

# 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 <code>stop</code> 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 <code>stop</code> . 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 <code>produceA</code> or <code>produceB</code> .

## 1.1.2 Customers

### A. People

- Attributes:

Name	Type
shopping	atomic<bool>
_shop	Shop *

- 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 <code>stop</code> 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.

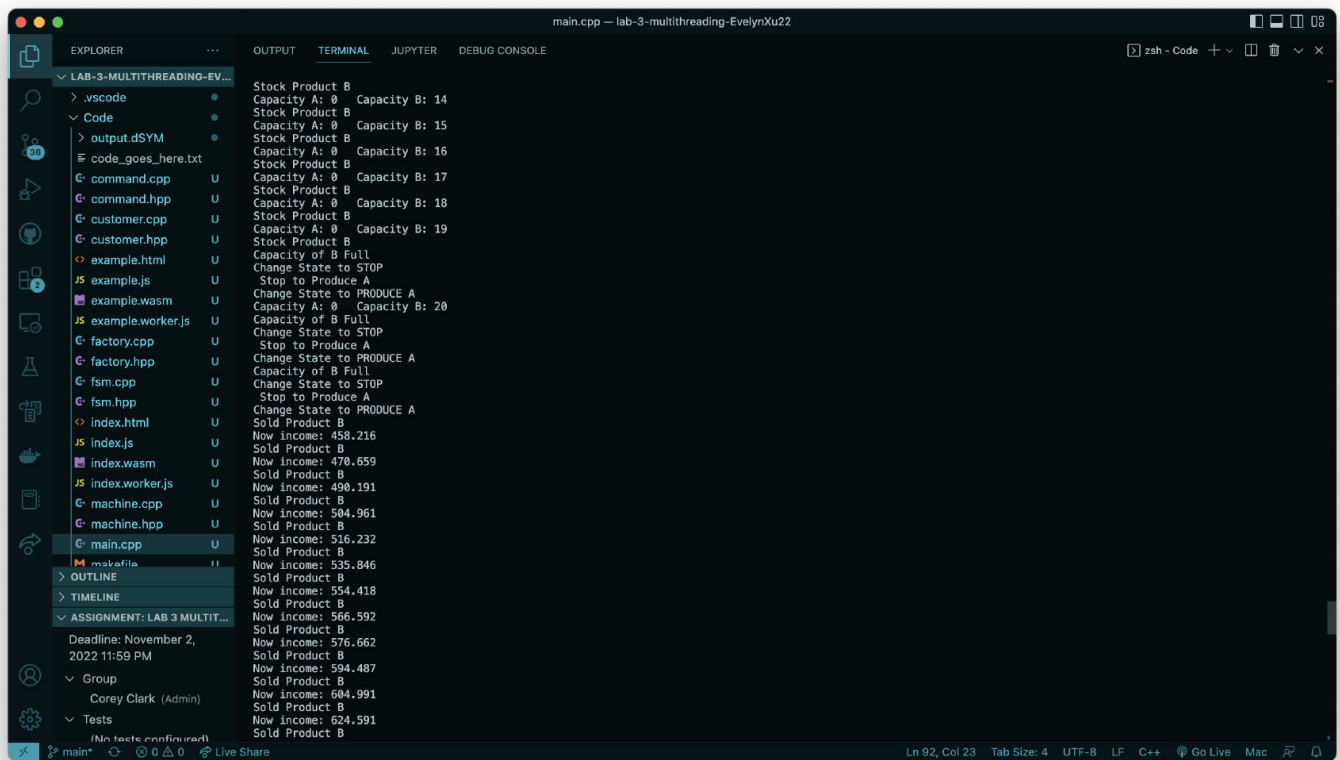
In this implementation, we have:

- 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.

```
main.cpp — lab-3-multithreading-EvelynXu22
Starting Machines (Thread)
Machine ID: 0x7000f039000 Online
Stop to Produce A
Change State to PRODUCE A
Machine ID: 0x7000f0bc000 Online
Stop to Produce A
Change State to PRODUCE A
Machine ID: 0x7000f13f000 Online
Stop to Produce A
Change State to PRODUCE A
Machine ID: 0x7000f1c2000 Online
Stop to Produce A
Change State to PRODUCE A
Stock Product A
Capacity A: 1 Capacity B: 0
Machine ID: 0x7000f245000 Online
Stop to Produce A
Change State to PRODUCE A
Stock Product A
Capacity A: 2 Capacity B: 0
Machine ID: 0x7000f2c8000 Online
Stop to Produce A
Change State to PRODUCE A
Stock Product A
Capacity A: 3 Capacity B: 0
Started 6 Machines
Customers Start Shopping (Thread)
Customer ID: 0x7000f34b000Online
Arrived 1 Customers
Not Finished
Sold Product A
Now income: 8.69143
Sold Product A
Now income: 12.5229
Sold Product A
Now income: 18.9989
Stock Product A
Capacity A: 1 Capacity B: 0
Sold Product A
Now income: 21.1322
Stock Product A
Capacity A: 1 Capacity B: 0
Sold Product A
Now income: 25.4952
Stock Product A
Capacity A: 1 Capacity B: 0
Sold Product A
Now income: 27.6948
Stock Product A
Capacity A: 1 Capacity B: 0
Sold Product A
```

