Final Project

Operating Systems Course

Computer Science at Ariel University

Samuel Lazareanu
Lior Trachtman

Code Solution

https://github.com/SamuraiPolix/Operating-Systems-Final-Project.git

Table of Contents

Overview

We decided to implement a simpel Server using threads for testing purposes - `Server.cpp`
We also implemented the server using Leader-Follower Thread Pool as required - `ThreadPoolServer.cpp`, which works with `ThreadPool.cpp`, `ThreadPool.hpp`

`make server` runs the normal server
`make threadpool_server` runs the ThreadPool server

We create a thread pool class that manages a pool of worker threads, implement the Leader-Follower pattern where one thread acts as the leader and the others as followers. With a list of tasks assigned to “workers”.

The server reads the graph from the client, processes it to solve the MST problem, computes the required metrics, and sends the results back to the client. The Leader-Follower thread pool uses a pool of worker threads to handle client requests concurrently, a thread for each client.

Why use Leader-Follower Thread Pool?
In this pattern, multiple threads take turns being the leader. The leader thread waits for an event (for example, a client request), processes it, and then becomes a follower, allowing another thread to become the leader. This pattern is useful for efficiently managing a pool of threads to handle multiple tasks concurrently, in our case, multiple clients.

In the next pages, we will provide a deep analysis for our code.

Code coverage (GCOV)

All the GCOV files and stdouts are located at `/gcov_outputs`

By using `make coverage`, we recompile the program with coverage flags, and run the servers one by one (first server is used with threads as usual, second server is using ThreadPool), we then connect from a splitted terminal to the server using `nc localhost 9034` and send the commands, we test all the commands possible and mistakes that users can make.

At the end, we move all the gcov related files (.gcda, .gcno, .gcov) to their folder to keep our root folder clean

```
samurai@DESKTOP-O04V85G:~/cs/Operating-Systems/Operating-Systems-Ex4/q1-4$ make coverage
make clean
make[1]: Entering directory '/home/samurai/cs/Operating-Systems/Operating-Systems-Ex4/q1-4'
rm -f main.o euler_circuit.o euler_circuit
make[1]: Leaving directory '/home/samurai/cs/Operating-Systems/Operating-Systems-Ex4/q1-4'
make all COVERAGE=1 DEBUG=0
make[1]: Entering directory '/home/samurai/cs/Operating-Systems/Operating-Systems-Ex4/q1-4'
g++ -std=c++17 -Wall -Wextra -fprofile-arcs -ftest-coverage -c main.cpp
g++ -std=c++17 -Wall -Wextra -fprofile-arcs -ftest-coverage -c euler_circuit.cpp
g++ -std=c++17 -Wall -Wextra -fprofile-arcs -ftest-coverage main.o euler_circuit.o -o euler_circuit
make[1]: Leaving directory '/home/samurai/cs/Operating-Systems/Operating-Systems-Ex4/q1-4'
 Valid input
./euler_circuit -n 100 -e 300 -s 1 > /dev/null || true
gcov main.cpp euler_circuit.cpp > gcov_outputs/stdout1.txt
 Invalid argument
./euler_circuit -x 5 > /dev/null || true
./euler_circuit: invalid option -- 'x'
Usage: ./euler_circuit -n num_vertices -e num_edges -s seed
gcov main.cpp euler_circuit.cpp > gcov_outputs/stdout2.txt
 Invalid argument values
./euler_circuit -n -1 -e 300 -s 1 > /dev/null || true
Number of vertices must be positive.
Usage: ./euler_circuit -n num_vertices -e num_edges -s seed
./euler_circuit -n 100 -e -1 -s 1 > /dev/null || true
Number of edges cannot be negative.
Usage: ./euler_circuit -n num_vertices -e num_edges -s seed
./euler_circuit -n 2 -e 300 -s 1 > /dev/null || true
Too many edges for the number of vertices.
Usage: ./euler_circuit -n num_vertices -e num_edges -s seed
gcov main.cpp euler_circuit.cpp > gcov_outputs/stdout3.txt
 0 edges
./euler_circuit -n 100 -e 0 -s 1 > /dev/null || true
/home/samurai/cs/Operating-Systems/Operating-Systems-Ex4/q1-4/show_graph.py:8: UserWarning: loadtxt: input contained no data: "graph.txt"
  data = np.loadtxt('graph.txt', skiprows=1)
gcov main.cpp euler_circuit.cpp > gcov_outputs/stdoutFinal.txt
mv *.gcov gcov_outputs || true
mv *.gcda gcov_outputs || true
mv *.gcno gcov_outputs || true
samurai@DESKTOP-O04V85G:~/cs/Operating-Systems/Operating-Systems-Ex4/q1-4$
```

