Lab3 Report

Yiwen Xu (48377645)

1. Document API for Library

1.1 Class Definition

1.1.1 Factory

A.Factory

- Attributes:

Name	Type
producing	atomic<bool>
machines	vector<Machine *>
threads	vector<thread>
commands	vector<Command *>

- Methods:

Name	Input	Output	Description
SetMachine	Shop *	N/A	Create machines instances, and set them to `stop` state.
Produce	N/A	N/A	Create threads of machines.
Stop	N/A	N/A	Stop machines action. Clean up threads.
ShutDown	N/A	N/A	Make sure all threads are stopped. Delete all instances of machines.

B. Machine

- Attributes:

Name	Type
workable	atomic<bool>
fsm	FSM
produceA	ProduceA
produceB	ProduceB
stop	Stop
_shop	Shop

- Methods:

Name	Input	Output	Description
Work	N/A	N/A	If the machine is in a workable state, let it execute the operation in the current state.
ShutDown	N/A	N/A	Turn the machine into an unworkable state.
Update	N/A	N/A	Let machine execute the operation in the current state.

C. FSM

- Attributes:

Name	Type
currentState	State *
previousState	State *

- Methods:

Name	Input	Output	Description
Update	Machine *	N/A	Let machine execute the operation in the current state.
ChangeState	Machine *, State *	N/A	Change the state of machine from current state to an input state.

D. ProduceA/ProduceB

- Methods:

Name	Input	Output	Description
Execute	Machine *	N/A	If the store storage is full, change the current status of the machine to `stop`. Otherwise, produce a new product with random value and store it in the store's warehouse (a queue).

E. Stop

- Methods:

Name	Input	Output	Description
Execute	Machine *	N/A	If the model still running, check whether the warehouse of store is empty. If it is, change the machine current state to `produceA` or `produceB`.

1.1.2 Customers

A. People

- Attributes:

Name	Type
_shop	Shop *
shopping	atomic<bool>

- Methods:

Name	Input	Output	Description
Shopping	N/A	N/A	If the model still running, check whether the warehouse of store is not empty. If it is, take a product A or B out of the shop.
Leave	N/A	N/A	Turn the person into a not-shopping state.

B. Customer

- Attributes:

Name	
shopping	atomic<bool>
customers	vector<People *>
threads	vector<thread>
commands	vector<Command *>
customerNum	int

- Methods:

Name	Input	Output	Description
CustomerReady	Shop *	N/A	Create people instances.
Shopping	N/A	N/A	Create threads of people.
Stop	N/A	N/A	Stop people action. Clean up threads.
LeaveAway	N/A	N/A	Make sure all threads are stopped. Delete all instances of people.

1.1.3 Shop

A. Shop

- Attributes:

Name	Type
isOpen	atomic<bool>
mutexLock	mutex
targetIncome	float
income	float
capacity	int
soldcounter	atomic_int
counterA	atomic_int
counterB	atomic_int
warehouseA	queue<Product>
warehouseB	queue<Product>

- Methods:

Name	Input	Output	Description
Stock	Product	N/A	Store a product in its warehouse.
Sold	string	N/A	Sold a product by popping out a specific product from its warehouse, and add the its value to total income of the shop.
SetIncome	N/A	N/A	Reset total income. (Used to initialize model)
ResetWarehouse	N/A	N/A	Reset all warehouse. (Used to initialize model)
CheckCapacity	N/A	bool	Return whether the warehouse is full by specific product name.
CheckInStock	N/A	bool	Return whether the warehouse is empty by specific product name.
GetSoldCount	N/A	int	Return the number of sold products.
GetIncome	N/A	float	Return the total income.

B. Product

- Attributes:

Name	Type
_id	int
_name	string
_price	float

1.1.4 API Definition

`mian.cpp`

Method	Input	Output	Description
Ready	N/A	N/A	Create both machines and people instances, and set them to `stop` state. Get ready for running model.
Set	float	N/A	Manually set the target income of the shop.
Start	N/A	N/A	Create threads of machines and people. Model start running.
Stop	N/A	N/A	Stop both machines and people actions. Clean up threads.
Kill	N/A	N/A	Make sure all threads are stopped. Delete all instances of machines and people.
Exit	N/A	N/A	Shows running time for each running time of model.

1.2 Code Base Description

1.2.1 Design Concepts

The code base of this lab is an implementation of the producer-consumer model.

The producer-consumer model is the classic multi-threaded concurrent collaboration model. The producer-consumer pattern is to solve the strong coupling problem of producer and consumer by a container. If the shared data area is full, the producer suspends production and waits for notification from the consumer before starting again.

Consumers are used to consume data, one by one, from the shared data area (warehouses of shop). If the shared data area is empty, the consumer pauses to fetch data and waits for notification from the producer before starting again. Producers and consumers cannot interact directly, but use a warehouse for the data shared between them.

In this implementation, we have:

- Three roles: producer, consumer, and warehouse
- Two relationships:
  - producer and producer are mutually exclusive
  - consumer and consumer are mutually exclusive
  - producer and consumer are synchronous and mutually exclusive
- One place of transaction: the warehouse

To clearly show how multiple threads work alternately, the program outputs a state change log when the state of a thread (producer/machine) changes. I assume one producer, one client to show the performance of a single-thread, and multiple producers, one client as a multi-threaded design. The program will complete one round of testing and join each thread (shut down machine) when the shop's income reaches the target value, which is set to 2000.

