ECE 8400 / ENGI 9875 Lab 2

Xinyu Jian

TOTAL POINTS

25.5 / 26

QUESTION 1

Prelab 9 pts

1.1 pthread_t 2 / 2

✓ + 0.5 pts type

✓ + 0.5 pts Meaning

✓ + 0.5 pts Uniqueness inside process

✓ + 0.5 pts Reusability after thread termination

1.2 Thread safety 2 / 2

✓ + 0.5 pts Function Definition

✓ + 0.75 pts Thread-unsafe Example

✓ + 0.75 pts Thread-safe Example

1.3 increment() 2 / 2

✓ + 1 pts Proof of work

✓ + 1 pts Showing how the f_count field is considered in a thread safe implementation

1.4 foo_create() 1 / 1

✓ + 1 pts Proof of work

1.5 C blocks vs lambda functions 2 / 2

✓ + 1 pts Description: How a C block is behaved.

✓ + 1 pts Differences: Syntax, Data type, Variables etc

QUESTION 2

Serial execution 5 pts

2.1 Execution time (one invocation) 1 / 1

✓ + 0.5 pts What does the code do?

✓ + 0.5 pts Showing the output and execution time taken for each unit of work

2.2 Run-time statistics 2 / 2

✓ + 1 pts Analysis based on time, No. of executions & iterations

✓ + 1 pts Proof of work

2.3 Run-time varying with JOBS and WORK_PER_JOB 2 / 2

✓ + 1 pts Analysis of Avg running time as per SD, no of measurements etc

✓ + 1 pts Proof of work

QUESTION 3

3 POSIX threads 4 / 4

✓ + 1 pts Updated Code files & snippet for test run

✓ + 2 pts Proof of work: POSIX threads execution for given JOBS & WORK_PER_JOB

✓ + 1 pts throughput plot

QUESTION 4

4 libdispatch 4 / 4

✓ + 1 pts Updated Code files & snippet for test run

✓ + 2 pts Proof of work: threads execution for given JOBS & WORK_PER_JOB

✓ + 1 pts throughput plot

QUESTION 5

5 Race conditions 3.5 / 4

+ 0.5 pts Updated Code files & snippet for test run

✓ + 1 pts Proof of work: threads execution for varying JOBS & WORK_PER_JOB (Highlighted changes)

✓ + 1.5 pts Proof of work: correct value are counted as per modifications

✓ + 0.5 pts throughput plot

✓ + 0.5 pts Observations Discussed

Threading Lab Report - Xinyu Jian

Purpose and outcomes

The goal of this lab is to explore two models for Unix userspace multithreading: POSIX threads and libdispatch. By the end of this lab you should be able to:

1. use POSIX threading primitives to write multithreaded programs,
2. explore the kernel threading behaviour of such a program and
3. compare the performance of alternative threading models for different workloads.

Preparation

Prelab

Answer all of the following questions. Provide evidence for your claims.

1. What is the underlying type of a `pthread_t`? What is its meaning? When is a `pthread_t` unique? When may it be reused?

   On my system - Ubuntu 18.04.5 LTS, `phthread_t` is defined as follows:

   ```
   /* Thread identifiers.  The structure of the attribute type is not
      exposed on purpose.  */
   typedef unsigned long int pthread_t;
   ```

   A thread ID is represented by the `pthread_t` data type. But the implementations can use different structures, which is system independent [1].

   > - Linux 3.2.0 uses an unsigned long integer for the pthread_t data type.
   > - Solaris 10 represents the pthread_t data type as an unsigned integer.
   > - FreeBSD 8.0 and Mac OS X 10.6.8 use a pointer to the pthread structure for the pthread_t data type.

   Unlike the process ID, which is unique in the system, the thread ID has significance only within the context of the process to which it belongs [1]. According to the `man pthread_self`:

   > Thread IDs are guaranteed to **be unique only within a process**. A thread ID may be reused after a terminated thread has been joined, or a detached thread has terminated.

2. What is a *thread-safe* function? Provide an example implementation of a simple C function that is **not** thread-safe and an improved version that **is** thread-safe.

   > A function is said to be thread-safe if and only if it will always produce correct results when called repeatedly from multiple concurrent threads. [2]

   A class of thread-unsafe functions are functions that do not protect shared variables. Here is an example:

1.1 pthread_t **2 / 2**

- ✓ **+ 0.5 pts** type
- ✓ **+ 0.5 pts** Meaning
- ✓ **+ 0.5 pts** Uniqueness inside process
- ✓ **+ 0.5 pts** Reusability after thread termination

Threading Lab Report - Xinyu Jian

Purpose and outcomes

The goal of this lab is to explore two models for Unix userspace multithreading: POSIX threads and libdispatch. By the end of this lab you should be able to:

1. use POSIX threading primitives to write multithreaded programs,
2. explore the kernel threading behaviour of such a program and
3. compare the performance of alternative threading models for different workloads.

Preparation

Prelab

Answer all of the following questions. Provide evidence for your claims.

1. What is the underlying type of a `pthread_t`? What is its meaning? When is a `pthread_t` unique? When may it be reused?

   On my system - Ubuntu 18.04.5 LTS, `phthread_t` is defined as follows:

   ```
   /* Thread identifiers.  The structure of the attribute type is not
      exposed on purpose.  */
   typedef unsigned long int pthread_t;
   ```

   A thread ID is represented by the `pthread_t` data type. But the implementations can use different structures, which is system independent [1].

   > - Linux 3.2.0 uses an unsigned long integer for the pthread_t data type.
   > - Solaris 10 represents the pthread_t data type as an unsigned integer.
   > - FreeBSD 8.0 and Mac OS X 10.6.8 use a pointer to the pthread structure for the pthread_t data type.

   Unlike the process ID, which is unique in the system, the thread ID has significance only within the context of the process to which it belongs [1]. According to the `man pthread_self`:

   > Thread IDs are guaranteed to **be unique only within a process**. A thread ID may be reused after a terminated thread has been joined, or a detached thread has terminated.

2. What is a *thread-safe* function? Provide an example implementation of a simple C function that is **not** thread-safe and an improved version that **is** thread-safe.

   > A function is said to be thread-safe if and only if it will always produce correct results when called repeatedly from multiple concurrent threads. [2]

   A class of thread-unsafe functions are functions that do not protect shared variables. Here is an example:

```
volatile long cnt = 0;
void *thread(void *vargp){
    long i, niters = *((long *)vargp);
    for(i = 0; i < niters; i++)
        cnt++;
    return NULL;
}
```

the result of `cnt` is unpredictable.

