 1 runtime (in microseconds): 8473.000000
slc265@engsoft:~/ECE451$ ./test
Version 1 runtime (in microseconds): 8084.000000
slc265@engsoft:~/ECE451$ ./test
Version 1 runtime (in microseconds): 7759.000000
slc265@engsoft:~/ECE451$ ./test
Version 1 runtime (in microseconds): 8345.000000
slc265@engsoft:~/ECE451$ ./test
Version 1 runtime (in microseconds): 7853.000000
slc265@engsoft:~/ECE451$ ./test
Version 1 runtime (in microseconds): 7622.000000
```



```
slc265@engsoft:~/ECE451$ ./test
Version 2 runtime (in microseconds): 590.000000
slc265@engsoft:~/ECE451$ ./test
Version 2 runtime (in microseconds): 615.000000
slc265@engsoft:~/ECE451$ ./test
Version 2 runtime (in microseconds): 636.000000
slc265@engsoft:~/ECE451$ ./test
Version 2 runtime (in microseconds): 738.000000
slc265@engsoft:~/ECE451$ ./test
Version 2 runtime (in microseconds): 605.000000
slc265@engsoft:~/ECE451$ ./test
Version 2 runtime (in microseconds): 592.000000
```



```
slc265@engsoft:~/ECE451$ ./test
Version 3 runtime (in microseconds): 590.000000
slc265@engsoft:~/ECE451$ ./test
Version 3 runtime (in microseconds): 613.000000
slc265@engsoft:~/ECE451$ ./test
Version 3 runtime (in microseconds): 584.000000
slc265@engsoft:~/ECE451$ ./test
Version 3 runtime (in microseconds): 564.000000
slc265@engsoft:~/ECE451$ ./test
Version 3 runtime (in microseconds): 580.000000
slc265@engsoft:~/ECE451$ ./test
Version 3 runtime (in microseconds): 529.000000
```

Q2: Version 3 took a similar amount of time as version 2, but appears slightly faster. The similar speed is due to the usage of the same number of pthreads (4), which is faster than using 100 pthreads as explained in Q1. Version 3 may be faster because the amount of computations done on each array element depends on how quickly we can determine whether it is prime or not. If, for example, the element is divisible by 2, then we are finished with the task very quickly. If it is a prime number, however, we have to check all numbers from 2 to the square root of the number for factors. Then, we need to check whether the reverse of the number is also prime. These computations would create a longer task. In version 2, one thread may encounter a block of the array that is all prime and requires more computations than another thread that has a block of composite numbers. The runtime would then be slowed down by the thread that takes longer time. In version 3, the tasks can be shared such that the workload is spread more evenly regardless of the number of tasks done by each thread.

Q3: Version 3 can be better than versions 1 and 2 when there are tasks of differing complexity. If they are all the same complexity, with similar computations, then version 2 is sufficient because the tasks are divided evenly, and thus the workload is too. If some tasks are much more complex than others, then version 3 excels because a thread that finishes a simple task can move on to the next task while another thread may still be working on a longer task.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#define numTasks 100
#define numWorkers 4

int output[numTasks];
int arr[numTasks];
pthread_mutex_t outputLock;
pthread_mutex_t taskLock;
```

```
int nextTask;

int isPrime( int n ) {
        int i;
        for ( i = 2; i <= ceil(sqrt(n)); i++ ) {
                if ( n%i == 0 ) {
                        return 0; // composite
                }
        }
        return 1; // prime
}

int reversiblePrime( int n ) {
        int dig1 = n / 100;
        int dig2 = (n % 100) / 10;
        int dig3 = n % 10;
        int reverse = 100*dig3 + 10*dig2 + dig1;
        if ( isPrime(reverse) ) {
                return 1;
        } else {
                return 0;
        }
}

int getOutput( int n ) {