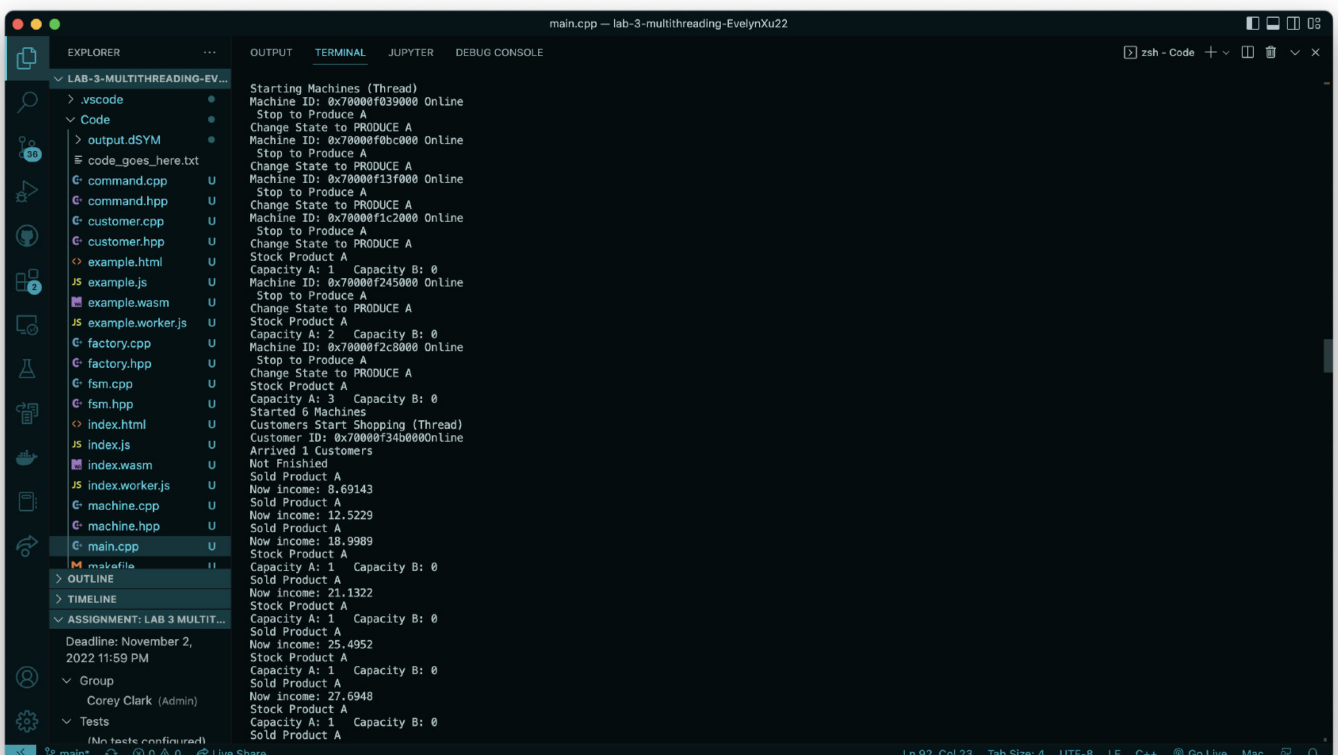


Figure 1. Running Log Reflecting Implemented Functionality (Multi-threaded)

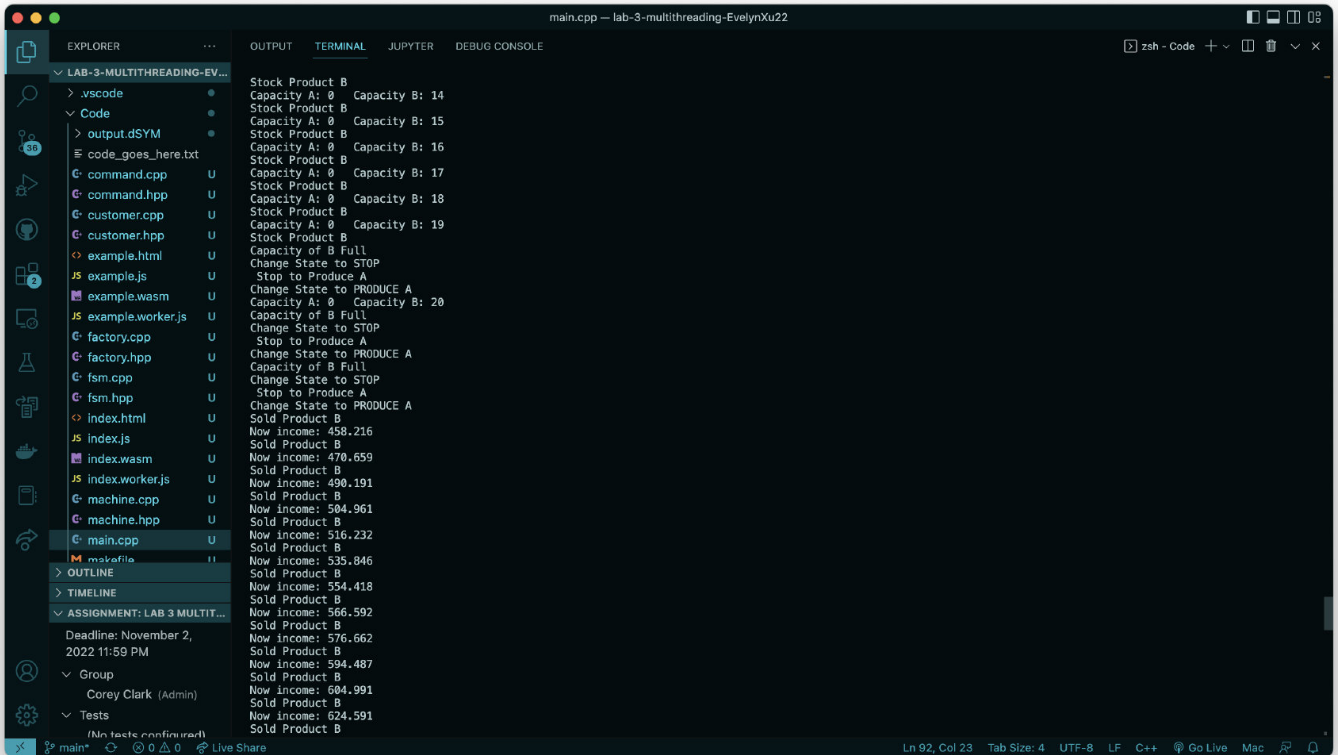


Figure 2. Running Log Reflecting Implemented Functionality (Multi-threaded)

As shown in the figures, we can see that multiple machines are created and their states change during this process. If the warehouse of shop is full, they will wait for customer, and start production if any warehouse is not full.