We can use semaphores to implement mutual exclusion:

```
volatile long cnt = 0;
sem_t mutex;
void *thread(void *vargp){
    long i, niters = *((long *)vargp);
    for(i = 0; i < niters; i++){
        P(&mutex);
        cnt++;
        V(&mutex);
    }
    return NULL;
}
```

Note that we should initialize `mutex` to unity in the main routine:

```
sem_init(&mutex, 0, 1);
```

3. Given the following C structure:

```
struct foo
{
        unsigned int    f_count;
        pthread_mutex_t   f_lock;
};
```

    1. Implement a function `void increment(struct foo *)`, which increments the `f_count` field in a thread-safe manner.
    2. Implement the function `struct foo * foo_create(void)`.

```
void increment(struct foo f)
{
    pthread_mutex_lock(&f.f_lock);
    f.f_count++;
    pthread_mutex_unlock(&f.f_lock);
}

struct foo * foo_create(void)
{
    struct foo *f = malloc(sizeof(struct foo *));
    f->f_count = 0;
    int rc = pthread_mutex_init(&f->f_lock, NULL);
    assert(rc == 0); // check if init successes
```

1.2 Thread safety **2 / 2**

- ✓ **+ 0.5 pts** Function Definition
- ✓ **+ 0.75 pts** Thread-unsafe Example
- ✓ **+ 0.75 pts** Thread-safe Example

```
volatile long cnt = 0;
void *thread(void *vargp){
    long i, niters = *((long *)vargp);
    for(i = 0; i < niters; i++)
        cnt++;
    return NULL;
}
```

the result of `cnt` is unpredictable.

We can use semaphores to implement mutual exclusion:

```
volatile long cnt = 0;
sem_t mutex;
void *thread(void *vargp){
    long i, niters = *((long *)vargp);
    for(i = 0; i < niters; i++){
        P(&mutex);
        cnt++;
        V(&mutex);
    }
    return NULL;
}
```

Note that we should initialize `mutex` to unity in the main routine:

```
sem_init(&mutex, 0, 1);
```

3. Given the following C structure:

```
struct foo
{
        unsigned int    f_count;
        pthread_mutex_t   f_lock;
};
```

    1. Implement a function `void increment(struct foo *)`, which increments the `f_count` field in a thread-safe manner.
    2. Implement the function `struct foo * foo_create(void)`.

```
void increment(struct foo f)
{
    pthread_mutex_lock(&f.f_lock);
    f.f_count++;
    pthread_mutex_unlock(&f.f_lock);
}

struct foo * foo_create(void)
{
    struct foo *f = malloc(sizeof(struct foo *));
    f->f_count = 0;
    int rc = pthread_mutex_init(&f->f_lock, NULL);
    assert(rc == 0); // check if init successes
```

1.3 increment() 2 / 2

✓ + 1 pts Proof of work

✓ + 1 pts Showing how the f_count field is considered in a thread safe implementation

```
volatile long cnt = 0;
void *thread(void *vargp){
    long i, niters = *((long *)vargp);
    for(i = 0; i < niters; i++)
        cnt++;
    return NULL;
}
```

the result of `cnt` is unpredictable.

We can use semaphores to implement mutual exclusion:

```
volatile long cnt = 0;
sem_t mutex;
void *thread(void *vargp){
    long i, niters = *((long *)vargp);
    for(i = 0; i < niters; i++){
        P(&mutex);
        cnt++;
        V(&mutex);
    }
    return NULL;
}
```

Note that we should initialize `mutex` to unity in the main routine:

```
sem_init(&mutex, 0, 1);
```

3. Given the following C structure:

```
struct foo
{
        unsigned int    f_count;
        pthread_mutex_t   f_lock;
};
```

    1. Implement a function `void increment(struct foo *)`, which increments the `f_count` field in a thread-safe manner.
    2. Implement the function `struct foo * foo_create(void)`.

```
void increment(struct foo f)
{
    pthread_mutex_lock(&f.f_lock);
    f.f_count++;
    pthread_mutex_unlock(&f.f_lock);
}

struct foo * foo_create(void)
{
    struct foo *f = malloc(sizeof(struct foo *));
    f->f_count = 0;
    int rc = pthread_mutex_init(&f->f_lock, NULL);
    assert(rc == 0); // check if init successes
```

1.4 foo_create() 1 / 1

✓ **+ 1 pts** **Proof of work**

```
    return f;
}
```

4. Read the first three sections of Apple's `libdispatch` tutorial (up to, and including, "Defining work items: Functions and Blocks"). How do the C blocks described in this document compare to C++11 lambda functions?

   From the definition, they perform similar abilities.

   > a block as a snippet of code that can be passed around and executed like a function where it is required.

   > a lambda expression, or lambda, is a convenient way of defining an anonymous function object (a closure) right at the location where it is invoked or passed as an argument to a function. Typically lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous methods. [3]

   From the syntax, they look similar. Both of them

   - have arguments
   - can specify return types
     - block defines return types after the caret (^) [4]
     - lambda defines return types using trailing-return-type Optional
   - can be assigned to variables, and then invoked using the variable name

```
// define a block
void (^speak)() = ^(char *x){ printf("%s %s\n", greeting, x); };
// defien a lambda
auto f1 = [](int x, int y) { return x + y; };
```

Procedure

Serial execution

Download the C source file serial.c. What does it do? Using the provided Makefile, compile the source file into an executable program and run it. How long is required to execute each unit of work?

```
xy@xy-vm ~/D/E/lab2> ./serial
Counted to 100 in 1,928 ns: 19.280000 ns/iter
```

Using the `awk(1)` program, filter the output of your program to keep only the time per work unit, outputting the result into a file:

```
$ ./serial | awk '{ print $7 }' | tee -a initial-serial-times.dat
```

Use the shell to run this command 999 more times. Using `head(1)` and `ministat(1)`, report the average and standard deviation for the first 3, 5, 10, 100 and 1000 runs. Comment on the significance of the number of measurements when measuring physical phenomena.

1.5 C blocks vs lambda functions **2 / 2**

✓ **+ 1 pts** **Description: How a C block is behaved.**

✓ **+ 1 pts** **Differences: Syntax, Data type, Variables etc**

```
    return f;
}
```

4. Read the first three sections of Apple's `libdispatch` tutorial (up to, and including, "Defining work items: Functions and Blocks"). How do the C blocks described in this document compare to C++11 lambda functions?

   From the definition, they perform similar abilities.

   > a block as a snippet of code that can be passed around and executed like a function where it is required.

