Github server code :
https://github.com/sena1771/multicore_web_server.git

Summary

This project is designed for optimizing the performance of a simple web server for multiple incoming requests. So the focused problem was handling the multiple concurrent incoming requests.

Server is designed with taking the number of threads as a parameter. At the server code is used thread pooling approach where multiple worker threads created to handle incoming client requests.

Producer-consumer model applied with a queue that stored incoming connections and worker threads would consume requests from the queue.

To ensure thread safety, mutex locks and condition variables were used to synchronize access to the queue and prevented the race conditions when multiple threads accessed it simultaneously.

Series of tests were conducted using ApacheBench which is a testing tool for performance analysis. These test sends multiple requests and multiple concurrent requests too. It measures metrics like requests per second, time per request and total execution time. These tests applied for analyzing results for different thread counts and different request counts.Results showed that increasing the number of threads improved throughput and reduced latency.

Introduction

Problem Statement and Objective

Actually, problem is if demand fort his simple web server grows, servers must be able to handle multiple simultaneous connections efficiently, minimizing latency and maximizing throughput.

And goals are

- Observing the number of threads for handling of requests.
- Measuring the resource usage and performance.

Context

For multicore programming, managing concurrent tasks effectively is critical for taking full advantage of modern hardware capabilities.

So, efficiently processing the requests are essential for web server applications.

This project addresses the growing demand for servers capable of handling large numbers of concurrent users which is significantly important in web services. With experimenting the different thread configurations, the project aims to provide insights into how web serves can scale effectively on multicore systems.

Methodology

Approach

The server was implemented using **POSIX threads (pthreads)**, a widely adopted and efficient threading framework in C that provides low-level control over thread management.

- Thread Pooling Approach :
  Given parameter count of threads are created and assigned to handle client connections.
- Producer Consumer Model :
  Incoming requests (client connections) are placed in a queue and worker threads picked up these requests from the queue for processing.
  Producer thread is the main thread(accept_connections) and consumers are worker threads.
- Synchronization :

Since multiple threads access the request queue, mutex locks and condition variables were used to ensure thread safety and prevent race conditions.

- Thread Count (Dynamic Thread Configuration):

  Allowing tests to be run with varying thread counts

- Scalability Testing:

  To test scalability, various configurations with different numbers of threads were tested under different load conditions. The load was simulated using ApacheBench **(**ab**)**, which generated a fixed number of requests with varying levels of concurrency (-c option in ab). This allowed for performance evaluation in real-world-like conditions and helped identify the optimal thread count for handling specific levels of traffic.

Task Scheduling:

If response is same (for my Project it is simple web page with a response “Hello” then using FIFO can be good choice.

But if some requests more important i mean for example real time health checks then FIFO won’t prioritize them. But at this simple web server Project requests only needed to be responsed with same page content “Hello” .

Syncronization:

With using mutex and condition variables I guaranteed the synchronization.

I stored a queue for the incoming client sockets and multipke threads and one accept thread has to modify this queue.

So I used a mutex to ensure only one thread can modify the queue with usin queue_mutex. My solution prevents race conditions .

If there is no new client socket connection in the queue then in the worker thread it is blocked with pthread_cond_wait .

For accepting new connections at the accept thread cond_signal mechanism is used. I used it to wake the waiting worker thread and handles the request.

Implementation Details

The server was implemented using the C programming language.

The pthreads library (POSIX threads) was used for managing threads within the server.

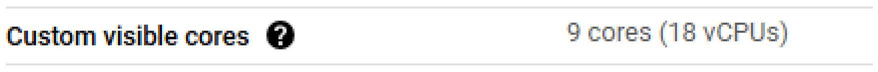
As an extra for standard C libraries networking libraries are used at this project:

- sys/socket.h : created sockets for communication between the server and clients.
- netinet/in.h : used for working with Internet Protocols (e.g., inet_addr, htons) to configure the server's socket and bind it to a port.
- arpa/inet.h: Provides functions to convert IP addresses into a format that can be used with sockets.

ApacheBench (ab) is used as a testing tool for analyzing multiple requests to test how well the server performs under load.

Hardware Setup

The server was tested on a machine with an 9-core CPU to ensure that multiple threads could run concurrently.