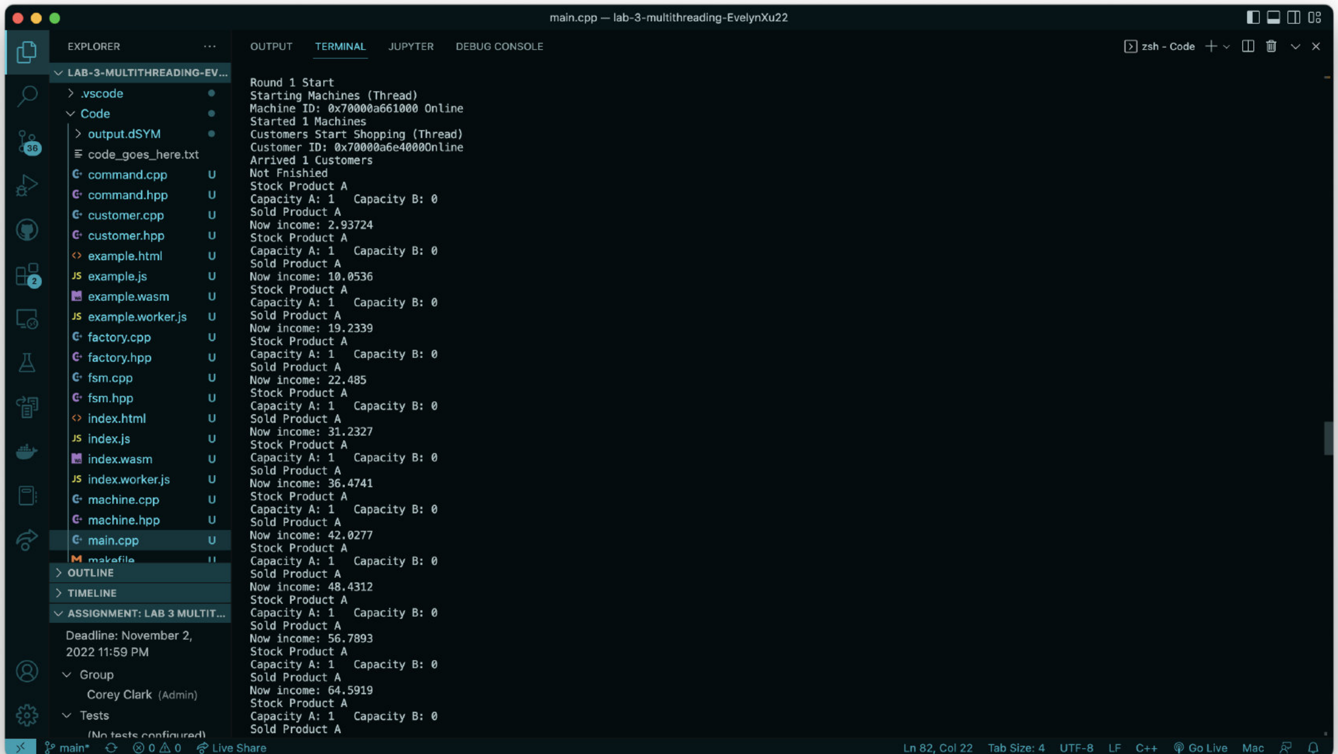
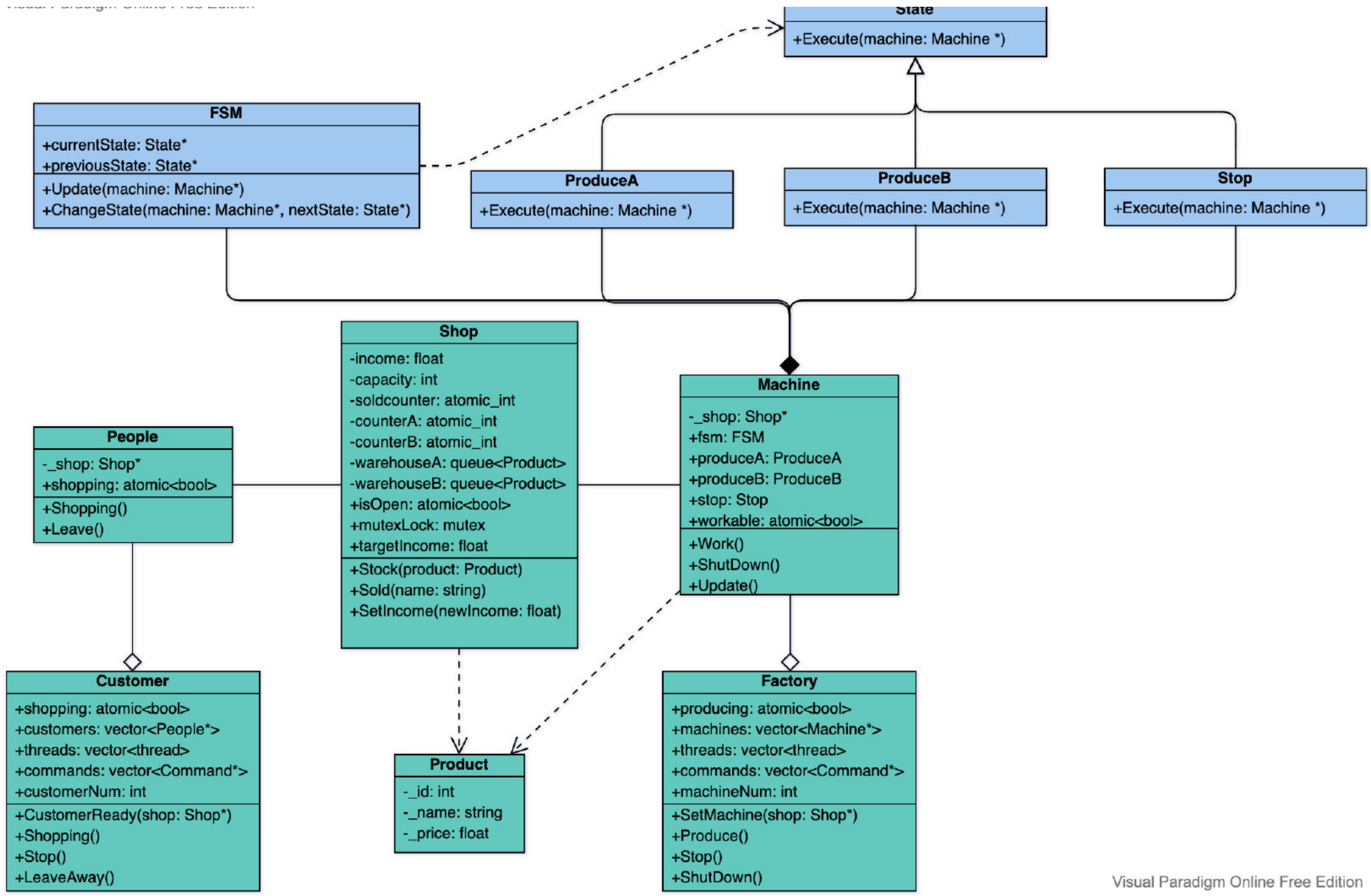


Figure 3. Running Log Reflecting Implemented Functionality (Single-threaded)

However, since I designed the production functions to require some runtime to complete, in order to simulate a real production situation, customer will keep waiting for the machine to produce a new product if there is only one thread (machine). We can see this process clearly through the running log.

1.3 Class Diagram



In this implementation of the producer-consumer model, we can create multiple people and machines threads which will operating at the same time. Therefore, `Customer` class and `Factory` class are used to manage those threads.

In this implementation, two kinds of product can be produced, so I applied State design patterns in this situation. As the behavior of an object depends on its state, this pattern allows the object to change its behavior at runtime depending on the state.

For `Customer`, `Factory` and `Shop` class, I used the singleton pattern to create their instances, to ensure that a class has only one instance and to provide a global access point to it. The application of the singleton pattern ensures that there is only one instance in memory, reducing memory consumption.

2. Compare and Contrast Execution Time

2.1 The Native Comparison Application Execution Time

The native comparison application is written in C/C++ code. In this case, I set the code to loop 40 times, automatically outputting a running log to reflect the functionality achieved by the program and recording the execution time.