   > a lambda expression, or lambda, is a convenient way of defining an anonymous function object (a closure) right at the location where it is invoked or passed as an argument to a function. Typically lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous methods. [3]

   From the syntax, they look similar. Both of them

   - have arguments
   - can specify return types
     - block defines return types after the caret (^) [4]
     - lambda defines return types using trailing-return-type Optional
   - can be assigned to variables, and then invoked using the variable name

```
// define a block
void (^speak)() = ^(char *x){ printf("%s %s\n", greeting, x); };
// defien a lambda
auto f1 = [](int x, int y) { return x + y; };
```

Procedure

Serial execution

Download the C source file serial.c. What does it do? Using the provided Makefile, compile the source file into an executable program and run it. How long is required to execute each unit of work?

```
xy@xy-vm ~/D/E/lab2> ./serial
Counted to 100 in 1,928 ns: 19.280000 ns/iter
```

Using the `awk(1)` program, filter the output of your program to keep only the time per work unit, outputting the result into a file:

```
$ ./serial | awk '{ print $7 }' | tee -a initial-serial-times.dat
```

Use the shell to run this command 999 more times. Using `head(1)` and `ministat(1)`, report the average and standard deviation for the first 3, 5, 10, 100 and 1000 runs. Comment on the significance of the number of measurements when measuring physical phenomena.

2.1 Execution time (one invocation) 1 / 1

✓ + 0.5 pts What does the code do?

✓ + 0.5 pts Showing the output and execution time taken for each unit of work

```
    return f;
}
```

4. Read the first three sections of Apple's `libdispatch` tutorial (up to, and including, "Defining work items: Functions and Blocks"). How do the C blocks described in this document compare to C++11 lambda functions?

   From the definition, they perform similar abilities.

   > a block as a snippet of code that can be passed around and executed like a function where it is required.

   > a lambda expression, or lambda, is a convenient way of defining an anonymous function object (a closure) right at the location where it is invoked or passed as an argument to a function. Typically lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous methods. [3]

   From the syntax, they look similar. Both of them

   - have arguments
   - can specify return types
     - block defines return types after the caret (^) [4]
     - lambda defines return types using trailing-return-type Optional
   - can be assigned to variables, and then invoked using the variable name

```
// define a block
void (^speak)() = ^(char *x){ printf("%s %s\n", greeting, x); };
// defien a lambda
auto f1 = [](int x, int y) { return x + y; };
```

Procedure

Serial execution

Download the C source file serial.c. What does it do? Using the provided Makefile, compile the source file into an executable program and run it. How long is required to execute each unit of work?

```
xy@xy-vm ~/D/E/lab2> ./serial
Counted to 100 in 1,928 ns: 19.280000 ns/iter
```

Using the `awk(1)` program, filter the output of your program to keep only the time per work unit, outputting the result into a file:

```
$ ./serial | awk '{ print $7 }' | tee -a initial-serial-times.dat
```

Use the shell to run this command 999 more times. Using `head(1)` and `ministat(1)`, report the average and standard deviation for the first 3, 5, 10, 100 and 1000 runs. Comment on the significance of the number of measurements when measuring physical phenomena.

```
$ for i in $(seq 1000); do ./serial | awk '{ print $7 }' | tee -a
initial_serial_times.dat; done

$ for i in {3,5,10,100,1000}; do head -$i initial_serial_times.dat | ministat -A
-s; done
x <stdin>
    N           Min           Max        Median           Avg        Stddev
x   3         12.71         26.99         26.91     22.203333     8.2215651
x <stdin>
    N           Min           Max        Median           Avg        Stddev
x   5         12.71         26.99         23.49        22.578     5.8413629
x <stdin>
    N           Min           Max        Median           Avg        Stddev
x  10         12.71         26.99         18.36         18.88     5.7357844
x <stdin>
    N           Min           Max        Median           Avg        Stddev
x 100         11.02         29.08         17.64        17.878     3.8285717
x <stdin>
    N           Min           Max        Median           Avg        Stddev
x 1000         10.26        327.97           17.4      18.83886     12.390978
```

Normally, the larger measurment numbers, the better. Because there are inevitablely some random errors, increaing the number of measurments can offset these random errors.

However, here we observed a tradeoff.

With the number of measurements increase, we are highly likely to have bad data, e.g., we have Max data point 327.97 with N=1000, which will dramatically influence the Standard Deviation. The Average didn't change alot compared with N=100, because this time we don't have several really really bad points. If we're no lucky, e.g., with Max point 10000, then the Average will change a lot. Since the number of bad points are small, they don't influence Median very much. By observation, N=100 is a relatively good choice.

Explore the behaviour of this program by measuring the average per-unit-of-work execution time over varying values of `JOBS` and `WORK_PER_JOB`. For example, to delete (`clean`) the current version of the `serial` program and compile a new version with `JOBS` set to 100 and `WORK_PER_JOB` set to 100, you can run the following command:

```
$ make JOBS=100 WORK_PER_JOB=100 clean serial
```

How does the execution time vary with respect to `JOBS` and `WORK_PER_JOB`?

To simplify the command, I wrote the bash script:

```
#!/bin/bash
JOB=${1?Error: no JOBS given}
WORK=${2?Error: no WORK_PER_JOB given}
make JOBS=$JOB WORK_PER_JOB=$WORK clean serial
export PATH="~/Documents/ENGI9875/lab2:$PATH"
echo $PATH
for i in $(seq 100)
do
        serial | awk '{ $7 }' | tee -a initial_serial_times.dat
done

for i in {5,10,100}
```

2.2 Run-time statistics 2 / 2

✓ + 1 pts Analysis based on time, No. of executions & iterations

✓ + 1 pts Proof of work

```
$ for i in $(seq 1000); do ./serial | awk '{ print $7 }' | tee -a
initial_serial_times.dat; done

$ for i in {3,5,10,100,1000}; do head -$i initial_serial_times.dat | ministat -A
-s; done
x <stdin>
    N           Min           Max        Median           Avg        Stddev
x   3         12.71         26.99         26.91     22.203333     8.2215651
x <stdin>
    N           Min           Max        Median           Avg        Stddev
x   5         12.71         26.99         23.49        22.578     5.8413629
x <stdin>
    N           Min           Max        Median           Avg        Stddev
x  10         12.71         26.99         18.36         18.88     5.7357844
x <stdin>
    N           Min           Max        Median           Avg        Stddev
x 100         11.02         29.08         17.64        17.878     3.8285717
x <stdin>
    N           Min           Max        Median           Avg        Stddev
x 1000         10.26        327.97           17.4      18.83886     12.390978
```

Normally, the larger measurment numbers, the better. Because there are inevitablely some random errors, increaing the number of measurments can offset these random errors.

However, here we observed a tradeoff.

With the number of measurements increase, we are highly likely to have bad data, e.g., we have Max data point 327.97 with N=1000, which will dramatically influence the Standard Deviation. The Average didn't change alot compared with N=100, because this time we don't have several really really bad points. If we're no lucky, e.g., with Max point 10000, then the Average will change a lot. Since the number of bad points are small, they don't influence Median very much. By observation, N=100 is a relatively good choice.