Custom visible cores 9 cores (18 vCPUs)

The remote machine system had 32 GB of RAM providing enough resources for handling multiple threads without running into memory limitations.

The server was deployed on a Linux-based OS (at Debian), which provides native support for pthreads and is well-suited for handling multithreaded applications.

Challenges

Of course parallelising a problem is a challenge. Ensuring about the thread synchronization for worker threads to access to the shared queue was a first challenge.

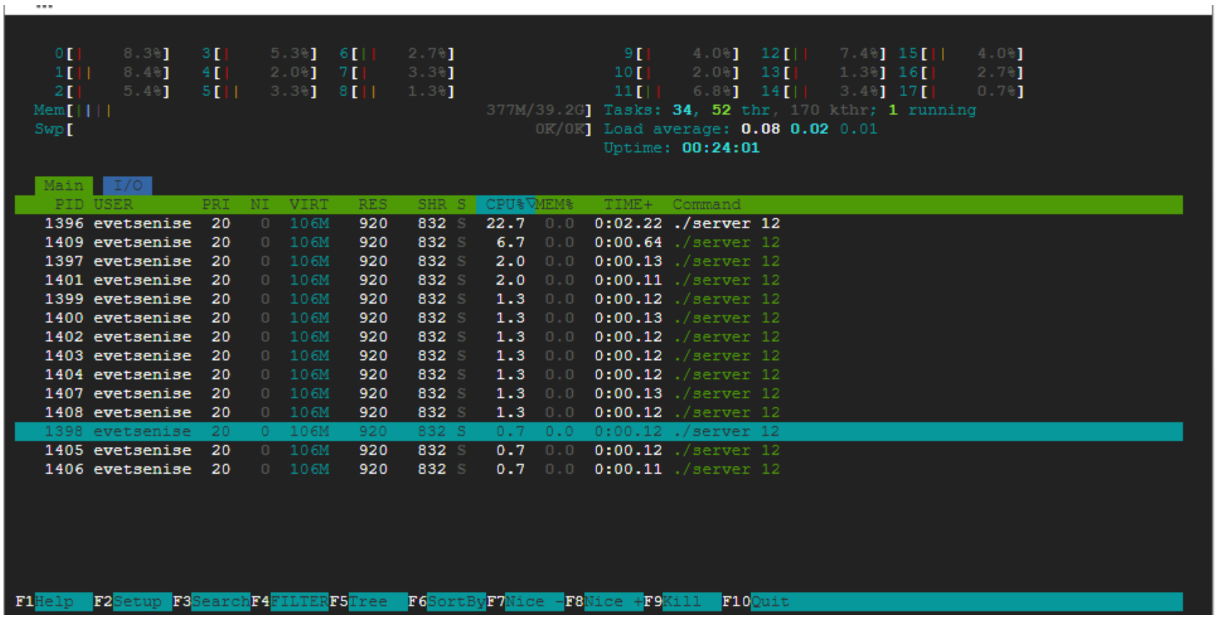
Choosing the best performance tools was the second challenge for me.

Analysis

Metrics

- Time taken for tests
- Requests per second (Throughput)
- Time per request (Latency) : average time taken by the server to process a single request.
- Transfer rate : how much data the server was able to send/receive per second during the test
- Connection Times (connect, waiting, processing)
- CPU and memory usage