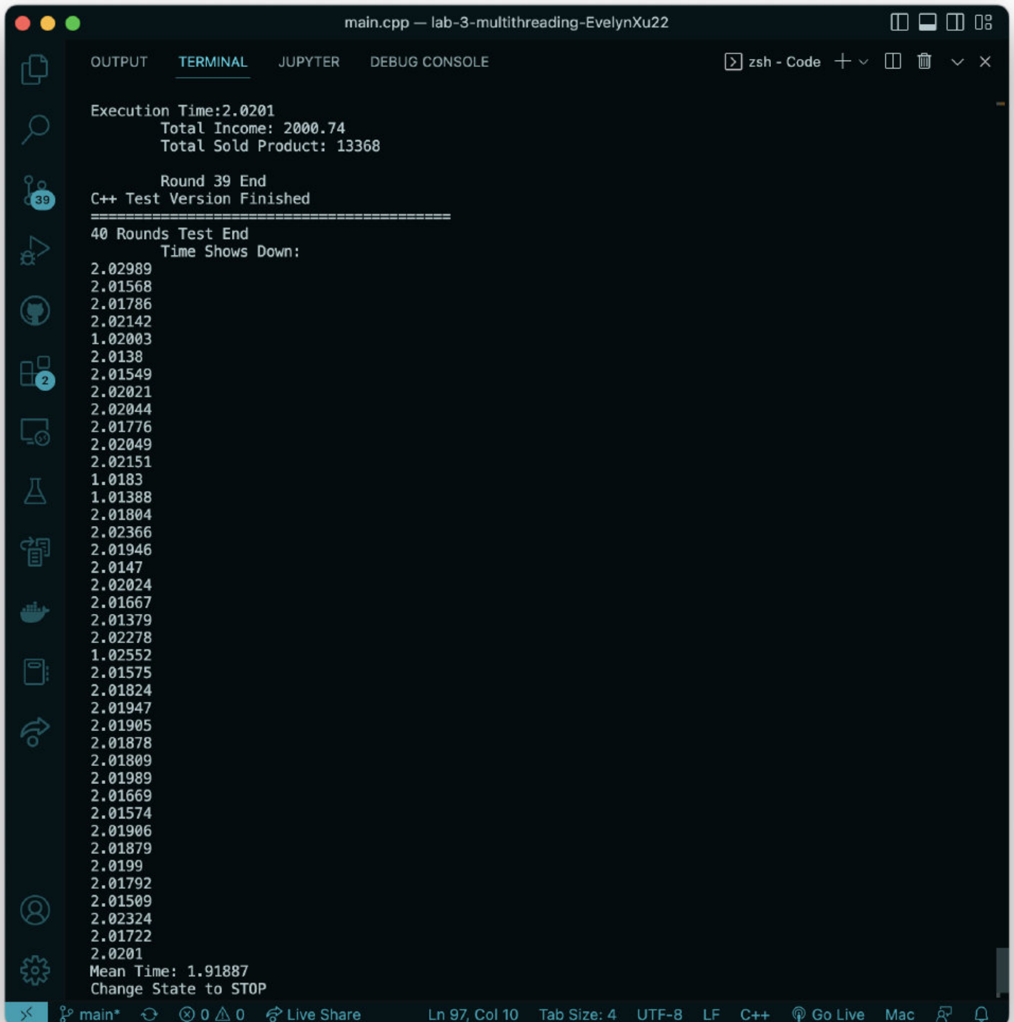


Figure 4. The Execution Time of The Native Comparison Application (Multi-threaded)

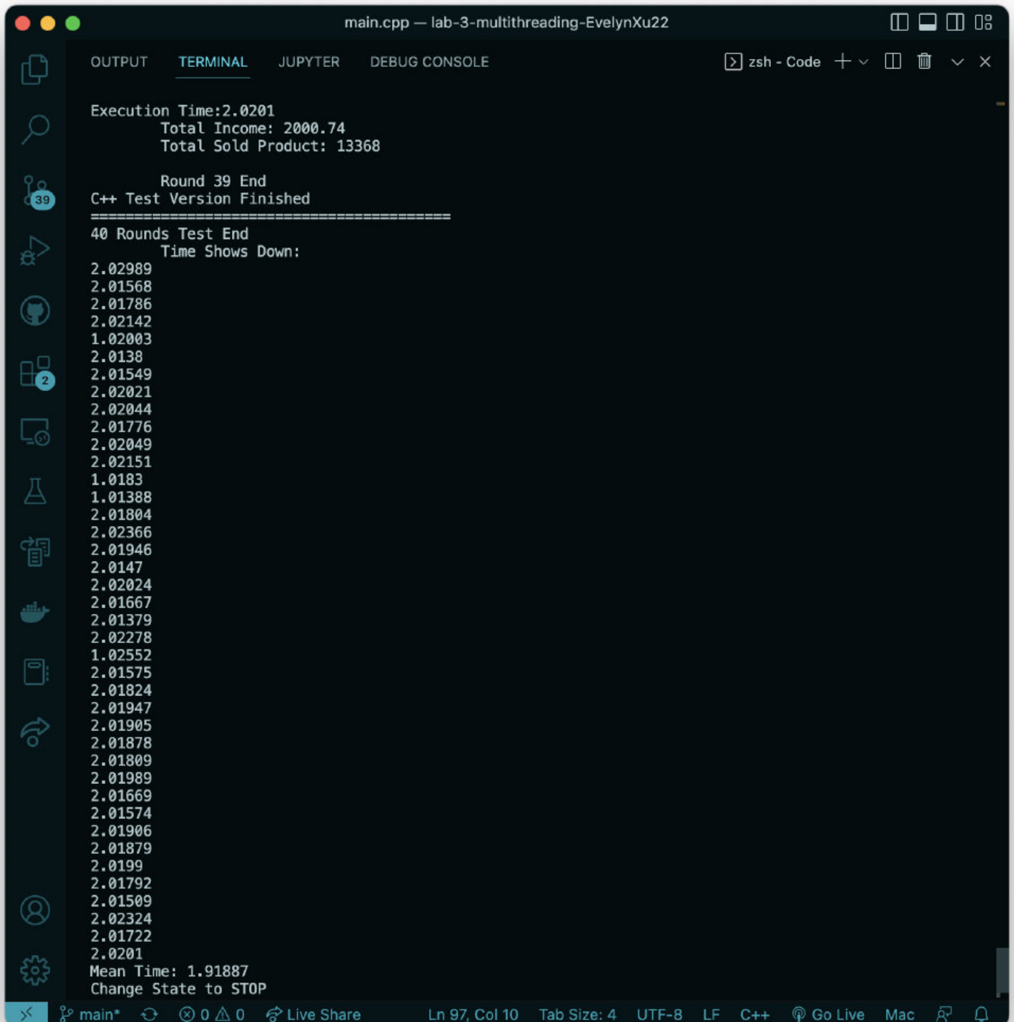


Figure 5. The Execution Time of The Native Comparison Application (Single-threaded)

2.2 The Demo Application Execution Time

The demo application is obtained by converting the C++ code base into JavaScript with Emscripten. In this case, I also set the code to loop 40 times, automatically outputting a run log to reflect the functionality achieved by the program and recording the execution time.