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



```
main.cpp — lab-3-multithreading-EvelynXu22
EXPLORER
LAB-3-MULTITHREADING-EV...
  .vscode
  Code
  output.dSYM
  code_goes_here.txt
  command.cpp
  command.hpp
  customer.cpp
  customer.hpp
  example.html
  example.js
  example.wasm
  example.worker.js
  factory.cpp
  factory.hpp
  fsm.cpp
  fsm.hpp
  index.html
  index.js
  index.wasm
  index.worker.js
  machine.cpp
  machine.hpp
  main.cpp
  makefile
  OUTLINE
  TIMELINE
  ASSIGNMENT: LAB 3 MULTIT...
    Deadline: November 2, 2022 11:59 PM
    Group
      Corey Clark (Admin)
    Tests
      (No tests configured)
  main*

OUTPUT
Terminal
JUPYTER
DEBUG CONSOLE
zsh - Code
Stock Product B
Capacity A: 0 Capacity B: 14
Stock Product B
Capacity A: 0 Capacity B: 15
Stock Product B
Capacity A: 0 Capacity B: 16
Stock Product B
Capacity A: 0 Capacity B: 17
Stock Product B
Capacity A: 0 Capacity B: 18
Stock Product B
Capacity A: 0 Capacity B: 19
Stock Product B
Capacity of B Full
Change State to STOP
Stop to Produce A
Change State to PRODUCE A
Capacity A: 0 Capacity B: 20
Capacity of B Full
Change State to STOP
Stop to Produce A
Change State to PRODUCE A
Sold Product B
Now income: 458.216
Sold Product B
Now income: 470.659
Sold Product B
Now income: 490.191
Sold Product B
Now income: 504.961
Sold Product B
Now income: 516.232
Sold Product B
Now income: 535.846
Sold Product B
Now income: 554.418
Sold Product B
Now income: 566.592
Sold Product B
Now income: 576.662
Sold Product B
Now income: 594.487
Sold Product B
Now income: 604.991
Sold Product B
Now income: 624.591
Sold Product B
```

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

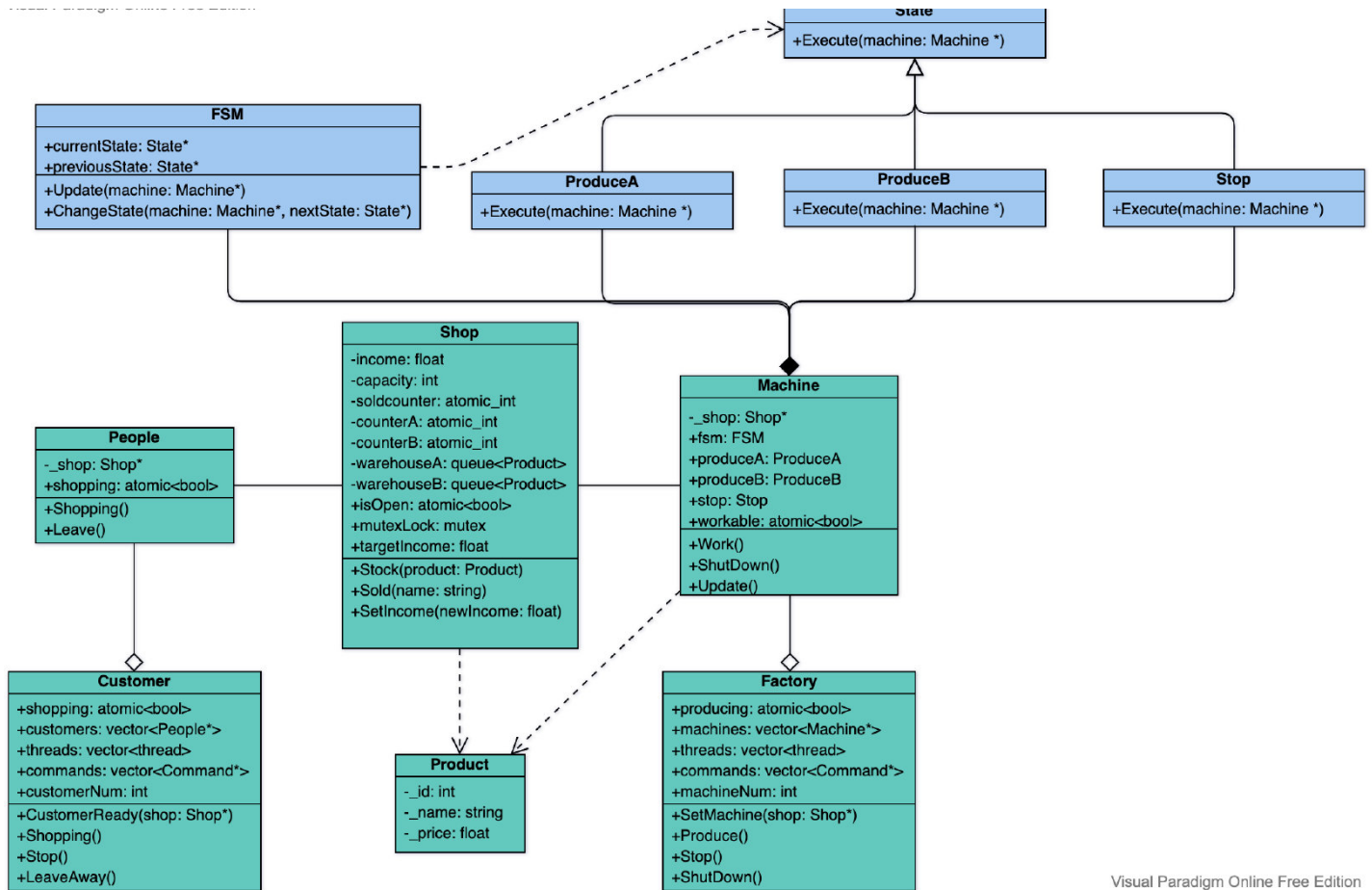
As shown in the figures, we can see that multiple machines are created and their states change during this process. If the warehouse of shop is full, they will wait for customer, and start production if any warehouse is not full.