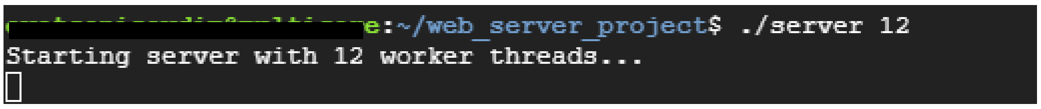
Experimental Results



```
    0[|     8.3%]   3[|     5.3%]  6[||    2.7%]                9[|     4.0%]  12[||    7.4%] 15[||    4.0%]
    1[||    8.4%]   4[|     2.0%]  7[|     3.3%]               10[|     2.0%]  13[|     1.3%] 16[|     2.7%]
    2[|     5.4%]   5[||    3.3%]  8[||    1.3%]               11[||    6.8%]  14[||    3.4%] 17[|     0.7%]
  Mem[||||                                       377M/39.2G] Tasks: 34, 52 thr, 170 kthr; 1 running
  Swp[                                                0K/0K] Load average: 0.08 0.02 0.01
                                                             Uptime: 00:24:01

  Main  I/O
    PID USER        PRI  NI  VIRT   RES   SHR S  CPU%▽MEM%   TIME+  Command
   1396 evetsenise   20   0  106M   920   832 S  22.7  0.0  0:02.22 ./server 12
   1409 evetsenise   20   0  106M   920   832 S   6.7  0.0  0:00.64 ./server 12