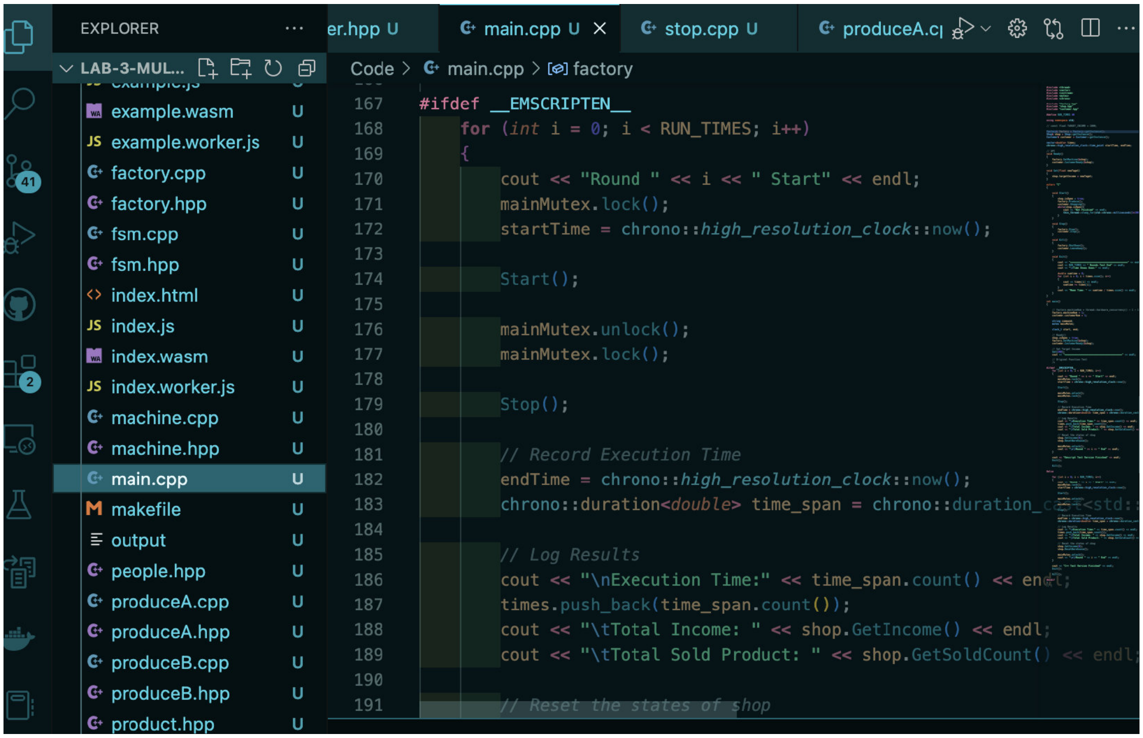


Figure 6. The Emscripten Version Test Application

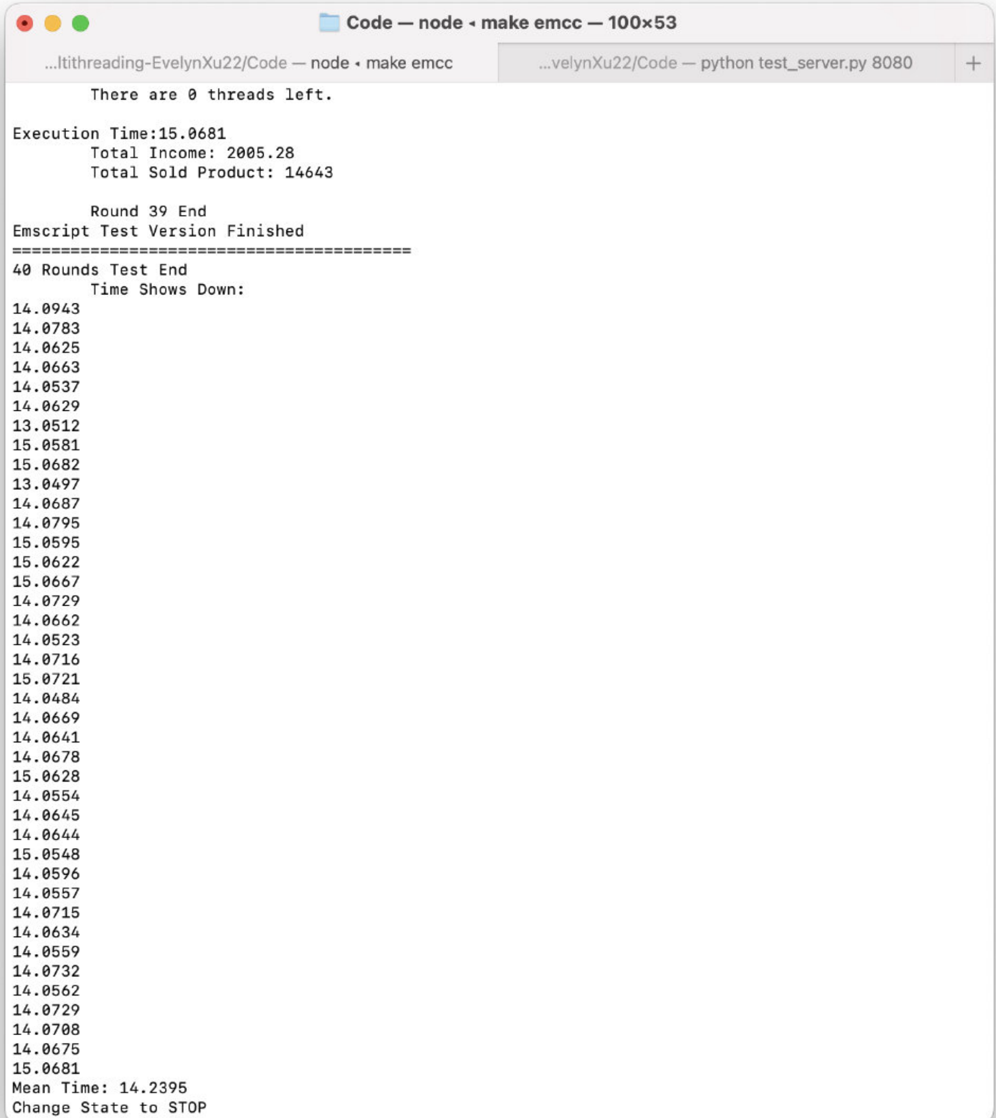


Figure 7. The Execution Time of The Demo Application (Multi-threaded)

