Ziting Liu
EECE5640
HW2

Q1: a)

The number of philosophers does not impact my solution in any way. More philosophers only mean that more philosophers can eat at the same time. More specifically N/2-1 philosophers can eat at the same time.

However even numbers of forks will impact my solution. There are basically three different cases: The first case is that some philosophers have more than one forks between them, then they don't have to check the status of each other. Second case will be that some philosophers have no fork between them, in which they will never eat. And the final case is that philosophers have a public fork in the middle. Third case is the one I implemented in code dining1.c, because it just makes more sense.

b)

When one philosopher has higher priority than others, she can eat whenever she feels like it and others will have to wait until she finishes. But generally, I believe the solution is not too different, the only difference is that one philosopher will eat more frequently.

c)

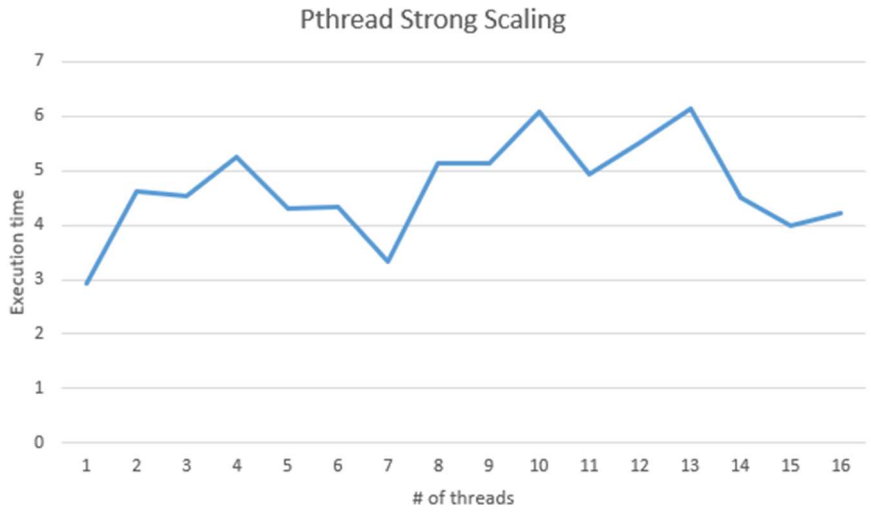
If the time it takes for each philosopher to eat is the same, the result is not different. As I implemented in dining3.c.

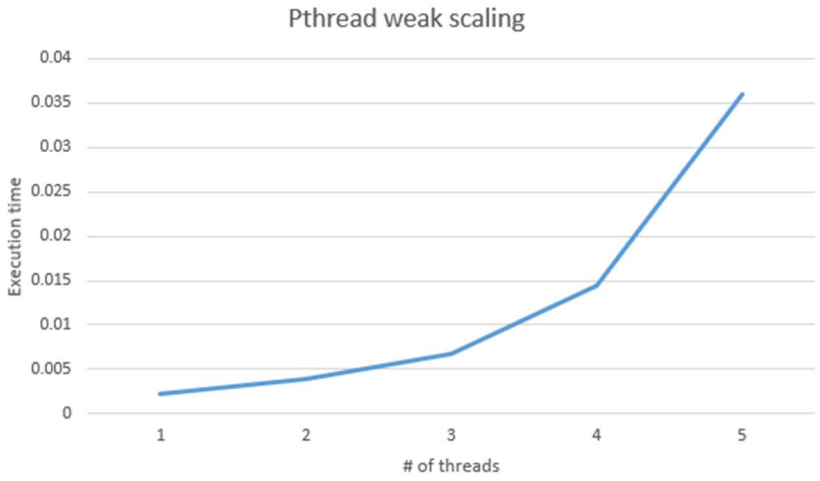
Q2:

Pthread:

I recorded the runtime needed for the pthread implementation to generate all prime numbers under 10^8. The average runtime with single thread is 2.92 sec. Average runtime with 16 threads is 4.21 sec. I believe the reason is creating threads and joining the threads are just wasting too much time. The code creates threads to remove all multiples of each number, and we are just creating too many threads. The execution time varies a lot on each run, I am taking average time over five runs for each of the test, but the result was still not good.