We can see that after the test:

Server.cpp had 86%, that's because we added a lot of error handling, for the sockets, and we faced 0 errors with the sockets so the code wasn't executed.

```
gcov_outputs > ≡ stdout.txt
1   File 'Server.cpp'
2   Lines executed:85.82% of 141
3   Creating 'Server.cpp.gcov'
```

Example code that wasn't executed:

```
    -:  168:    // Allow reuse of address
    1:  169:    int yes = 1;
    1:  170:    if (setsockopt(server, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
#####:  171:        std::cerr << "setsockopt: Could not set socket options.\n";
#####:  172:        return 2;
    -:  173:    }
    -:  174:
    -:  175:    // Bind to port
    1:  176:    if (bind(server, (sockaddr *) &serverAddr, sizeof(serverAddr)) == -1) {
#####:  177:        std::cerr << "bind: Could not bind to port " << PORT << ".\n";
#####:  178:        return 3;
    -:  179:    }
    -:  180:
    -:  181:    // Listen on port for incoming connections
    1:  182:    if (listen(server, MAXCONNECTIONS) == -1) {
#####:  183:        std::cerr << "listen: Could not listen on port " << PORT << ".\n";
    -:  184:        // print error
#####:  185:        std::cout << "Error: " << strerror(errno) << "\n";
#####:  186:        return 4;
    -:  187:    }
    -:  188:
```

Without them, code coverage would have been around 100%.

This is true for all other source files:
Test.cpp:

```
117   File 'Test.cpp'
118   Lines executed:78.46% of 65
119   Creating 'Test.cpp.gcov'
```

The reason is we tested an unconnected graph and then ran over all edges inside solution to see all expected edges exist, but they are both empty so we didn't go into the first loop.
This was kept because it's a good test, we just don't always need it

```
30      #####:   29:        bool found = false;
31      #####:   30:        for (const Edge& expected : expectedEdges) {
32      #####:   31:            if (expected.u == edge.u && expected.v == edge.v && expected.weight == edge.weight) {
33      #####:   32:                found = true;
34      #####:   33:                break;
35          -:   34:            }
36          -:   35:        }
37      #####:   36:        std::cout << "Edge: " << edge.u << " -> " << edge.v << " (" << edge.weight << ")\n";
38      #####:   37:        CHECK(found);
39          -:   38:    }
```

MSTFactory.cpp:

```
gcov_outputs > stdout1.txt
1   File 'MSTFactory.cpp'
2   Lines executed:66.67% of 9
3   Creating 'MSTFactory.cpp.gcov'
```

Default is kept there for safety if someone else uses our factory with an invalid MST.

```
        8:    3:std::unique_ptr<MSTSolver> MSTFactory::createSolver(MSTType type) {
        8:    4:    switch (type) {
        5:    5:        case BORUVKA:
        5:    6:            return std::unique_ptr<MSTSolver>(new BoruvkaSolver());
        3:    7:        case PRIM:
        3:    8:            return std::unique_ptr<MSTSolver>(new PrimSolver());
    #####:    9:        default:
    #####:   10:            std::cout << "Invalid MST type" << std::endl;
    #####:   11:            return nullptr;
        -:   12:    }
```

MSTSolver.cpp:

```
File 'MSTSolver.cpp'
Lines executed:83.33% of 132
Creating 'MSTSolver.cpp.gcov'
```

We made functions to calculate metrics with the graph and not the mst result, that way they don’t have to calculate it first, we use the functions with the calculation to calculate only once for all metrics, again, this is good practice but ruins coverage data.

```
        -:    9:// --------------------------- Calculate Metrics ---------------------------
    #####:   10:int MSTSolver::totalWeight(Graph& graph) {
    #####:   11:    std::vector<Edge> mst = solve(graph);
    #####:   12:    return totalWeight(mst);
    #####:   13:}
        -:   14:
    #####:   15:int MSTSolver::longestDistance(Graph& graph) {
    #####:   16:    std::vector<Edge> mst = solve(graph);
    #####:   17:    return longestDistance(mst);
    #####:   18:}
        -:   19:
    #####:   20:int MSTSolver::shortestDistance(Graph& graph) {
    #####:   21:    std::vector<Edge> mst = solve(graph);
    #####:   22:    return shortestDistance(mst);
    #####:   23:}
        -:   24:
    #####:   25:double MSTSolver::averageDistance(Graph& graph) {
    #####:   26:    std::vector<Edge> mst = solve(graph);
    #####:   27:    return totalWeight(mst);
    #####:   28:}
```