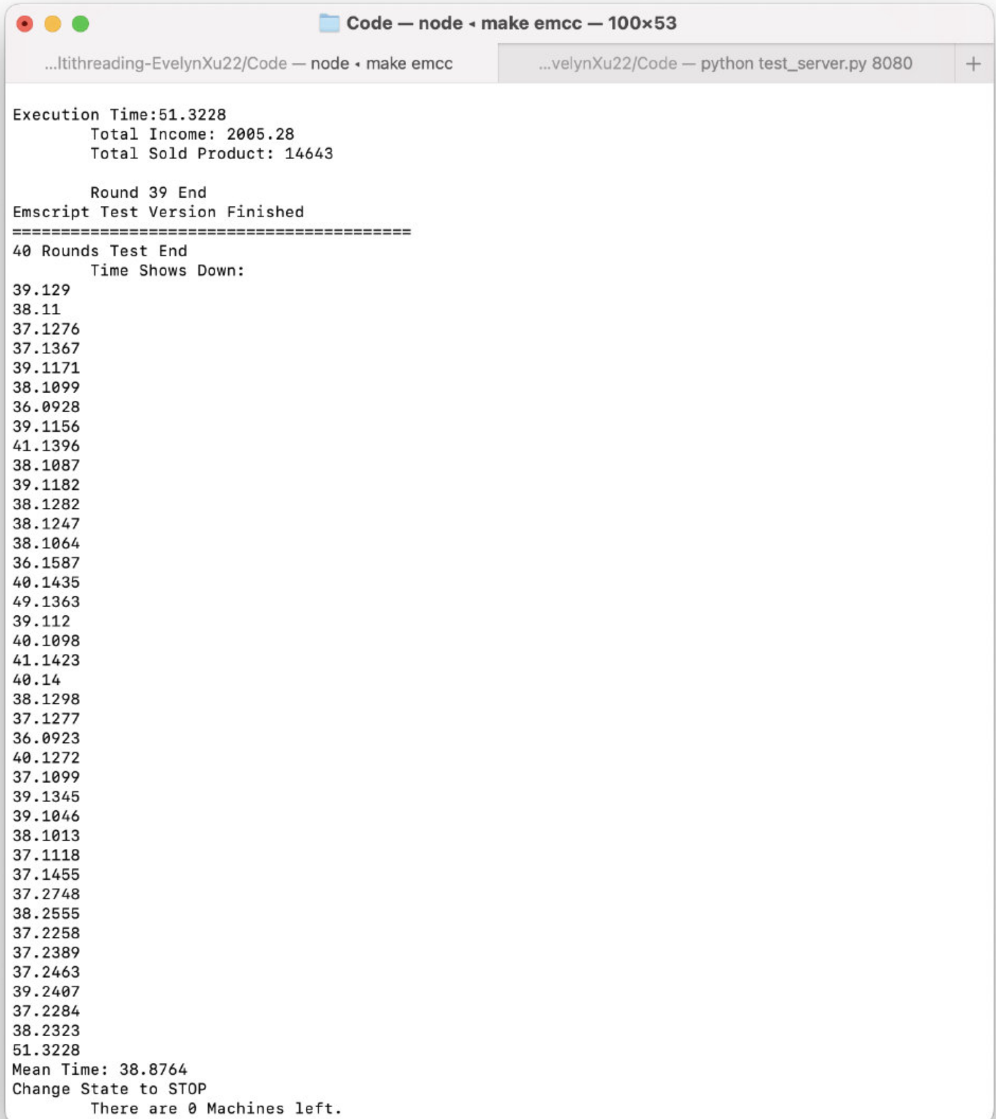


Figure 8. The Execution Time of The Demo Application (Single-threaded)

2.3 The Optimized Demo Application Execution Time

We can optimize by specifying the optimization flags, which are:-O0, -O1, -O2, -Os, -Oz, -O3. In this case, I choosed -O2 level optimization for compiled. I set the code to loop 40 times and recording the execution time.

```
emcc -std=c++14 -O2 -pthread -s PROXY_TO_PTHREAD -s ALLOW_MEMORY_GROWTH=1 -s
NO_DISABLE_EXCEPTION_CATCHING -s LLD_REPORT_UNDEFINED -s ERROR_ON_UNDEFINED_SYMBOLS=1
./*.cpp -o example.js
```

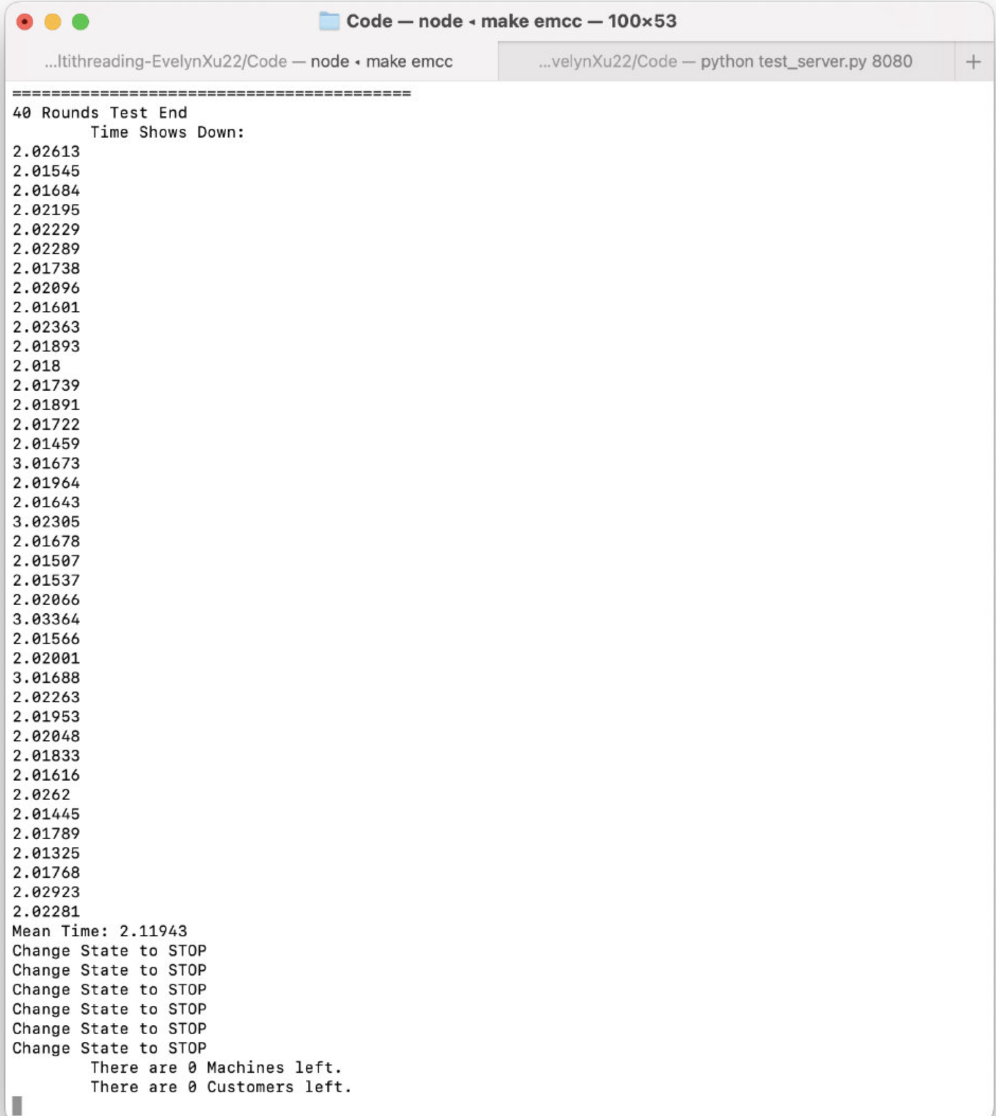


Figure 9. The Execution Time of The Optimized Demo Application (Multi-threaded)