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



Visual Paradigm Online Free Edition

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.

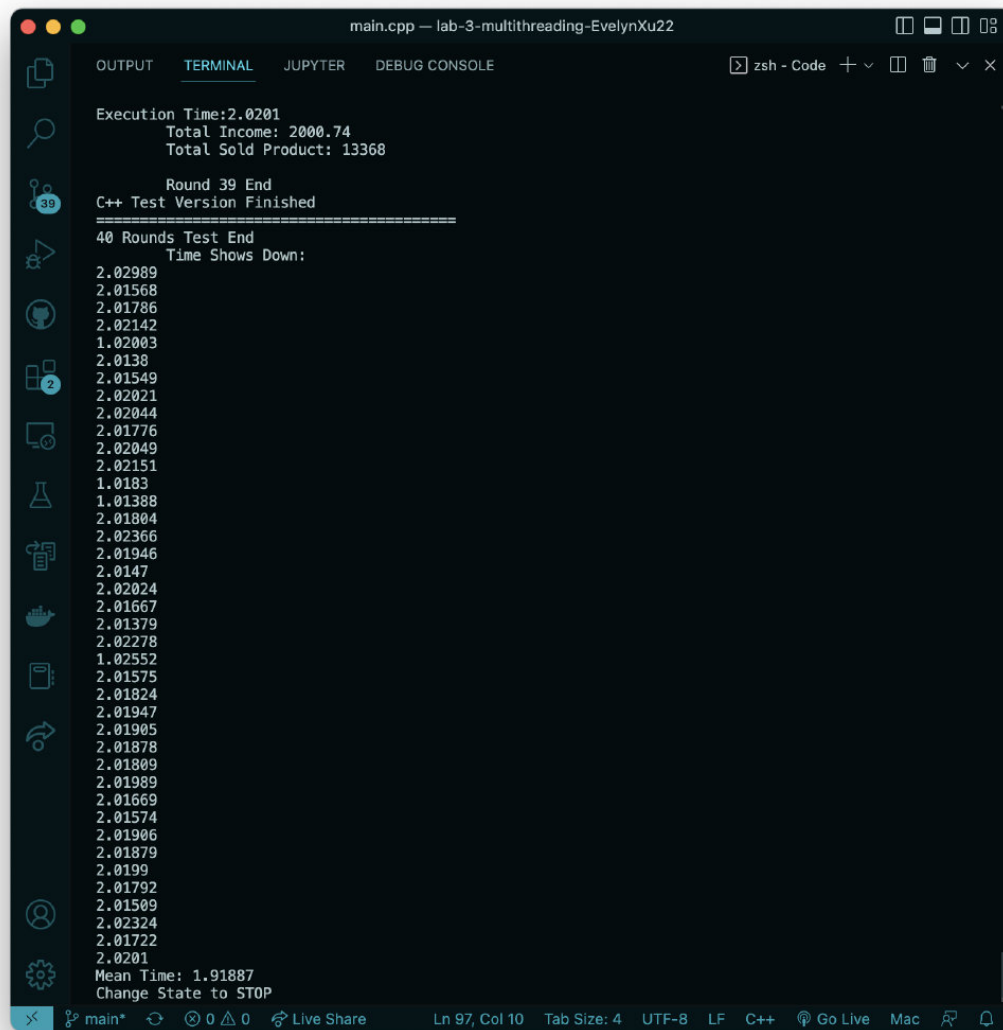
```
main.cpp — lab-3-multithreading-EvelynXu22
OUTPUT TERMINAL JUPYTER DEBUG CONSOLE
zsh - Code + - [ ] [ ] [ ] [ ]

Execution Time:2.0201
Total Income: 2000.74
Total Sold Product: 13368

Round 39 End
C++ Test Version Finished
=====
40 Rounds Test End
Time Shows Down:
2.02989
2.01568
2.01786
2.02142
1.02003
2.0138
2.01549
2.02021
2.02044
2.01776
2.02049
2.02151
1.0183
1.01388
2.01804
2.02366
2.01946
2.0147
2.02024
2.01667
2.01379
2.02278
1.02552
2.01575
2.01824
2.01947
2.01905
2.01878
2.01809
2.01989
2.01669
2.01574
2.01906
2.01879
2.0199
2.01792
2.01509
2.02324
2.01722
2.0201
Mean Time: 1.91887
Change State to STOP

main* 0 0 0 Live Share Ln 97, Col 10 Tab Size: 4 UTF-8 LF C++ Go Live Mac
```