Strong scaling was tested with max number 10^8. Weak scaling was tested with 1 thread 50000 max, 2 threads 100000 max, 4 threads 200000 max, 8 threads 400000 max, and finally 16 threads 800000 max.

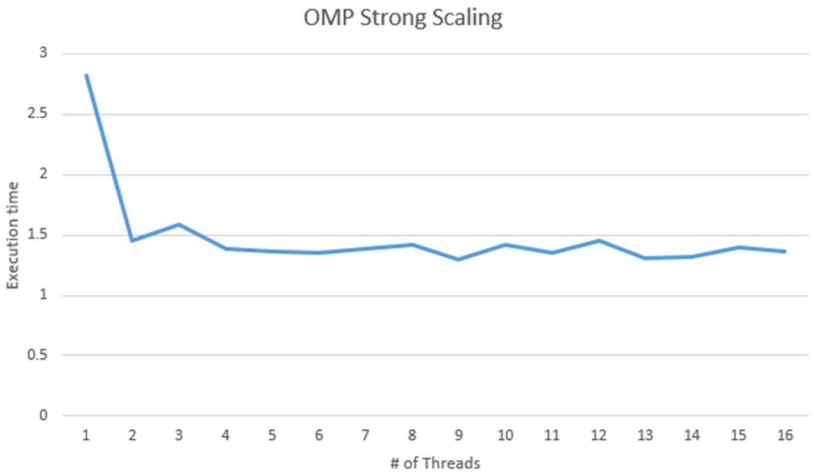


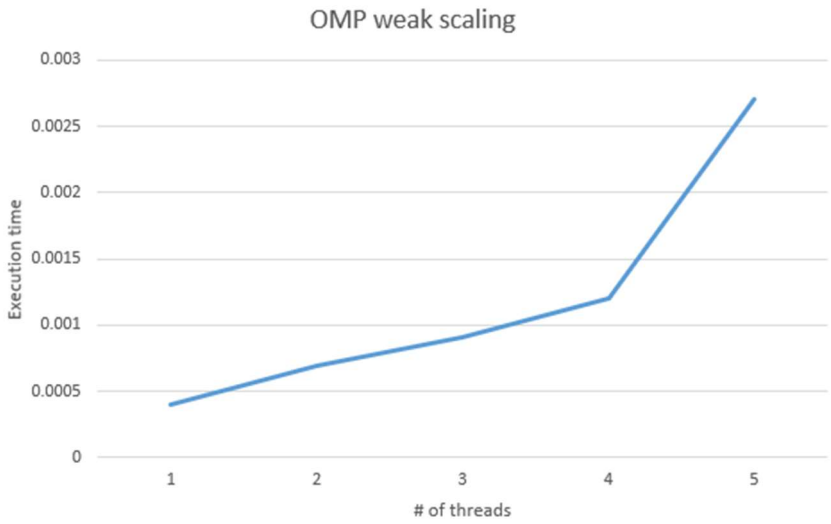


*** Although the X axis says 1,2,3,4,5, please refer them as 1,2,4,8,16.**

Open MP:

Then I recorded the runtime needed for the Open MP implementation to generate all prime numbers under 10^8. The average runtime with single thread is 2.82 sec. Average runtime with 16 threads is 1.44 sec. Unlike pthread, open MP will speed up the execution of the program. I believe the reason is Open MP can run each iteration of a for loop in parallel. Strong scaling was tested with a limit of 10^8. Weak scaling was tested with 1 thread 50000 max, 2 threads 100000 max, 4 threads 200000 max, 8 threads 400000 max, and finally 16 threads 800000 max.





*** Although the X axis says 1,2,3,4,5, please refer them as 1,2,4,8,16.**

Q3:

a)

Using Open MP will give user a linear increase in time when creating joining threads. However, LWTs are highly dependent on many other factors. The time needed is not easily predictable. Secondly, Open MP is relatively easier to use, the basic mechanism is to use pragma to tell the compiler what code can be executed in parallel.