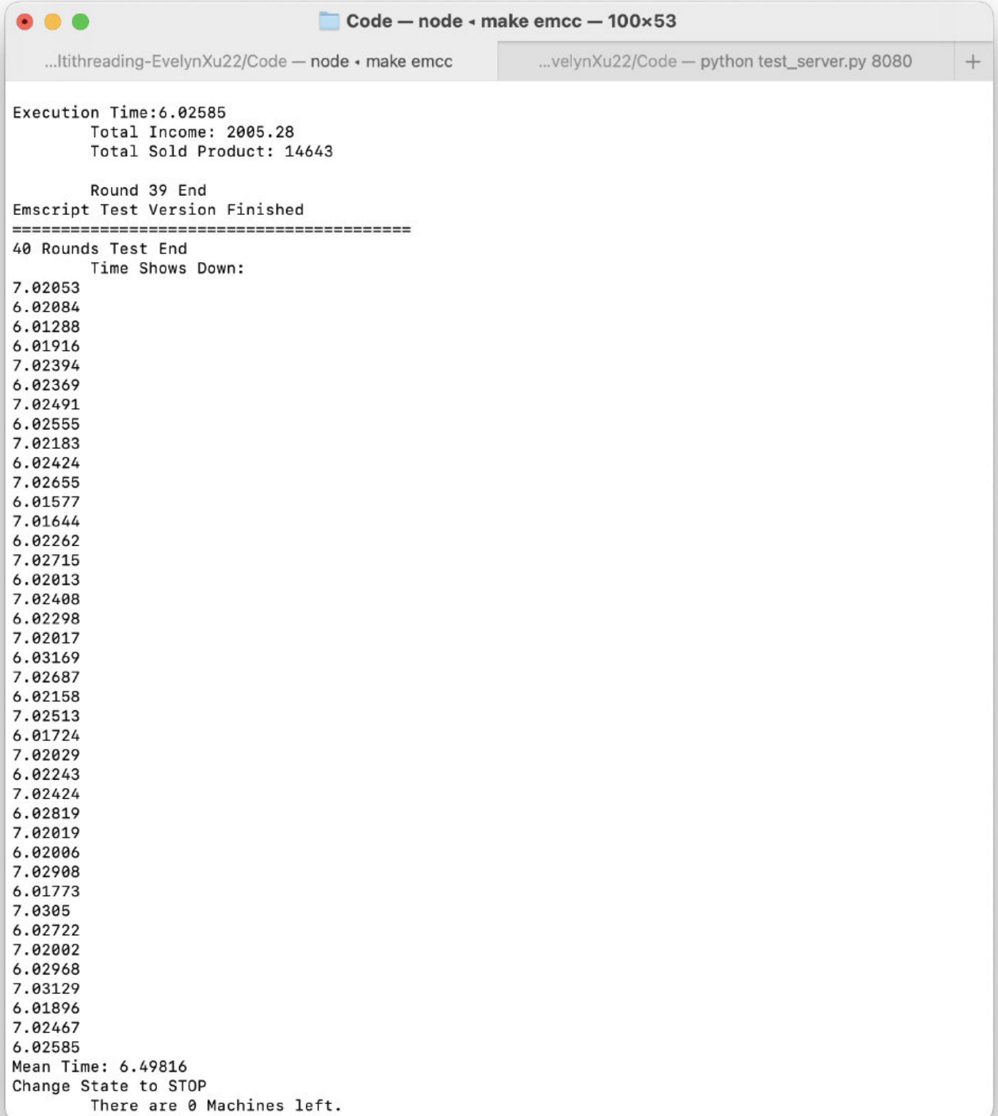


Figure 10. The Execution Time of The Optimized Demo Application (Single-threaded)

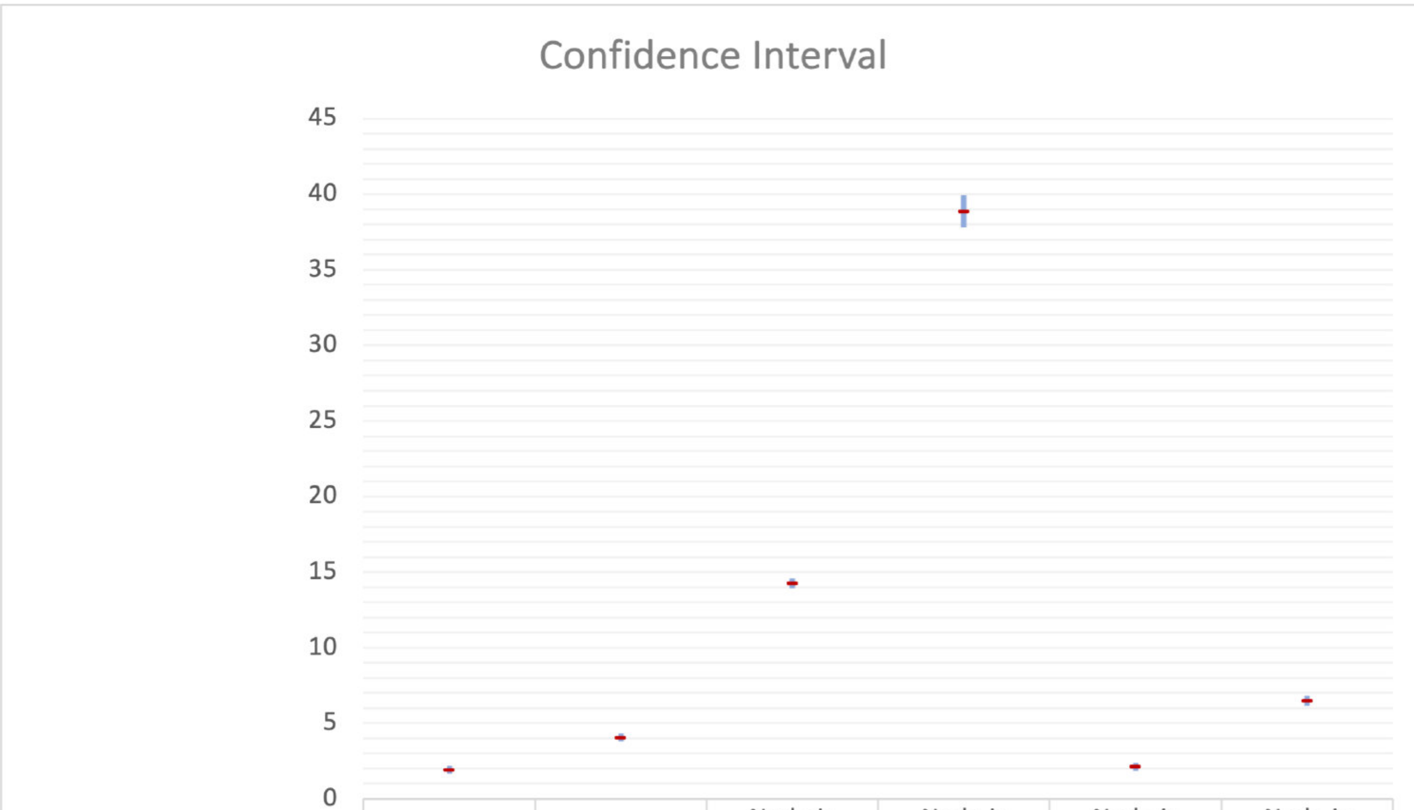
3. Discussion

I put all of the execution time data into Excel table and calculate the 95% confidence interval. The obtained results are displayed in Excel table and stored in the `Data` folder.

- For the native comparison application results:
  - Performance of Multi-threaded: the 95% confidence interval is [1.825949886, 2.011782114], the average execution time is 1.918866 s.
  - Performance of Single-threaded: the 95% confidence interval is [3.996275258, 4.131111242], the average execution time is 4.06369325 s.
- For the demo comparison application results:
  - Performance of Multi-threaded: the 95% confidence interval is [14.08598664, 14.39305336], the

average execution time is 14.23952 s.
    - Performance of Single-threaded: the 95% confidence interval is [37.97651173, 39.77634827], the average execution time is 38.87643 s.
- For the optimized demo comparison application,
    - Performance of Multi-threaded: the 95% confidence interval is [2.026125417, 2.212731083], the average execution time is 2.11942825 s.
    - Performance of Single-threaded: the 95% confidence interval is [6.343121703, 6.653196797], the average execution time is 6.49815925 s.