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



```
main.cpp — lab-3-multithreading-EvelynXu22
OUTPUT TERMINAL JUPYTER DEBUG CONSOLE
zsh - Code + - [ ] [ ] [ ] [ ]
Execution Time:2.0201
Total Income: 2000.74
Total Sold Product: 13368
Round 39 End
C++ Test Version Finished
=====
40 Rounds Test End
Time Shows Down:
2.02989
2.01568
2.01786
2.02142
1.02003
2.0138
2.01549
2.02021
2.02044
2.01776
2.02049
2.02151
1.0183
1.01388
2.01804
2.02366
2.01946
2.0147
2.02024
2.01667
2.01379
2.02278
1.02552
2.01575
2.01824
2.01947
2.01905
2.01878
2.01809
2.01989
2.01669
2.01574
2.01906
2.01879
2.0199
2.01792
2.01509
2.02324
2.01722
2.0201
Mean Time: 1.91887
Change State to STOP
main* 0 0 0 Live Share Ln 97, Col 10 Tab Size: 4 UTF-8 LF C++ Go Live Mac
```

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.

```
167 #ifdef __EMSCRIPTEN__
168 for (int i = 0; i < RUN_TIMES; i++)
169 {
170     cout << "Round " << i << " Start" << endl;
171     mainMutex.lock();
172     startTime = chrono::high_resolution_clock::now();
173     Start();
174
175
176     mainMutex.unlock();
177     mainMutex.lock();
178
179     Stop();
180
181     // Record Execution Time
182     endTime = chrono::high_resolution_clock::now();
183     chrono::duration<double> time_span = chrono::duration_cast<chrono::duration<double>>(endTime - startTime);
184
185     // Log Results
186     cout << "\nExecution Time:" << time_span.count() << endl;
187     times.push_back(time_span.count());
188     cout << "\tTotal Income: " << shop.GetIncome() << endl;
189     cout << "\tTotal Sold Product: " << shop.GetSoldCount() << endl;
190
191     // Reset the states of shop
```

Figure 6. The Emscripten Version Test Application

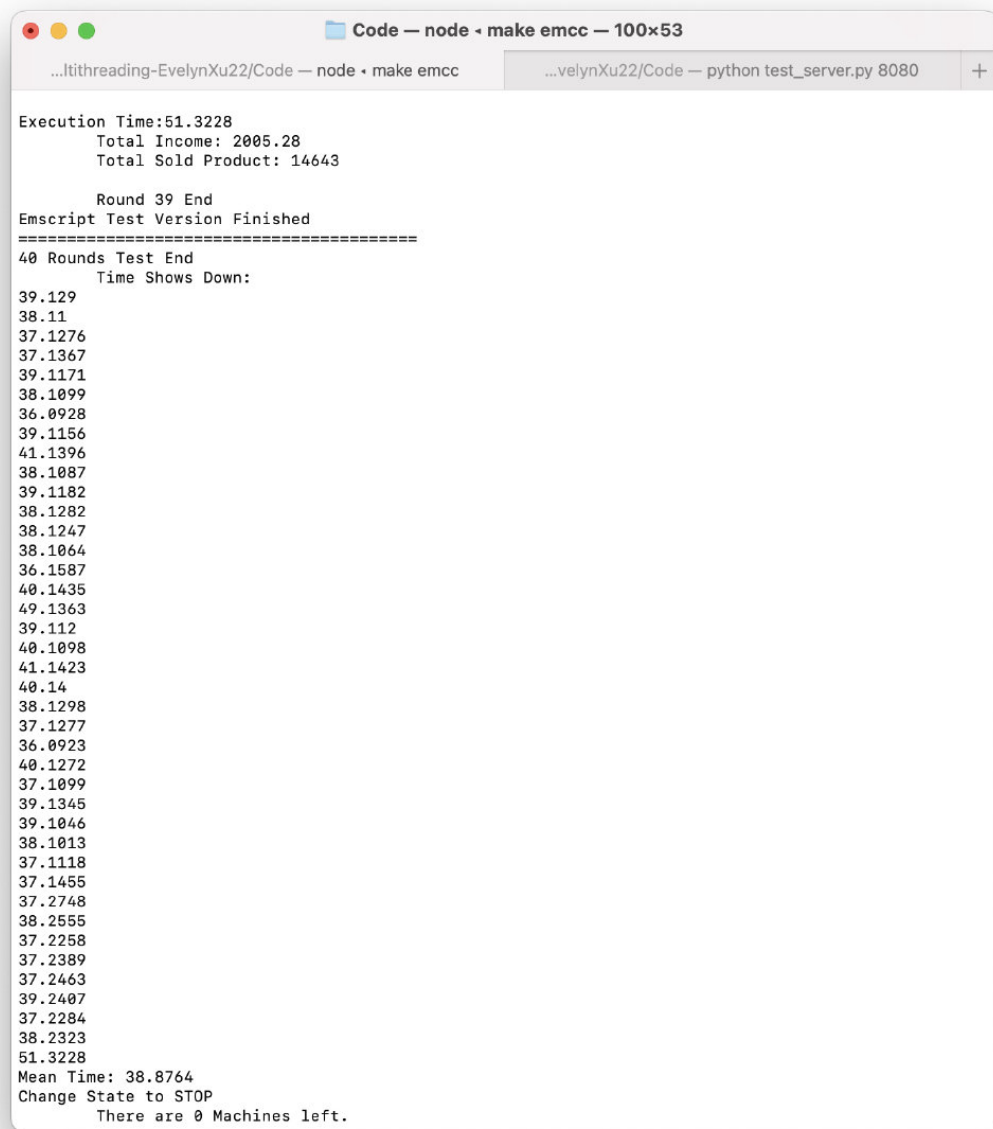
```
Code — node • make emcc — 100x53
...lthreading-EvelynXu22/Code — node • make emcc
...velynXu22/Code — python test_server.py 8080 +

There are 0 threads left.

Execution Time:15.0681
Total Income: 2005.28
Total Sold Product: 14643

Round 39 End
Emscript Test Version Finished
=====
40 Rounds Test End
Time Shows Down:
14.0943
14.0783
14.0625
14.0663
14.0537
14.0629
13.0512
15.0581
15.0682
13.0497
14.0687
14.0795
15.0595
15.0622
15.0667
14.0729
14.0662
14.0523
14.0716
15.0721
14.0484
14.0669
14.0641
14.0678
15.0628
14.0554
14.0645
14.0644
15.0548
14.0596
14.0557
14.0715
14.0634
14.0559
14.0732
14.0562
14.0729
14.0708
14.0675
15.0681
Mean Time: 14.2395
Change State to ST0P
```