Explore the behaviour of this program by measuring the average per-unit-of-work execution time over varying values of `JOBS` and `WORK_PER_JOB`. For example, to delete (`clean`) the current version of the `serial` program and compile a new version with `JOBS` set to 100 and `WORK_PER_JOB` set to 100, you can run the following command:

```
$ make JOBS=100 WORK_PER_JOB=100 clean serial
```

How does the execution time vary with respect to `JOBS` and `WORK_PER_JOB`?

To simplify the command, I wrote the bash script:

```
#!/bin/bash
JOB=${1?Error: no JOBS given}
WORK=${2?Error: no WORK_PER_JOB given}
make JOBS=$JOB WORK_PER_JOB=$WORK clean serial
export PATH="~/Documents/ENGI9875/lab2:$PATH"
echo $PATH
for i in $(seq 100)
do
        serial | awk '{ $7 }' | tee -a initial_serial_times.dat
done

for i in {5,10,100}
```

```
do
        head -$i initial_serial_times.dat | ministat -A -s
done
```

And save it as `lab.sh`

Since N=100 is a good choice, I only run 100 times and calculate their statistics attributes.

At first, I increased `WORK_PER_JOB` solely, set `JOBS=10 WORK_PER_JOB=100`:

```
$ ./lab.sh 10 100
```

Then change `JOBS` solely. Finally, change both. Here are results:

JOBS	WORK	N	Min	Max	Median	Average
10	10	100	13.6	53.9	19.6	20.6
10	100	100	3.1	29	5.3	5.6
10	1000	100	2.2	9.3	3.9	3.9
100	10	100	2.9	13.2	5.1	5.4
100	100	100	2.1	5.5	3.6	3.6
100	1000	100	2	4.7	3.6	3.5
1000	10	100	2	6.3	3.4	3.5
1000	100	100	2.2	5.2	3.8	3.9
1000	1000	100	2.1	4.6	3.2	3.2

It is very clear that with the increase of `JOBS` and `WORK_PER_JOB`, the execution time is decreasing. Here are possible reasons:

- there are some basic time consumption. The increased execution number will dilute this effect.
- system schedules better when the execution number increase.

POSIX threads

> Presenting complex data:
> Your computer should have R, a statistical analysis program, installed (you can start an R GUI session by running `R --gui=Tk`). R is not the only way to present statistical information, but it can draw boxplots, violin plots, 3D surface plots and contour plots. It's a useful tool to learn.

Copy `serial.c` to a new file called `pthreads.c`; add it to the `Makefile`. Modify this new file to execute the work in parallel using `JOBS` threads (note: do **not** use your thread-safe `increment` implementation from the prelab — that will come in later). As you did in the previous section, measure the average time to complete each unit of work for varying values of `JOBS` and `WORK_PER_JOB`. Present these results using appropriate graphical techniques. Also plot *throughput speedup* vs number of parallel threads.

Please refer to Appendix for the modified code. Also, in order to avoid typing the same commands again and again, I improved the bash script, as shown in the Appendix. To better plot, refer to Appendix to see how I plot using R. The followings are raw data and plot. As we can see, when `WORK_PER_JOB` is small, the improvement that increasing `JOBS` brings is significant.

2.3 Run-time varying with JOBS and WORK_PER_JOB **2 / 2**

✓ **+ 1 pts** **Analysis of Avg running time as per SD, no of measurements etc**

✓ **+ 1 pts** **Proof of work**

```
do
        head -$i initial_serial_times.dat | ministat -A -s
done
```

And save it as `lab.sh`

Since N=100 is a good choice, I only run 100 times and calculate their statistics attributes.

At first, I increased `WORK_PER_JOB` solely, set `JOBS=10 WORK_PER_JOB=100`:

```
$ ./lab.sh 10 100
```

Then change `JOBS` solely. Finally, change both. Here are results:

JOBS	WORK	N	Min	Max	Median	Average
10	10	100	13.6	53.9	19.6	20.6
10	100	100	3.1	29	5.3	5.6
10	1000	100	2.2	9.3	3.9	3.9
100	10	100	2.9	13.2	5.1	5.4
100	100	100	2.1	5.5	3.6	3.6
100	1000	100	2	4.7	3.6	3.5
1000	10	100	2	6.3	3.4	3.5
1000	100	100	2.2	5.2	3.8	3.9
1000	1000	100	2.1	4.6	3.2	3.2

It is very clear that with the increase of `JOBS` and `WORK_PER_JOB`, the execution time is decreasing. Here are possible reasons:

- there are some basic time consumption. The increased execution number will dilute this effect.
- system schedules better when the execution number increase.

POSIX threads

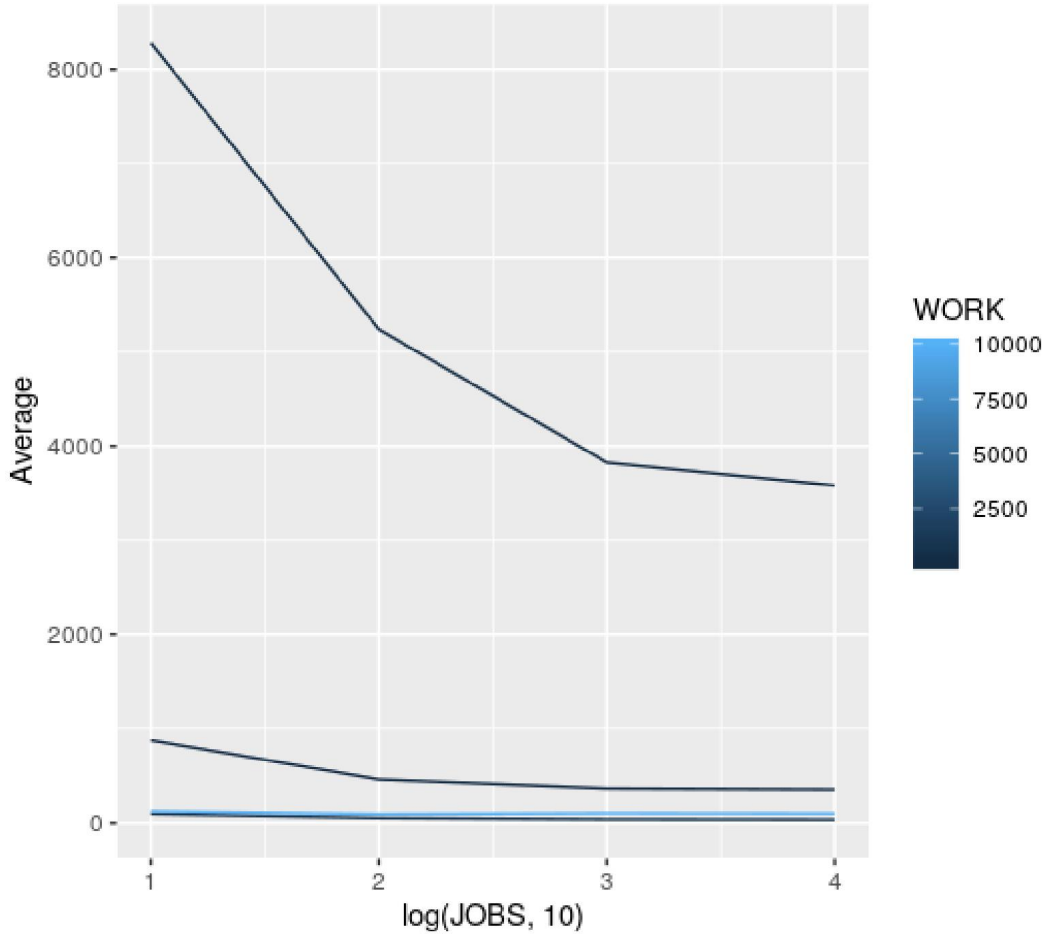
> Presenting complex data:
> Your computer should have R, a statistical analysis program, installed (you can start an R GUI session by running `R --gui=Tk`). R is not the only way to present statistical information, but it can draw boxplots, violin plots, 3D surface plots and contour plots. It's a useful tool to learn.