We can plot these confidence intervals on a figure to compare them.



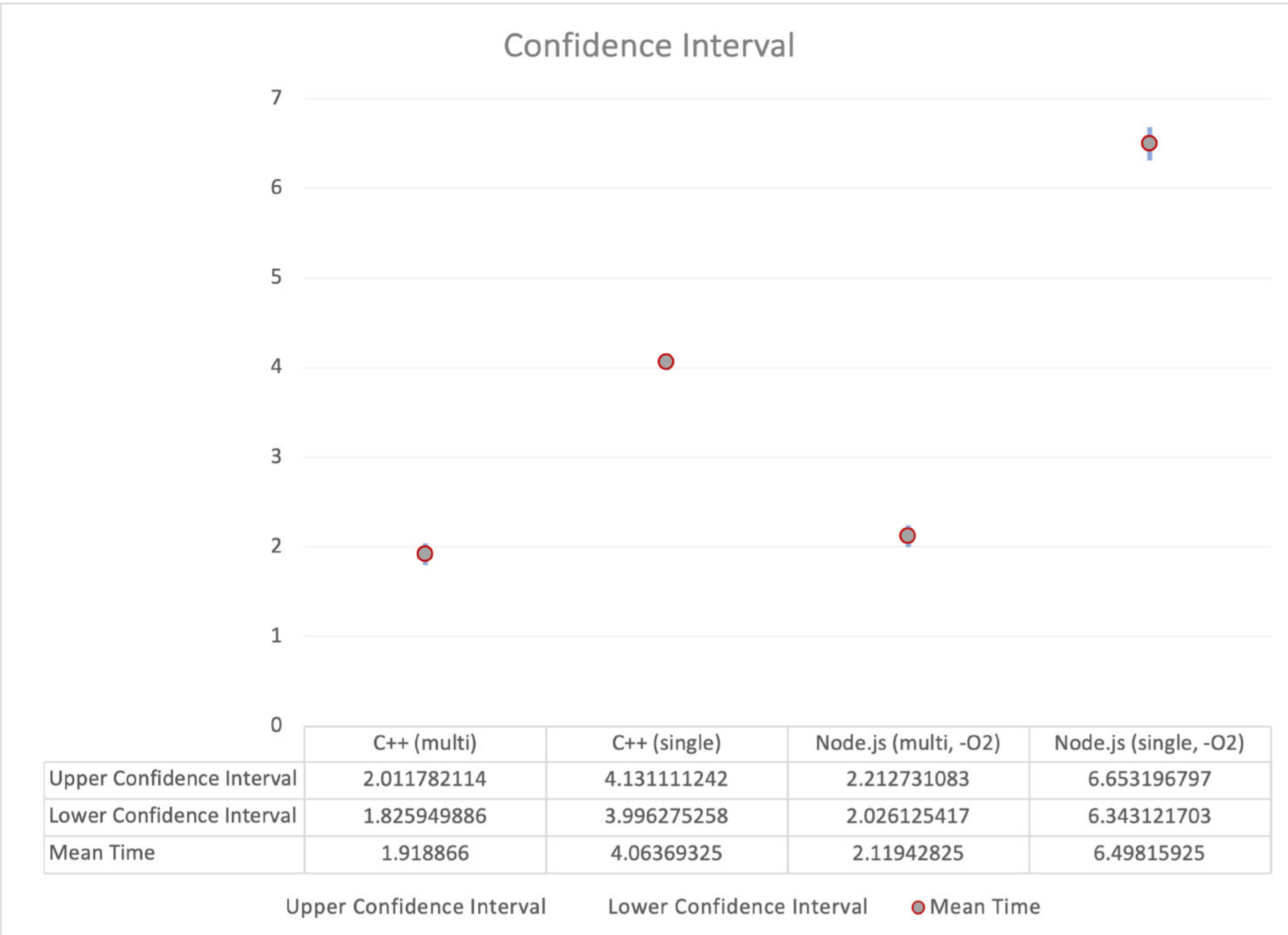
	C++ (multi)	C++ (single)	Node.js (multi)	Node.js (single)	Node.js (multi, -O2)	Node.js (single, -O2)
Upper Confidence Interval	2.011782114	4.131111242	14.39305336	39.77634827	2.212731083	6.653196797
Lower Confidence Interval	1.825949886	3.996275258	14.08598664	37.97651173	2.026125417	6.343121703
Mean Time	1.918866	4.06369325	14.23952	38.87643	2.11942825	6.49815925

Figure 7. 95% Confidence Interval Comparison

As we can see in the plot, both the results of demo application running in NodeJs without optimization are much higher than native application. Their lower confidence intervals are higher than the upper confidence interval of native application. Therefore, we can learn that the results are statistically significant.

Besides, in all cases, multi-threaded programs reach the target faster and use less execution time than single-threaded programs. we can see that compare multi-threaded results with single-threaded results, all of them are statistically significant. I think this is due to the fact that multi-threaded execution on multi-core processors allows concurrent execution of programs, allowing multiple items to be produced at the same time, and greatly reducing the waiting time for customer programs.

However, the result of optimized demo application is much faster than that of the demo application without optimization and seems very close to native application. I further draw only these four results in one figure to clearly compare them.



	C++ (multi)	C++ (single)	Node.js (multi, -O2)	Node.js (single, -O2)
Upper Confidence Interval	2.011782114	4.131111242	2.212731083	6.653196797
Lower Confidence Interval	1.825949886	3.996275258	2.026125417	6.343121703
Mean Time	1.918866	4.06369325	2.11942825	6.49815925

Figure 7. 95% Confidence Interval Comparison for native code base and optimized WASM

Now we can see the results are so close but still not overlapped, which means the results are statistically significant. Therefore, we can say the runtime of native base code is faster than optimized WASM.

4. Compilation Instructions

```
compile:
```

```
compile:
  clang++ -g -std=c++14 ./*.cpp -o output
  ./output

emcc:
  emcc -std=c++14 -pthread -s PROXY_TO_PTHREAD -s ALLOW_MEMORY_GROWTH=1 -s
NO_DISABLE_EXCEPTION_CATCHING -s LLD_REPORT_UNDEFINED -s ERROR_ON_UNDEFINED_SYMBOLS=1
./*.cpp -o example.js
  node --experimental-wasm-threads --experimental-wasm-bulk-memory example.js

emccOpt:
  emcc -std=c++14 -O2 -pthread -s PROXY_TO_PTHREAD -s ALLOW_MEMORY_GROWTH=1 -s
NO_DISABLE_EXCEPTION_CATCHING -s LLD_REPORT_UNDEFINED -s ERROR_ON_UNDEFINED_SYMBOLS=1
./*.cpp -o example.js
  node --experimental-wasm-threads --experimental-wasm-bulk-memory example.js
```

```
# Running Emscripten
cd emsdk
./emsdk activate latest
source ./emsdk_env.sh

# Runing python server
python test_server.py 8080
```

Reference

https://emscripten.org/docs/porting/pthreads.html