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



```
Code — node • make emcc — 100x53
...lthreading-EvelynXu22/Code — node • make emcc
...velynXu22/Code — python test_server.py 8080 +

Execution Time:51.3228
  Total Income: 2005.28
  Total Sold Product: 14643

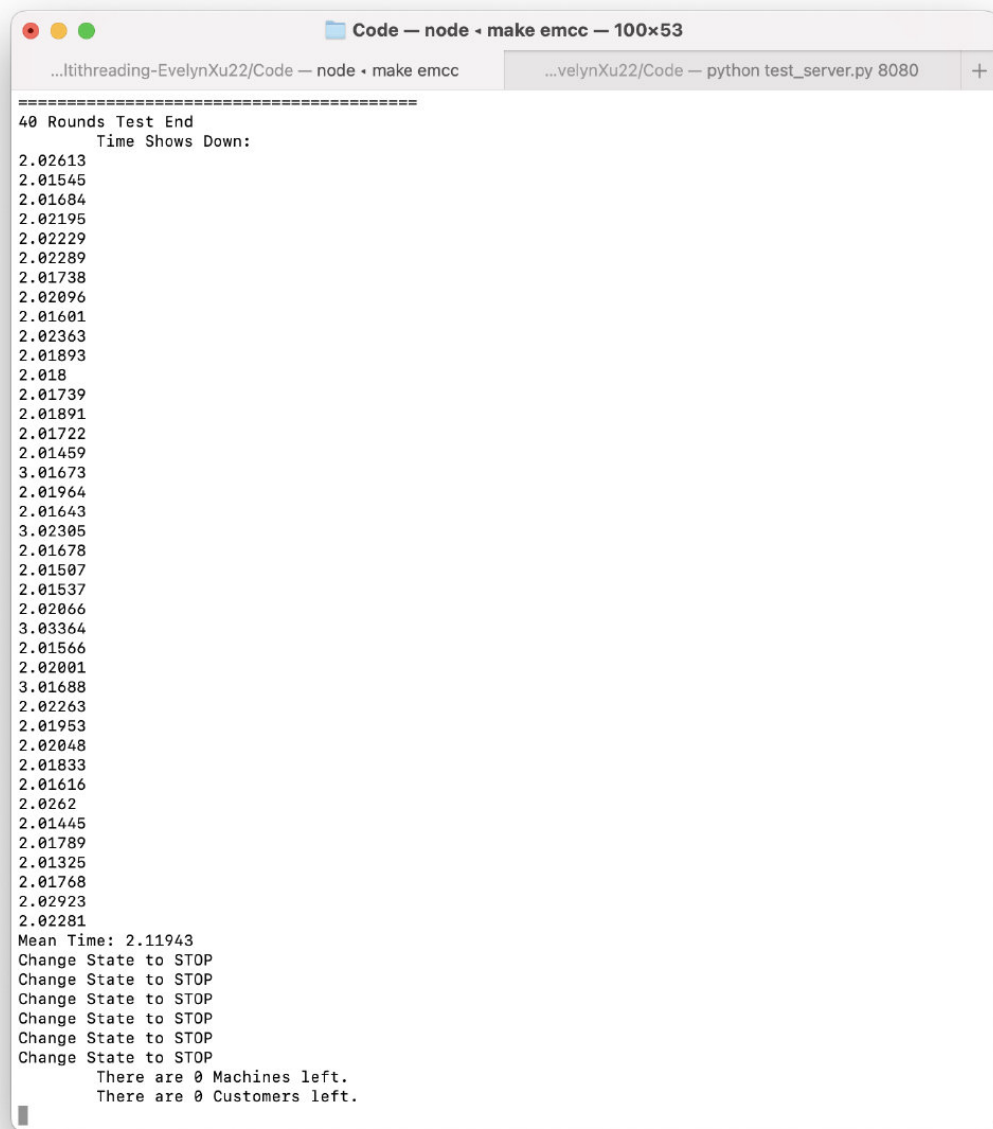
Round 39 End
Emscript Test Version Finished
=====
40 Rounds Test End
Time Shows Down:
39.129
38.11
37.1276
37.1367
39.1171
38.1099
36.0928
39.1156
41.1396
38.1087
39.1182
38.1282
38.1247
38.1064
36.1587
40.1435
49.1363
39.112
40.1098
41.1423
40.14
38.1298
37.1277
36.0923
40.1272
37.1099
39.1345
39.1046
38.1013
37.1118
37.1455
37.2748
38.2555
37.2258
37.2389
37.2463
39.2407
37.2284
38.2323
51.3228
Mean Time: 38.8764
Change State to STOP
There are 0 Machines left.
```