   1397 evetsenise   20   0  106M   920   832 S   2.0  0.0  0:00.13 ./server 12
   1401 evetsenise   20   0  106M   920   832 S   2.0  0.0  0:00.11 ./server 12
   1399 evetsenise   20   0  106M   920   832 S   1.3  0.0  0:00.12 ./server 12
   1400 evetsenise   20   0  106M   920   832 S   1.3  0.0  0:00.13 ./server 12
   1402 evetsenise   20   0  106M   920   832 S   1.3  0.0  0:00.12 ./server 12
   1403 evetsenise   20   0  106M   920   832 S   1.3  0.0  0:00.12 ./server 12
   1404 evetsenise   20   0  106M   920   832 S   1.3  0.0  0:00.12 ./server 12
   1407 evetsenise   20   0  106M   920   832 S   1.3  0.0  0:00.13 ./server 12
   1408 evetsenise   20   0  106M   920   832 S   1.3  0.0  0:00.12 ./server 12
   1398 evetsenise   20   0  106M   920   832 S   0.7  0.0  0:00.12 ./server 12
   1405 evetsenise   20   0  106M   920   832 S   0.7  0.0  0:00.12 ./server 12
   1406 evetsenise   20   0  106M   920   832 S   0.7  0.0  0:00.11 ./server 12

F1Help  F2Setup F3SearchF4FILTERF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

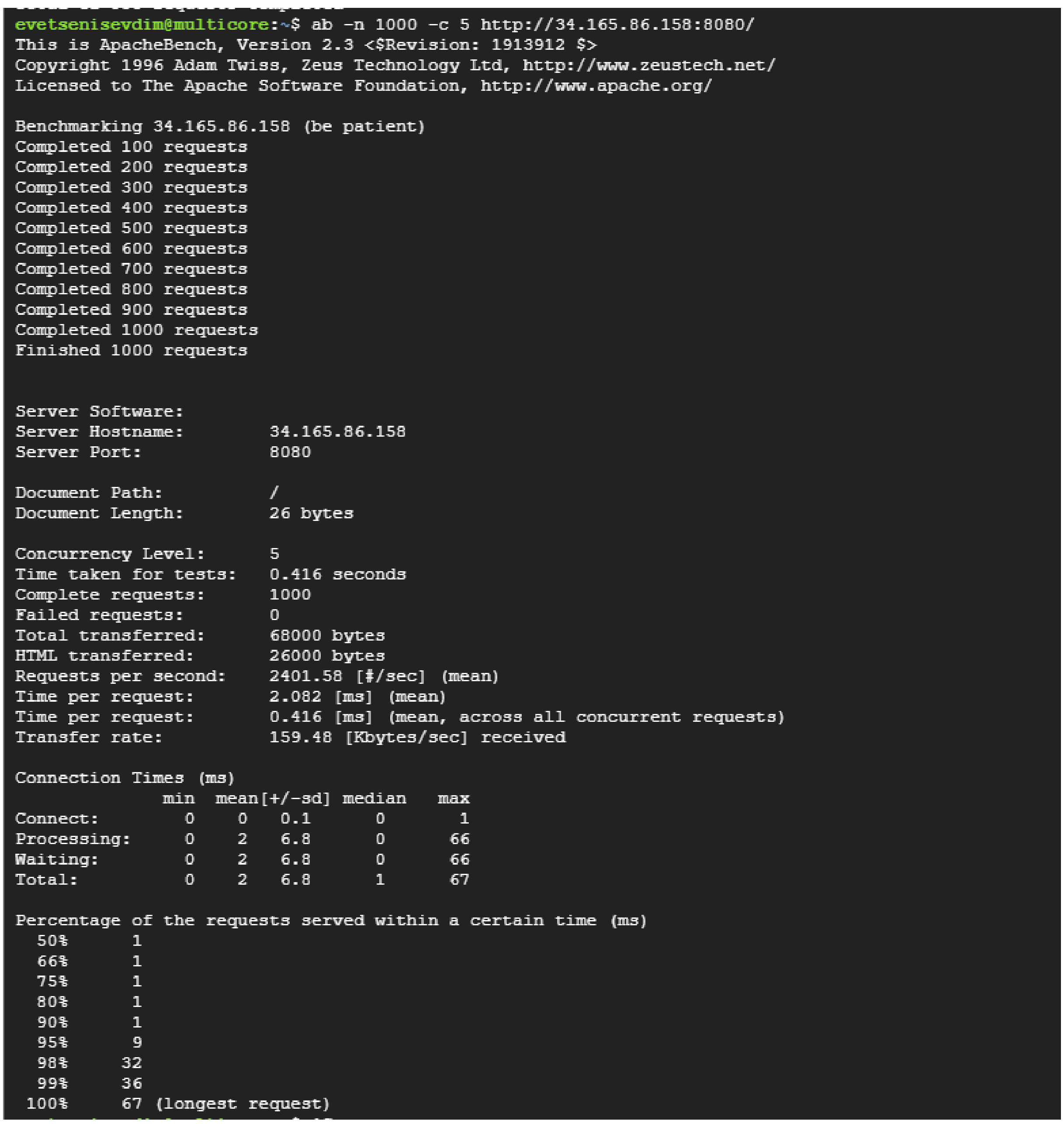
Figure 1: An Example of CPU Usage and Time (htop)

For the 20 thread count sending 100 requests and 50 concurrent requests:



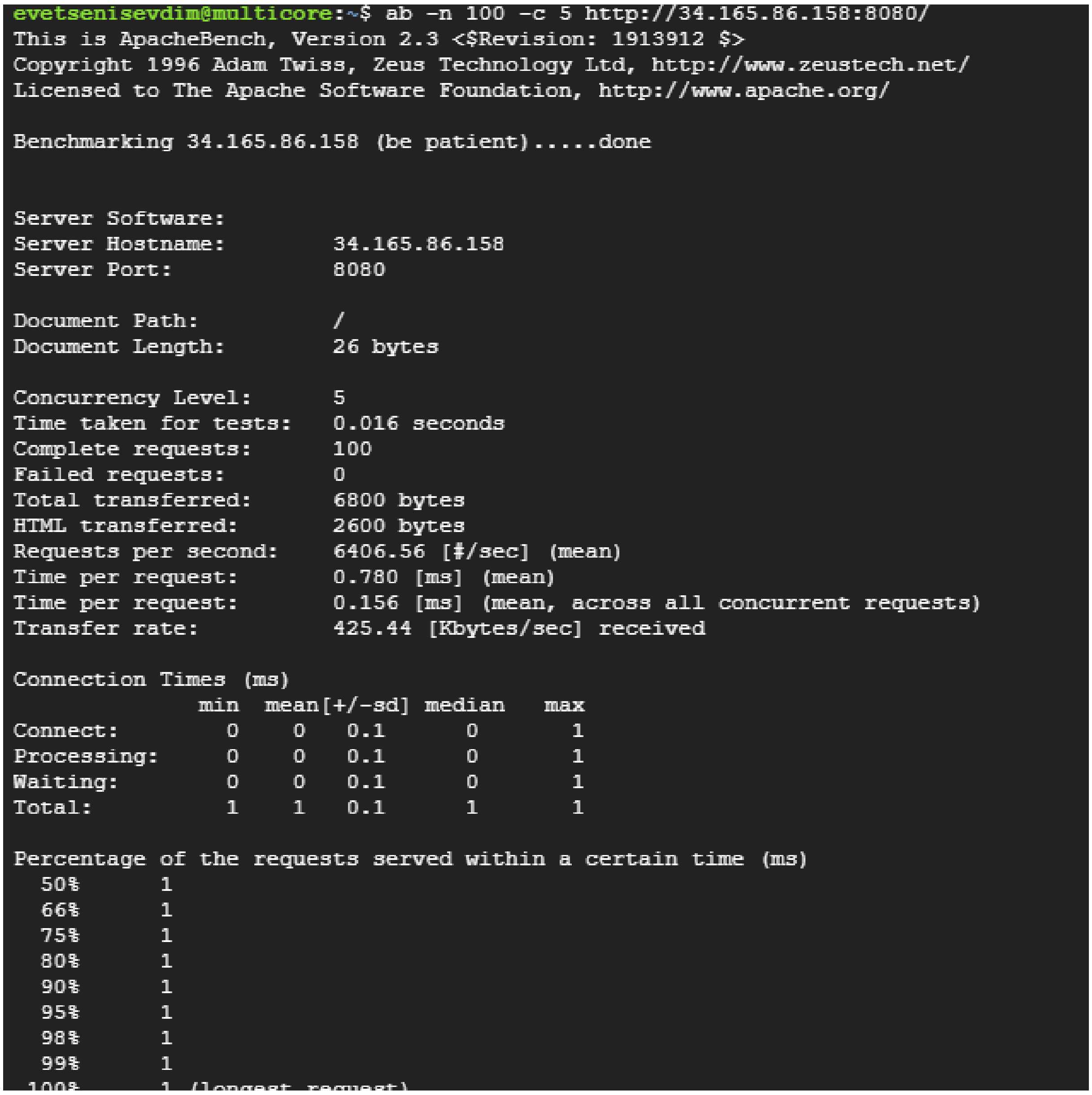
```
[redacted]e:~/web_server_project$ ./server 12
Starting server with 12 worker threads...
```

Figure 2: Server with 12 thread count



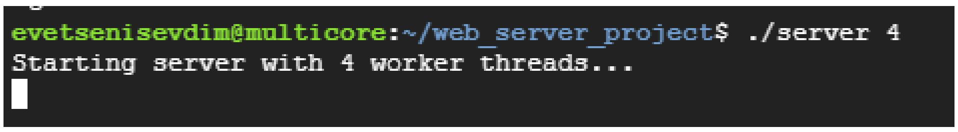
```
evetsenisevdim@multicore:~$ ab -n 1000 -c 5 http://34.165.86.158:8080/
This is ApacheBench, Version 2.3 <$Revision: 1913912 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 34.165.86.158 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:
Server Hostname:        34.165.86.158
Server Port:            8080