        if ( isPrime(n) ) {
                if ( reversiblePrime(n) ) {
                        return 2; // prime forwards and backwards
                } else {
                        return 1; // prime
                }
        } else {
                return 0; // composite
        }
}

void* worker1( void *passArg ) {
        int ind = *((int*)passArg);
        int num = arr[ind];
        int out = getOutput(num);

        pthread_mutex_lock(&outputLock);
        output[ind] = out;
        pthread_mutex_unlock(&outputLock);
}
```

```
void part1() { // #1: Generate 100 pthreads
	struct timeval startTime, endTime;
	gettimeofday(&startTime,NULL);

	pthread_t thread;
	int i;
	int threadNum[numTasks];
	for ( i = 0; i < numTasks; i++ ) {
		threadNum[i] = i;
		pthread_create(&thread, NULL, &worker1, &threadNum[i]);
		pthread_join(thread, NULL);
	}

	gettimeofday(&endTime,NULL);
	float totalTime = (1000000 * endTime.tv_sec + endTime.tv_usec) - (1000000 * startTime.tv_sec +
startTime.tv_usec);
	printf("Version 1 runtime (in microseconds): %f\n",totalTime);
}

void* worker2( void *passArg ) {
	int ind = *((int*)passArg);
	int startIndex = ind*25;
	int endIndex = (ind+1)*25;
	int i;

	for ( i = startIndex; i < endIndex; i++ ) {
		int num = arr[i];
		int out = getOutput(num);
		pthread_mutex_lock(&outputLock);
		output[i] = out;
		pthread_mutex_unlock(&outputLock);
	}
}

void part2() { // #2: Generate 4 pthreads, 1 per block
	struct timeval startTime, endTime;
	gettimeofday(&startTime,NULL);

	int i;
	int threadNum[numWorkers];
	pthread_t thread[numWorkers];
	for ( i = 0; i < numWorkers; i++ ) {
		threadNum[i] = i;
		pthread_create(&thread[i], NULL, &worker2, &threadNum[i]);
	}
	for ( i = 0; i < numWorkers; i++ ) {
		pthread_join(thread[i], NULL);
	}

	gettimeofday(&endTime,NULL);
	float totalTime = (1000000 * endTime.tv_sec + endTime.tv_usec) - (1000000 * startTime.tv_sec +
startTime.tv_usec);
	printf("Version 2 runtime (in microseconds): %f\n",totalTime);
}

void* worker3( void *passArg ) {
	int currTask;
	int num;

	while ( nextTask < numTasks ) {
		pthread_mutex_lock(&taskLock);
		if ( nextTask < numTasks ) { // another check..
			currTask = nextTask; // index to check
			num = arr[currTask];
			nextTask++;
		}
		pthread_mutex_unlock(&taskLock);
		int out = getOutput(num);
		pthread_mutex_lock(&outputLock);
		output[currTask] = out;
		pthread_mutex_unlock(&outputLock);
	}
}

void part3() { // #3: Generate 4 pthreads, take next available element
	struct timeval startTime, endTime;
	gettimeofday(&startTime,NULL);

	int i;
	pthread_t thread[numWorkers];
	nextTask = 0;

	for ( i = 0; i < numWorkers; i++ ) {
		pthread_create(&thread[i], NULL, &worker3, NULL);
	}
	for ( i = 0; i < numWorkers; i++ ) {
		pthread_join(thread[i], NULL);
	}

	gettimeofday(&endTime,NULL);
	float totalTime = (1000000 * endTime.tv_sec + endTime.tv_usec) - (1000000 * startTime.tv_sec +
startTime.tv_usec);
```

```
        printf("Version 3 runtime (in microseconds): %f\n",totalTime);
}

void oddRand() { // return odd number between 400-800
        int i;
        for ( i = 0; i < numTasks; i++ ) {
                arr[i] = ( rand() % 200 ) * 2 + 401;
        }
}

int main(int argc, char** argv) {
        if ( pthread_mutex_init(&outputLock, NULL) != 0 ) {
                exit(1); // mutex failed
        }
        if ( pthread_mutex_init(&taskLock, NULL) != 0 ) {
                exit(1); // mutex failed
        }

        oddRand(); // populate arr with random numbers

        // UNCOMMENT ONE OF THE FOLLOWING AT A TIME:
        //part1();
        //part2();
        //part3();

        pthread_mutex_destroy(&outputLock); // destroy mutex
        pthread_mutex_destroy(&taskLock);
        return 0;
}
```