The Open MP sometimes will be slowed down if the number of threads exceeds the number of threads available in the CPU. Also, Open MP has less control over the layers comparing to other LWTs.

From section "IX Evaluation", we can see that OMP with ICC compiler has the best performance in single loop when the number of thread used exceeds the machine limit, however has the worst performance before exceeding it.

b) learned Argobots from https://github.com/pmodels/argobots.

This code won't compile, it just shows how to create threads and do parallel addition.

Important part highlighted in yellow.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdarg.h>
#include <abt.h>

#define DEFAULT_NUM_XSTREAMS 2
#define DEFAULT_NUM_THREADS 8

typedef struct {
	int tid;
} thread_arg_t;

void add(vector<double> v1,vector<double> v2, int i)
{
	V1[i]+=v2[i];
}

int main(int argc, char **argv)
{
	vector<double> v1=some random double of size n
	vector<double> v2=some random double of size n
	int i;
	/* Read arguments. */
	int num_xstreams = DEFAULT_NUM_XSTREAMS;
	int num_threads = DEFAULT_NUM_THREADS;
	while (1) {
		int opt = getopt(argc, argv, "he:n:");
		if (opt == -1)
			break;
		switch (opt) {
			case 'e':
				num_xstreams = atoi(optarg);
				break;
			case 'n':
				num_threads = atoi(optarg);
				break;
			case 'h':
			default:
				printf("Usage: ./hello_world [-e NUM_XSTREAMS] "
					"[-n NUM_THREADS]\n");
				return -1;
		}
```

```
}
if (num_xstreams <= 0)
    num_xstreams = 1;
if (num_threads <= 0)
    num_threads = 1;

/* Allocate memory. */
ABT_xstream *xstreams =
    (ABT_xstream *)malloc(sizeof(ABT_xstream) * num_xstreams);
ABT_pool *pools = (ABT_pool *)malloc(sizeof(ABT_pool) * num_xstreams);
ABT_thread *threads =
    (ABT_thread *)malloc(sizeof(ABT_thread) * num_threads);
thread_arg_t *thread_args =
    (thread_arg_t *)malloc(sizeof(thread_arg_t) * num_threads);

/* Initialize Argobots. */
ABT_init(argc, argv);

/* Get a primary execution stream. */
ABT_xstream_self(&xstreams[0]);

/* Create secondary execution streams. */
for (i = 1; i < num_xstreams; i++) {
    ABT_xstream_create(ABT_SCHED_NULL, &xstreams[i]);
}

/* Get default pools. */
for (i = 0; i < num_xstreams; i++) {
    ABT_xstream_get_main_pools(xstreams[i], 1, &pools[i]);
}

/* Create ULTs. */
for (i = 0; i < n; i++) {
    int pool_id = i %;
    thread_args[i].tid = i;
    ABT_thread_create(pools[pool_id], add(v1,v2,i), &thread_args[i],
                      ABT_THREAD_ATTR_NULL, &threads[i]);
}

/* Join and free ULTs. */
for (i = 0; i < num_threads; i++) {
    ABT_thread_free(&threads[i]);
}
```

```
    /* Join and free secondary execution streams. */
    for (i = 1; i < num_xstreams; i++) {
        ABT_xstream_join(xstreams[i]);
        ABT_xstream_free(&xstreams[i]);
    }

    /* Finalize Argobots. */
    ABT_finalize();

    /* Free allocated memory. */
    free(xstreams);
    free(pools);
    free(threads);
    free(thread_args);

    return 0;
}
```

c)

BLAS (basic linear algebra subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. BLAS-1 perform scalar, vector, and vector-vector operations.

From http://www.netlib.org/blas/

d)

Nested parallel structures and nested tasks are an essential part in modern computation, Including image processing, deep neural network training, or simply massive data bases. These computations will take forever if we do them in serial. Therefore, these parallel patterns are common in HPC.