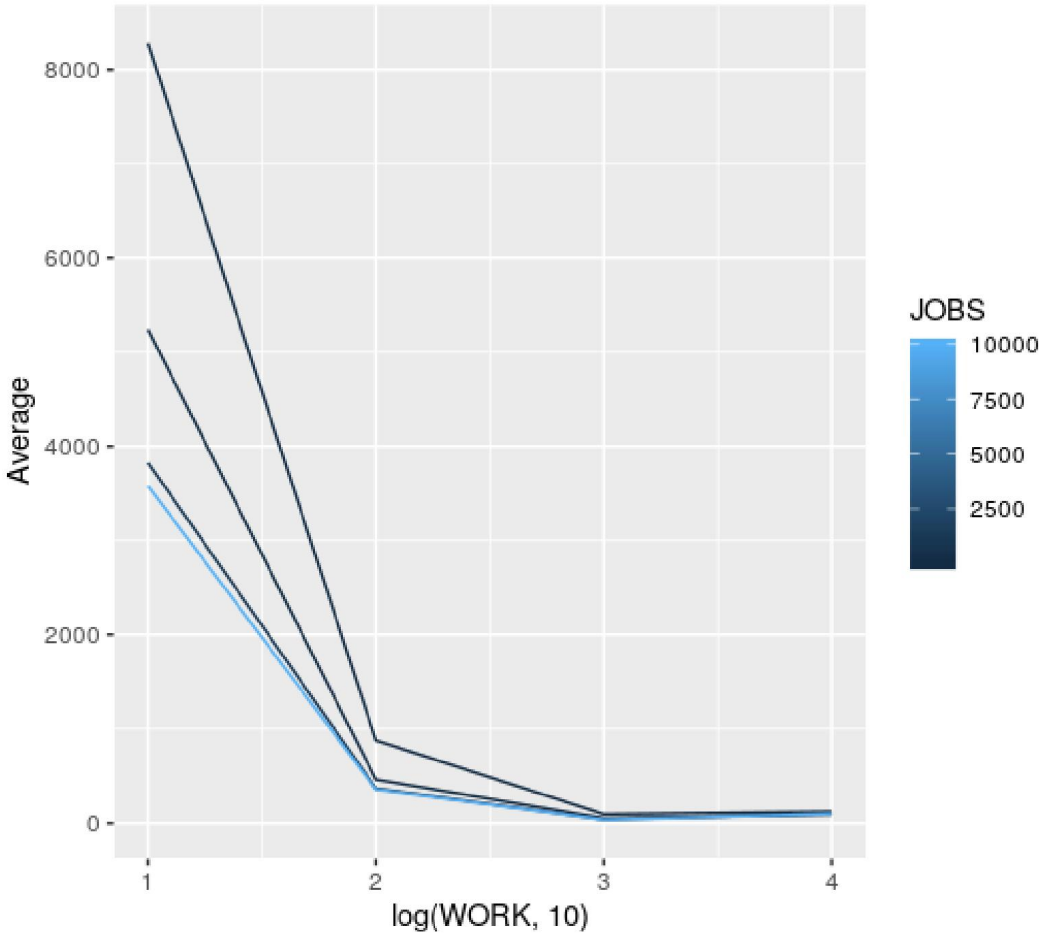
Copy `serial.c` to a new file called `pthreads.c`; add it to the `Makefile`. Modify this new file to execute the work in parallel using `JOBS` threads (note: do **not** use your thread-safe `increment` implementation from the prelab — that will come in later). As you did in the previous section, measure the average time to complete each unit of work for varying values of `JOBS` and `WORK_PER_JOB`. Present these results using appropriate graphical techniques. Also plot *throughput speedup* vs number of parallel threads.