Document Path:          /
Document Length:        26 bytes

Concurrency Level:      5
Time taken for tests:   0.416 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      68000 bytes
HTML transferred:       26000 bytes
Requests per second:    2401.58 [#/sec] (mean)
Time per request:       2.082 [ms] (mean)
Time per request:       0.416 [ms] (mean, across all concurrent requests)
Transfer rate:          159.48 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       1
Processing:     0    2   6.8      0      66
Waiting:        0    2   6.8      0      66
Total:          0    2   6.8      1      67

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      1
  90%      1
  95%      9
  98%     32
  99%     36
 100%     67 (longest request)
```

Figure 3: 1000 client requests with 5 concurrent level



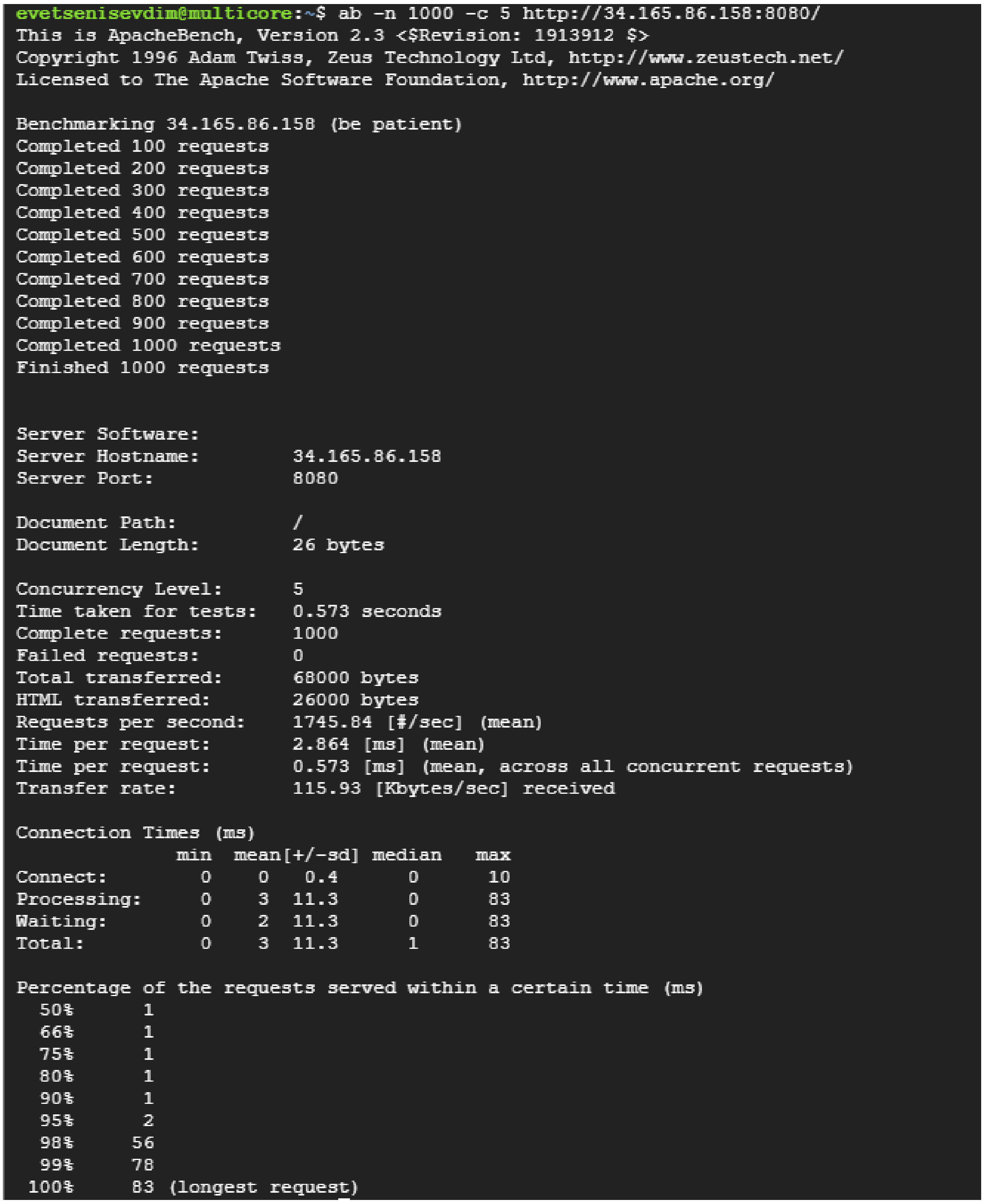
```
evetsenisevdim@multicore:~$ ab -n 100 -c 5 http://34.165.86.158:8080/
This is ApacheBench, Version 2.3 <$Revision: 1913912 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 34.165.86.158 (be patient).....done


Server Software:
Server Hostname:        34.165.86.158
Server Port:            8080

Document Path:          /
Document Length:        26 bytes

Concurrency Level:      5
Time taken for tests:   0.016 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      6800 bytes
HTML transferred:       2600 bytes
Requests per second:    6406.56 [#/sec] (mean)
Time per request:       0.780 [ms] (mean)
Time per request:       0.156 [ms] (mean, across all concurrent requests)
Transfer rate:          425.44 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       1
Processing:     0    0   0.1      0       1
Waiting:        0    0   0.1      0       1
Total:          1    1   0.1      1       1