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
```

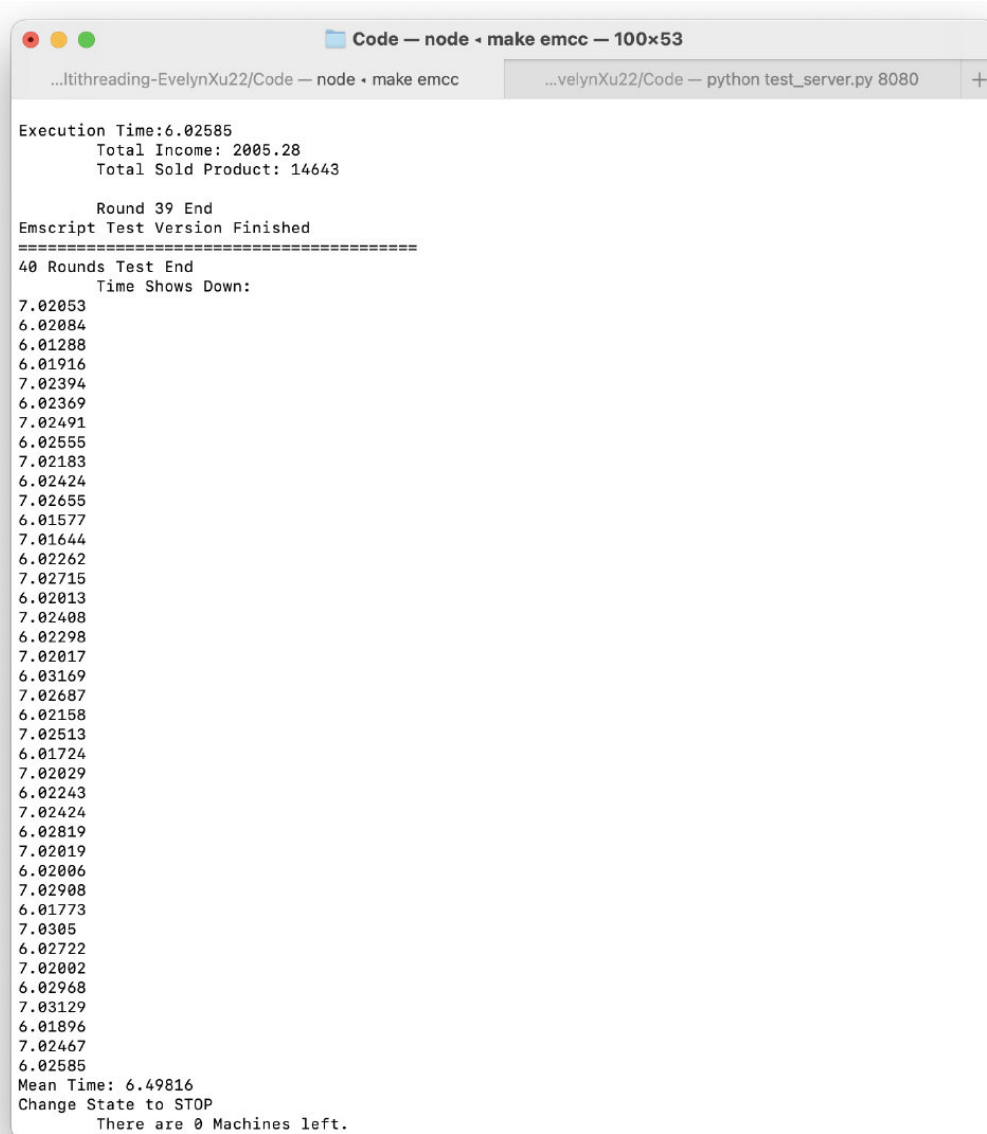