ThreadPoolServer.cpp:

```
File 'ThreadPoolServer.cpp'
Lines executed:85.83% of 127
Creating 'ThreadPoolServer.cpp.gcov'
```

All the code that is not covered is error handling

```
171        1:  170:    if (setsockopt(server, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
172    #####:  171:        std::cerr << "setsockopt: Could not set socket options.\n";
173    #####:  172:        return 2;
174        -:  173:    }
175        -:  174:
176        -:  175:    // Bind to port
177        1:  176:    if (bind(server, (sockaddr *) &serverAddr, sizeof(serverAddr)) == -1) {
178    #####:  177:        std::cerr << "bind: Could not bind to port " << PORT << ".\n";
179    #####:  178:        return 3;
180        -:  179:    }
181        -:  180:
182        -:  181:    // Listen on port for incoming connections
183        1:  182:    if (listen(server, MAXCONNECTIONS) == -1) {
184    #####:  183:        std::cerr << "listen: Could not listen on port " << PORT << ".\n";
185        -:  184:        // print error
186    #####:  185:        std::cout << "Error: " << strerror(errno) << "\n";
187    #####:  186:        return 4;
```

Profiling

GPROF

All the GPROF files and stdouts are located at `/gprof_outputs`

By using `make profile` we get the profiling data, by recompiling the program with profiling flags and running it. We created a python script to generate large data, we connect to the server with nc localhost PORT and paste the input.

For 50,000 vertices and 50,000 edges, total cumulative time is 0.09 for all.
For the algorithms themself:
Prim took 0.05s, 57.7% of total time

```
467   -----------------------------------------------
468                  0.00    0.05       1/1           main [1]
469   [2]     57.7   0.00    0.05       1         PrimSolver::solve(Graph&) [2]
```

Boruvka took 0.03s, 34.9% of total time

```
508   -----------------------------------------------
509                  0.00    0.03       1/1           main [1]
510   [3]     34.9   0.00    0.03       1         BoruvkaSolver::solve(Graph&) [3]
```

Meaning Boruvka is slightly faster

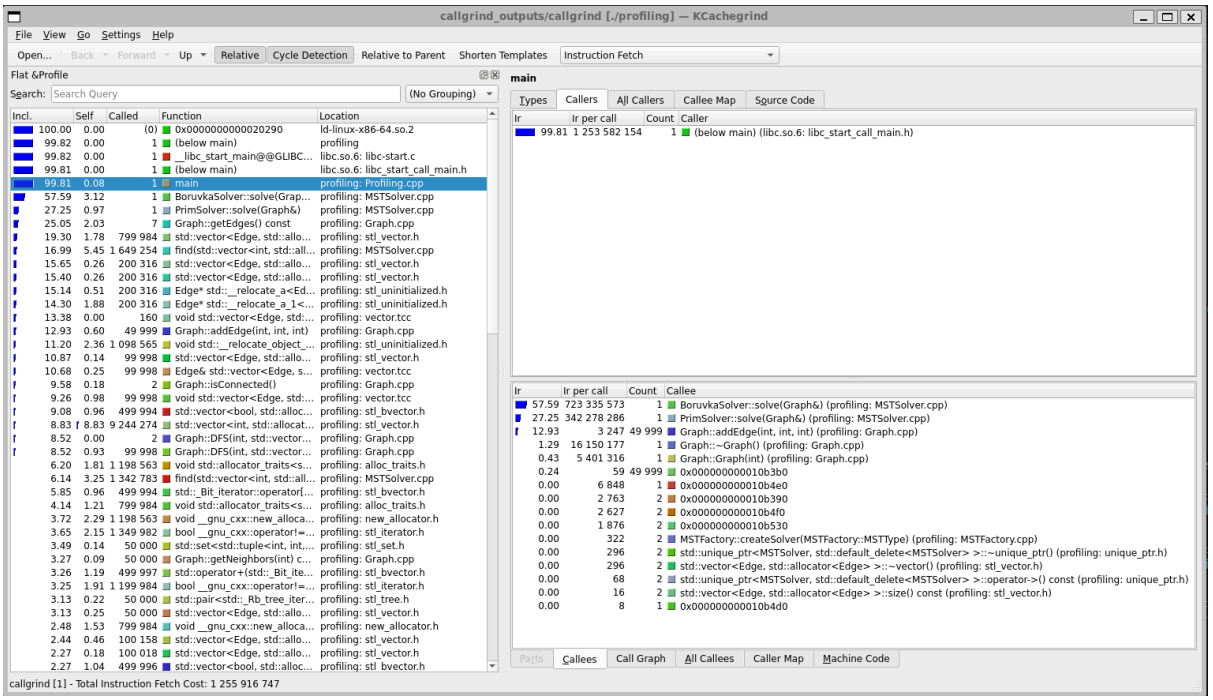
Callgrind (using kcachegrind to visualize)