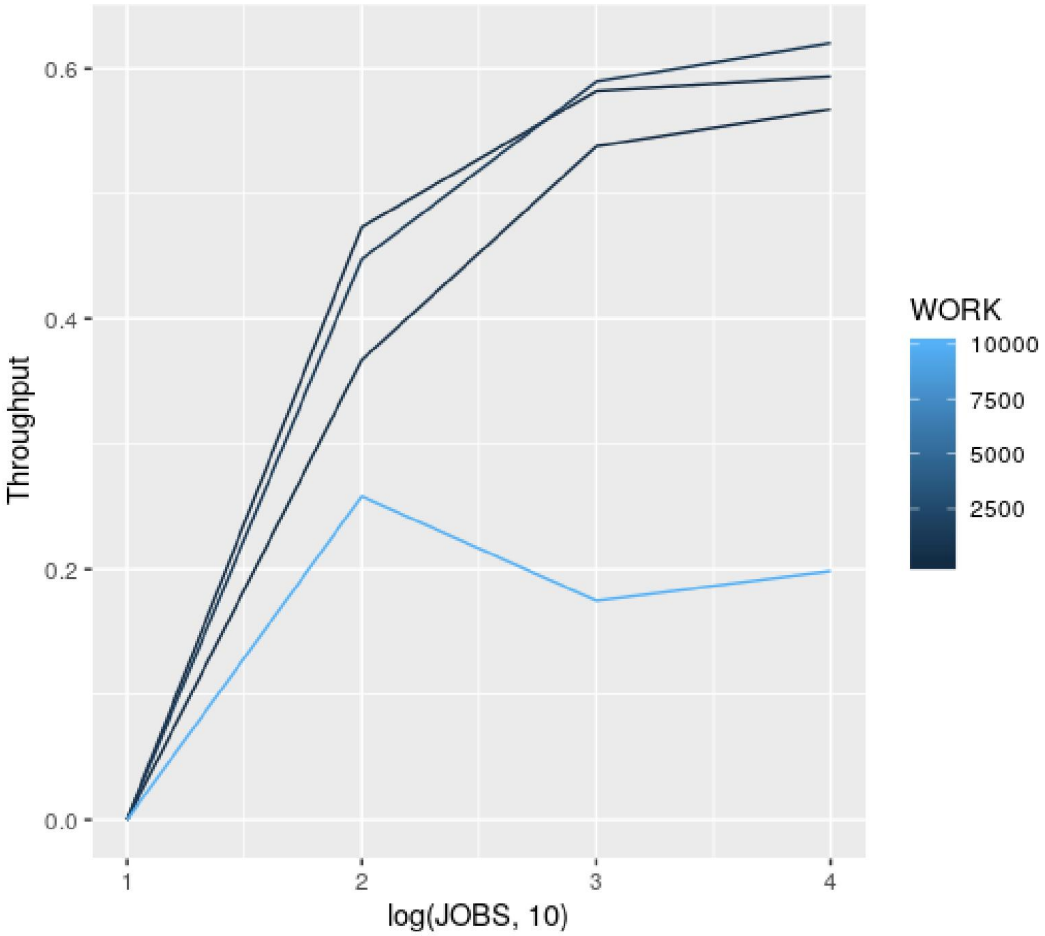
Please refer to Appendix for the modified code. Also, in order to avoid typing the same commands again and again, I improved the bash script, as shown in the Appendix. To better plot, refer to Appendix to see how I plot using R. The followings are raw data and plot. As we can see, when `WORK_PER_JOB` is small, the improvement that increasing `JOBS` brings is significant.

However, when `WORK_PER_JOB` is big, e.g., 10000, increasing `JOBS` will also increase the average time. The reason might partly be the synchronizatoin error.

JOBS	WORK	N	Min	Max	Median	Average
10	10	100	3490.2	13972.6	7915.5	8277.7
10	100	100	441.6	1393.4	817.4	876.2
10	1000	100	44.3	147.4	94.4	94.5
10	10000	100	19.9	204.3	125.8	122.4
100	10	100	3358.1	7767.3	5281.4	5238.6
100	100	100	303.2	642	452.9	461.5
100	1000	100	33	76.3	51.1	52.2
100	10000	100	8.8	169.8	99.5	90.8
1000	10	100	3257.2	4639.4	3811.7	3825.6
1000	100	100	327.1	448.1	362.9	366.3
1000	1000	100	33	52.2	38.7	38.8
1000	10000	100	15.3	142.1	111.7	101
10000	10	100	3362.1	4280.9	3528.7	3581.1
10000	100	100	340.8	427.5	353.4	356.3
10000	1000	100	33.2	44	35.5	35.9
10000	10000	100	28.6	123.4	101.7	98.1