```
Code — node • make emcc — 100x53
...tithreading-EvelynXu22/Code — node • make emcc
...velynXu22/Code — python test_server.py 8080 +

=====
40 Rounds Test End
Time Shows Down:
2.02613
2.01545
2.01684
2.02195
2.02229
2.02289
2.01738
2.02096
2.01601
2.02363
2.01893
2.018
2.01739
2.01891
2.01722
2.01459
3.01673
2.01964
2.01643
3.02305
2.01678
2.01507
2.01537
2.02066
3.03364
2.01566
2.02001
3.01688
2.02263
2.01953
2.02048
2.01833
2.01616
2.0262
2.01445
2.01789
2.01325
2.01768
2.02923
2.02281
Mean Time: 2.11943
Change State to STOP
Change State to STOP
Change State to STOP
Change State to STOP
Change State to STOP
Change State to STOP
Change State to STOP
There are 0 Machines left.
There are 0 Customers left.
```

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





```
Code — node • make emcc — 100×53
...lithreading-EvelynXu22/Code — node • make emcc
...velynXu22/Code — python test_server.py 8080 +

Execution Time:6.02585
Total Income: 2005.28
Total Sold Product: 14643

Round 39 End
Emscript Test Version Finished
=====
40 Rounds Test End
Time Shows Down:
7.02053
6.02084
6.01288
6.01916
7.02394
6.02369
7.02491
6.02555
7.02183
6.02424
7.02655
6.01577
7.01644
6.02262
7.02715
6.02013
7.02408
6.02298
7.02017
6.03169
7.02687
6.02158
7.02513
6.01724
7.02029
6.02243
7.02424
6.02819
7.02019
6.02006
7.02908
6.01773
7.0305
6.02722
7.02002
6.02968
7.03129
6.01896
7.02467
6.02585
Mean Time: 6.49816
Change State to STOP
There are 0 Machines left.
```

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.

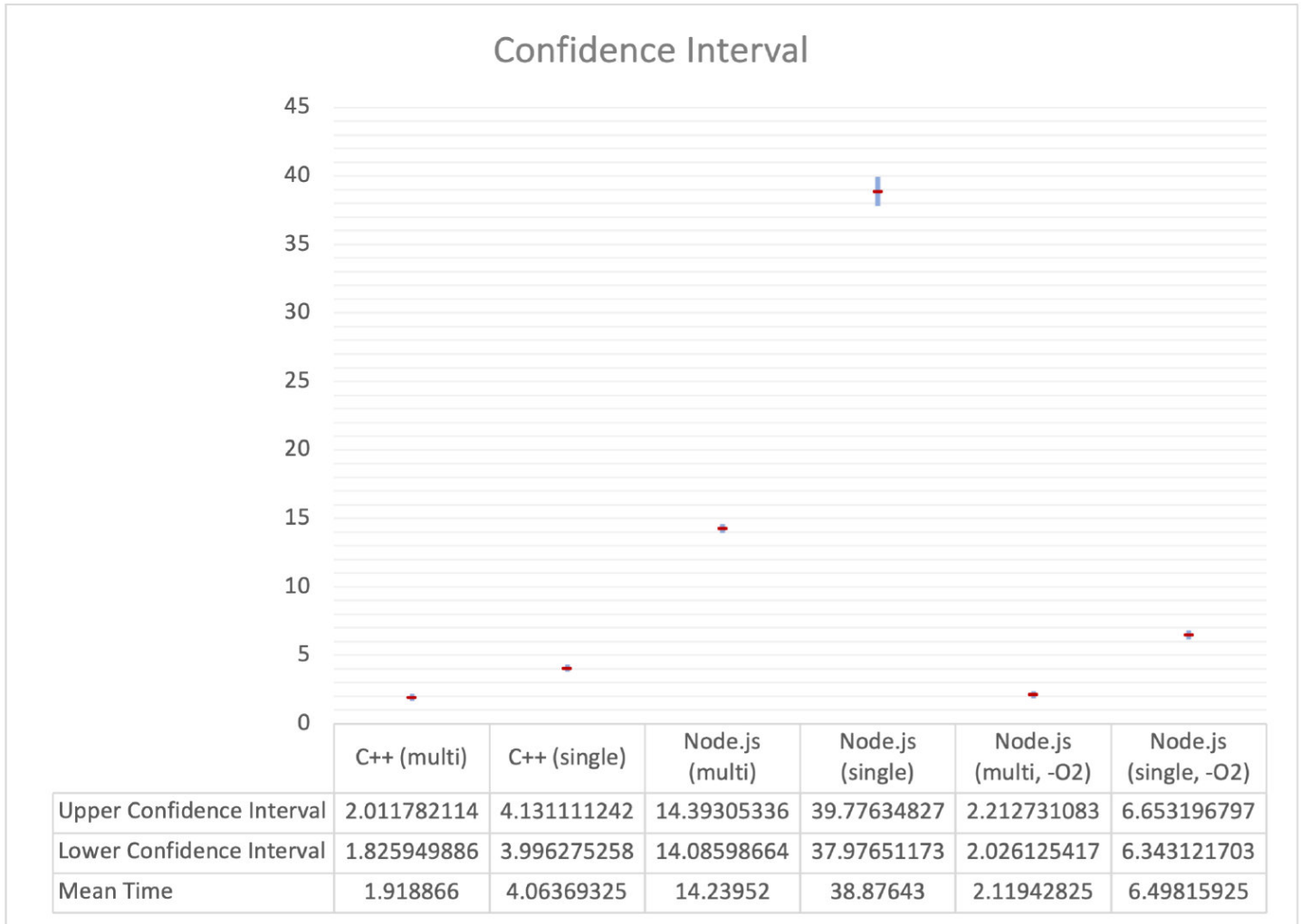


Figure 7. 95% Confidence Interval Comparison

As we can see in the plot, both the results of demo application running in Node.js 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.

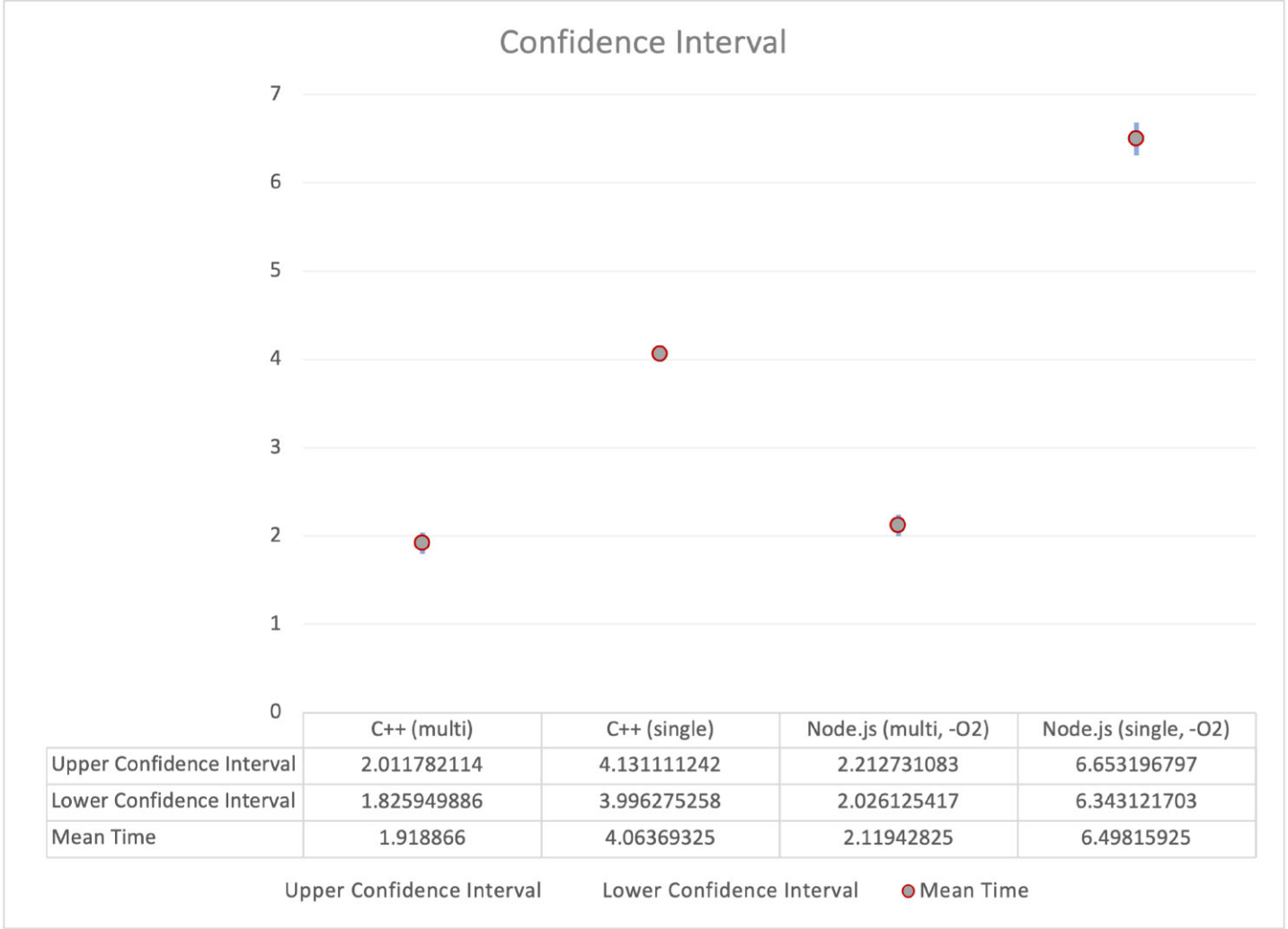


Figure 7. 95% Confidence Interval Comparison for native code base and optimized WASM

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:

```
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>