We ran callgrind, using `make callgrind`

All outputs, including the stdout of the run are in `/callgrind_outputs`

```
==801390== Callgrind, a call-graph generating cache profiler
==801390== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==801390== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==801390== Command: ./profiling
==801390==
==801390== For interactive control, run 'callgrind_control -h'.
==801390==
==801390== Events    : Ir
==801390== Collected : 1255916747
==801390==
==801390== I   refs:      1,255,916,747
```

And then ran kcachegrind to visualize the graph



Valgrind:

Memcheck

The test results are located at `/valgrind_outputs`

By using `make valgrind`, the makefile runs

```
valgrind: $(TARGET)
    valgrind --tool=memcheck $(VALGRIND_FLAGS) ./$(TARGET) -n 10000 -e 30000 -s 1 2> valgrind_output.txt
```

Results:

Using Server.cpp:

```
==802508==
==802508== HEAP SUMMARY:
==802508==     in use at exit: 0 bytes in 0 blocks
==802508==   total heap usage: 81 allocs, 81 frees, 197,361 bytes allocated
==802508==
==802508== All heap blocks were freed -- no leaks are possible
==802508==
==802508== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Using ThreadPoolServer.cpp:

```
--803190-- REDIR: 0x4b543e0 (libc.so.6:free) redirected to 0x484b210 (free)
==803190==
==803190== HEAP SUMMARY:
==803190==     in use at exit: 0 bytes in 0 blocks
==803190==   total heap usage: 95 allocs, 95 frees, 80,868 bytes allocated
==803190==
==803190== All heap blocks were freed -- no leaks are possible
==803190==
==803190== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

No leaks :))

Helgrind

Used `make valgrind_helgrind` to run helgrind

Full output is at `/valgrind_outputs/helgrind.txt`

Server.cpp:

```
==805780== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

ThreadPoolServer.cpp:

```
==805985== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 208 from 14)
```

Cg

Used `make valgrind_cachegrind` to un cg

Full output is at `/valgrind_outputs/cachegrind.txt`

Server.cpp:

```
==806942==
==806942== I   refs:      2,601,428
==806942== I1  misses:        5,408
==806942== LLi misses:        3,331
==806942== I1  miss rate:      0.21%
==806942== LLi miss rate:      0.13%
==806942==
==806942== D   refs:        892,275  (640,534 rd   + 251,741 wr)
==806942== D1  misses:       17,383  ( 14,687 rd   +   2,696 wr)
==806942== LLd misses:        9,776  (  7,996 rd   +   1,780 wr)
==806942== D1  miss rate:       1.9% (    2.3%     +     1.1%  )
==806942== LLd miss rate:       1.1% (    1.2%     +     0.7%  )
==806942==
==806942== LL refs:          22,791  ( 20,095 rd   +   2,696 wr)
==806942== LL misses:        13,107  ( 11,327 rd   +   1,780 wr)
==806942== LL miss rate:        0.4% (    0.3%     +     0.7%  )
```

ThreadPoolServer.cpp:

```
==807819==
==807819== I   refs:      2,668,907
==807819== I1  misses:        5,938
==807819== LLi misses:        3,362
==807819== I1  miss rate:      0.22%
==807819== LLi miss rate:      0.13%
==807819==
==807819== D   refs:        924,514  (660,356 rd   + 264,158 wr)
==807819== D1  misses:       18,450  ( 15,313 rd   +   3,137 wr)
==807819== LLd misses:       10,163  (  8,048 rd   +   2,115 wr)
==807819== D1  miss rate:       2.0% (    2.3%     +     1.2%  )
==807819== LLd miss rate:       1.1% (    1.2%     +     0.8%  )
==807819==
==807819== LL refs:          24,388  ( 21,251 rd   +   3,137 wr)
==807819== LL misses:        13,525  ( 11,410 rd   +   2,115 wr)
==807819== LL miss rate:        0.4% (    0.3%     +     0.8%  )
```