8000
6000
4000
2000
0
Average
1
2
3
4
log(WORK, 10)
JOBS
10000
7500
5000
2500


0.6
0.4
0.2
0.0
Throughput
1
2
3
4
log(JOBS, 10)
WORK
10000
7500
5000
2500

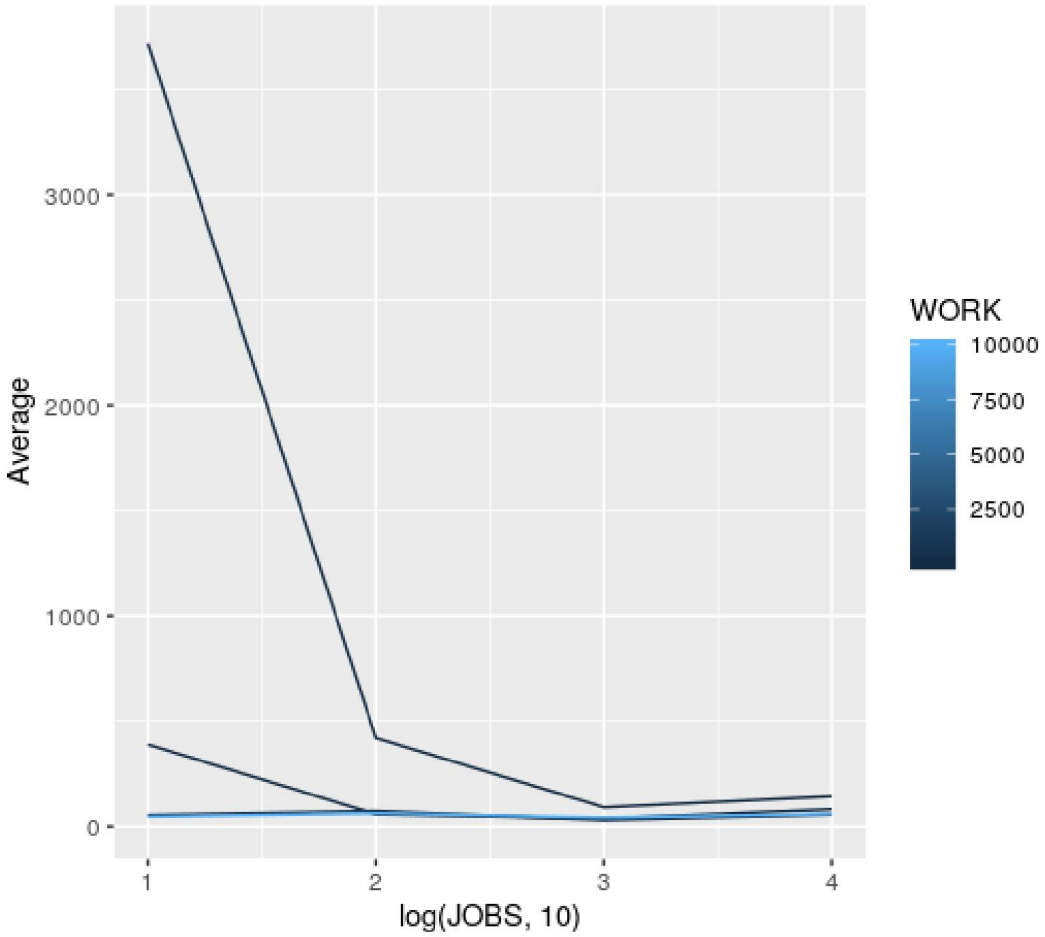
3 POSIX threads 4 / 4

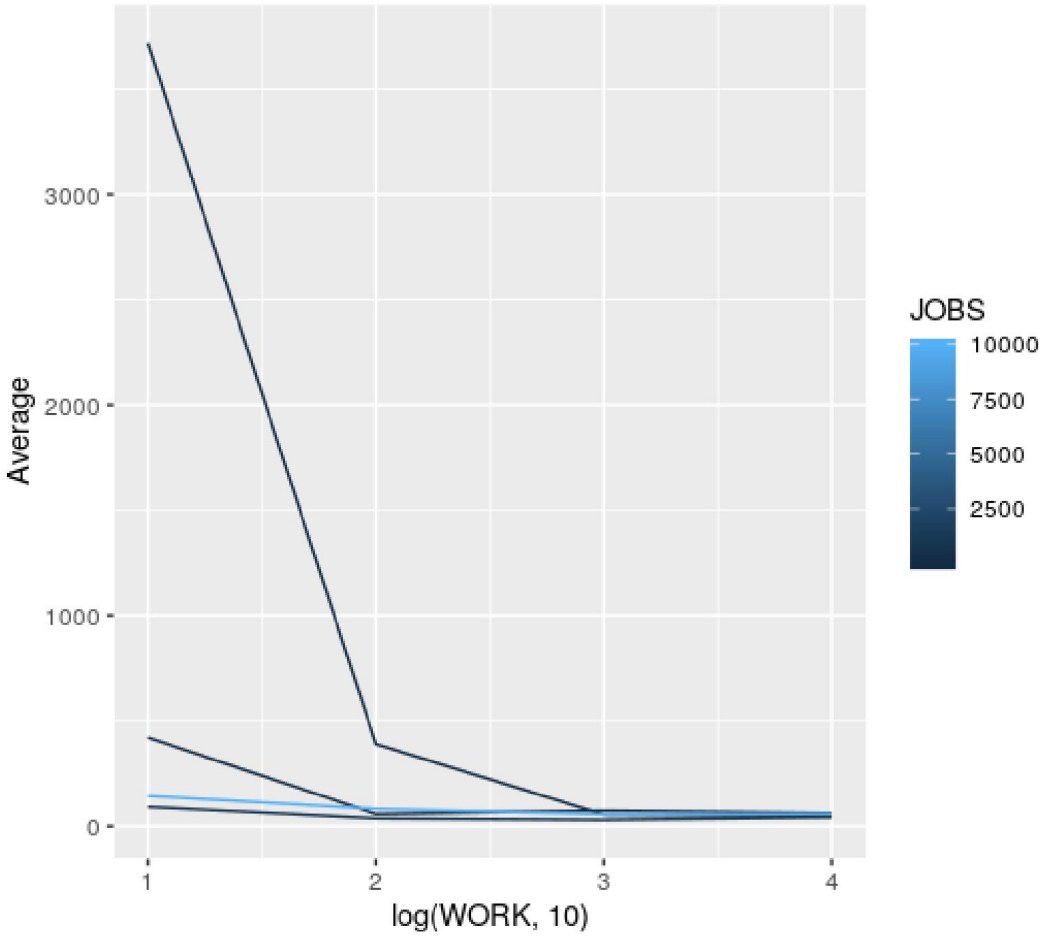
- ✓ + 1 pts Updated Code files & snippet for test run
- ✓ + 2 pts Proof of work: POSIX threads execution for given JOBS & WORK_PER_JOB
- ✓ + 1 pts throughput plot

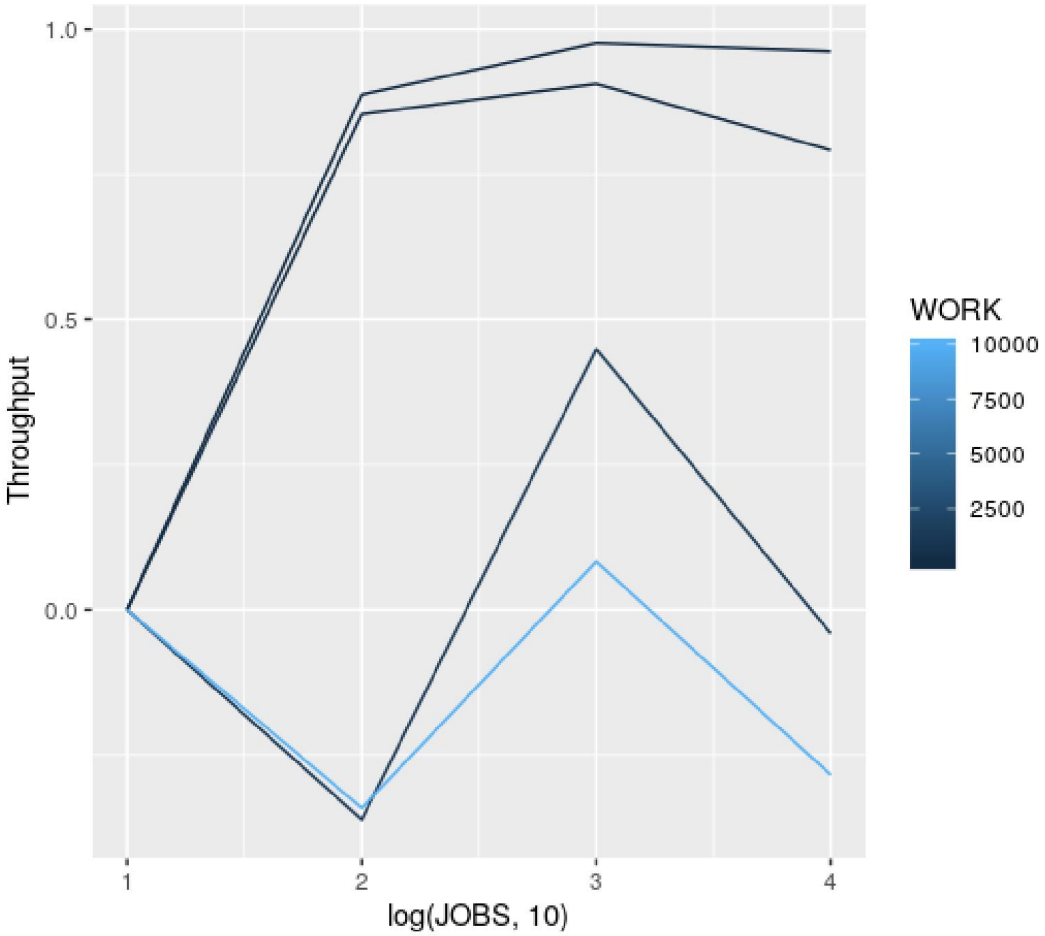
libdispatch

Copy `serial.c` to a new file called `libdispatch.c`; add it to the `Makefile` as well. Modify this new file to execute the work using `libdispatch` with `JOBS` asynchronous jobs. As you did in the previous section, measure the average time to complete each unit of work for varying values of `JOBS` and `WORK_PER_JOB`. Present these results using appropriate graphical techniques.

The modified code is shown in Appendix. Here are results:




Average
3000
2000
1000
0
1
2
3
4
log(WORK, 10)
JOBS
10000
7500
5000
2500


Throughput
1.0
0.5
0.0
1
2
3
4
log(JOBS, 10)
WORK
10000
7500
5000
2500

4 libdispatch 4 / 4

- ✓ + 1 pts Updated Code files & snippet for test run
- ✓ + 2 pts Proof of work: threads execution for given JOBS & WORK_PER_JOB
- ✓ + 1 pts throughput plot

Race conditions

Modify all three of your programs to print the actual value of `counter` rather than `JOBS * WORK_PER_JOB`. Explore how this value changes for varying values of `JOBS` and `WORK_PER_JOB` for both POSIX threads and `libdispatch`. Finally, modify your parallel programs to ensure that the correct value is counted. Plot throughput speed vs number of parallel threads in the POSIX case. Discuss your observations.

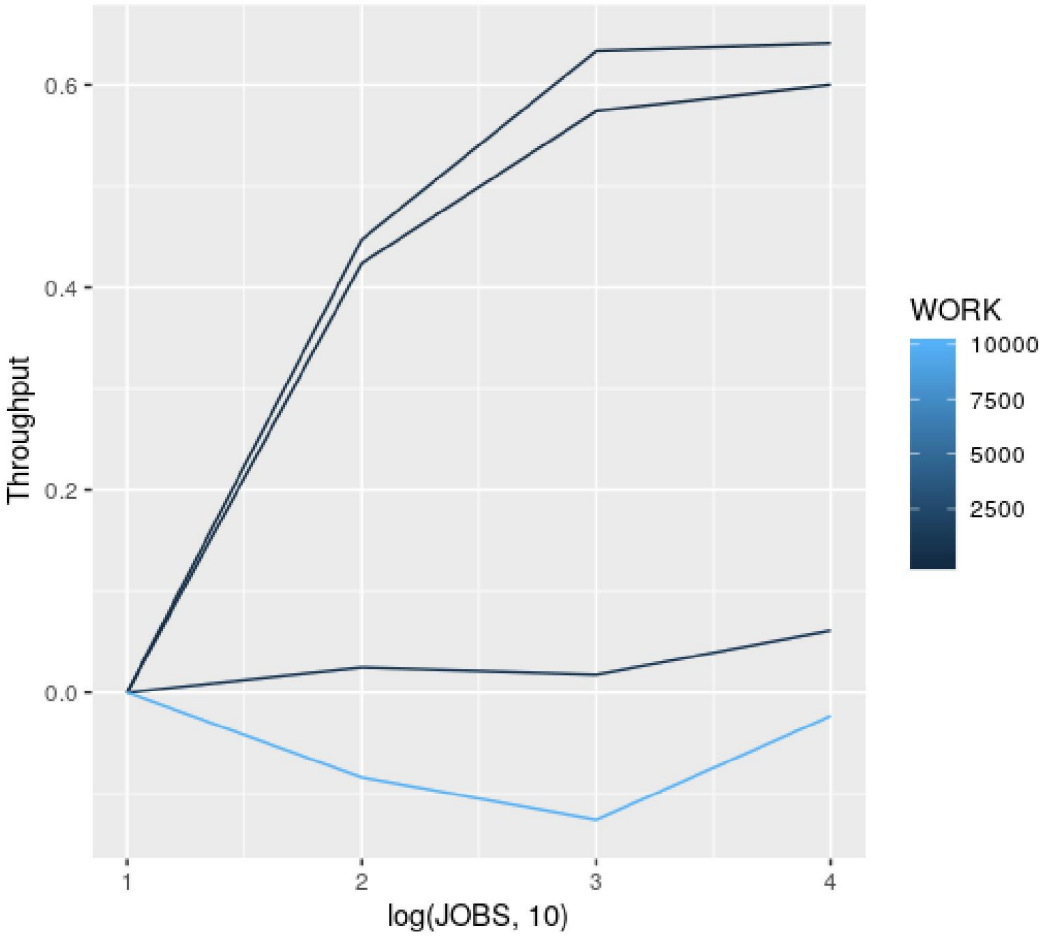
Running multiple times with different `JOBS` and `WORK_PER_JOB` with each program. the result are as follows:

JOBS	WORK	serial Counter	pthreads Counter	libdispatch Counter
10	10	100	100	100
10	100	1,000	1,000	1,000
10	1000	10,000	9,211	10,000
10	10000	100,000	28,966	58,944
100	10	1,000	1,000	1,000
100	100	10,000	10,000	7,664
100	1000	100,000	99,244	44,351
100	10000	1,000,000	340,230	1,000,000
1000	10	10,000	10,000	9,721
1000	100	100,000	98,752	65,102
1000	1000	1,000,000	997,912	1,000,000
1000	10000	10,000,000	2,725,413	5,164,541
10000	10	100,000	99,966	93,451
10000	100	1,000,000	997,696	693,325
10000	1000	10,000,000	9,825,318	3,351,309
10000	10000	100,000,000	24,247,497	50,299,645

Apparently, when `JOB*WORK_PER_JOB` becomes big, the POSIX thread counter and the libdispatch counter wouldn't perform as we expected. In addition, we have different values each time. This is called sychronization error [5]. Because the counter loop actually have several instructions.

> each concurrent execution defines some total ordering (or interleaving) of the instructions in the different threads. Unfortunately, some of these orderings will produce correct results, but others will not.

Then I modified the `pthreads.c` to make it thread-safe, as shown in Appendix. Then thee counters are correct. Here is the plot:



As we observed before, it works well when `WORK_PER_JOB` is small, the improvement that increasing `JOBS` brings is significant. However, when `WORK_PER_JOB` is big, increasing `JOBS` even has bad effects. But that is acceptable, since the absolute value of `Throughput` is small. It might because some random errors.

Appendix

POSIX threads code

pthreads.c: thread unsafe

To make it thread safe, uncomment comments.

```
#include <err.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <string.h>

void *increment(void *vargp);
// volatile pthread_mutex_t plock;
int main(void)
{
    int counter = 0;
    struct timespec begin, end;
    pthread_t **tids = malloc(sizeof(pthread_t *)*JOBS);
    // int rc1 = pthread_mutex_init(&plock, NULL);
    // if(rc1 != 0)
```

5 Race conditions 3.5 / 4

+ **0.5 pts** Updated Code files & snippet for test run

✓ **+ 1 pts** **Proof of work: threads execution for varying JOBS & WORK_PER_JOB (Highlighted changes)**

✓ **+ 1.5 pts** **Proof of work: correct value are counted as per modifications**

✓ **+ 0.5 pts** **throughput plot**

✓ **+ 0.5 pts** **Observations Discussed**