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      1
  90%      1
  95%      1
  98%      1
  99%      1
 100%      1 (longest request)
```

Figure 4: 100 client requests with 5 concurrent level



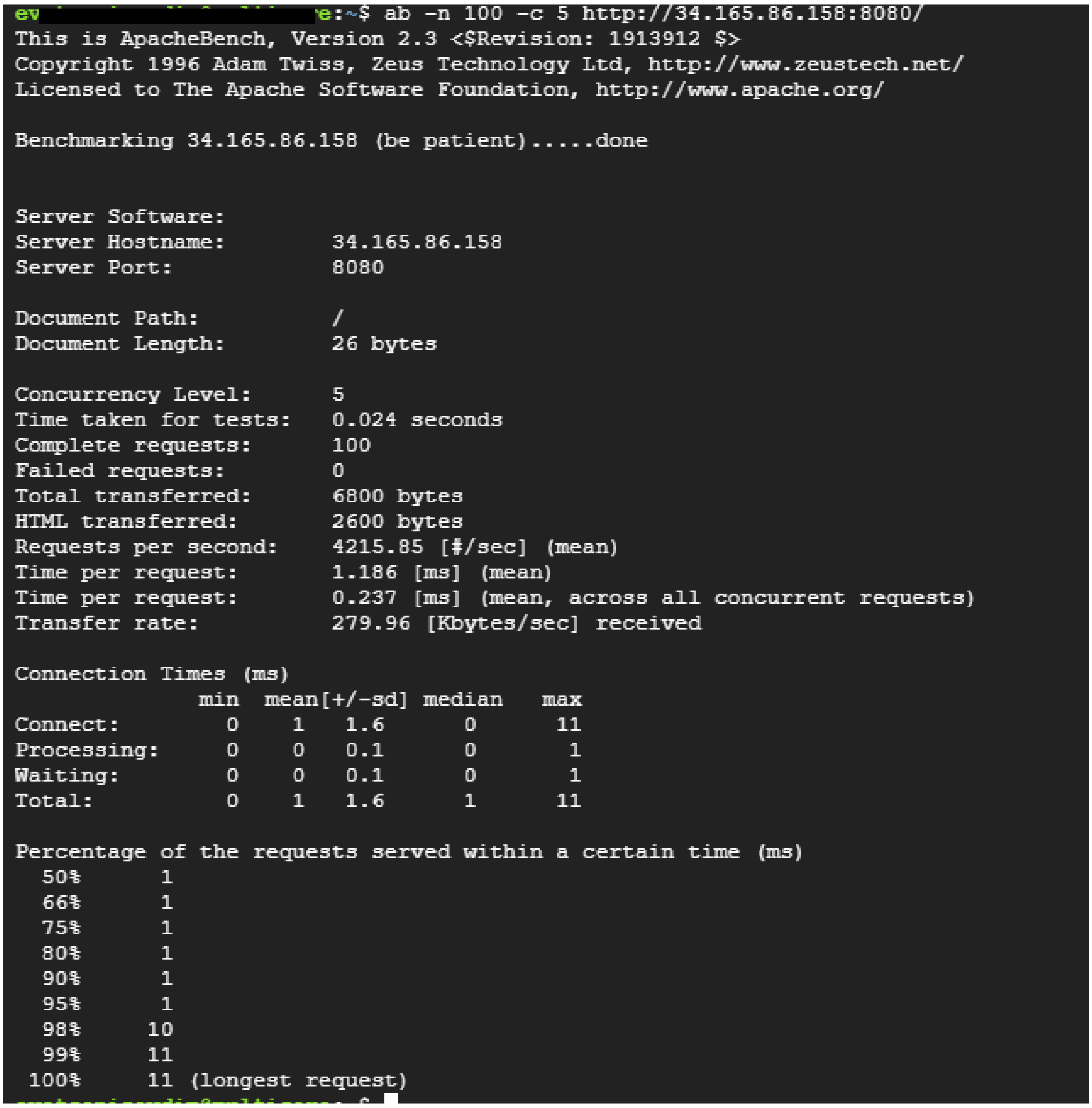
```
evetsenisevdim@multicore:~/web_server_project$ ./server 4
Starting server with 4 worker threads...
```

Figure 5: Server with 4 thread count



```
evetsenisevdim@multicore:~$ ab -n 1000 -c 5 http://34.165.86.158:8080/
This is ApacheBench, Version 2.3 <$Revision: 1913912 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 34.165.86.158 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:
Server Hostname:        34.165.86.158
Server Port:            8080

Document Path:          /
Document Length:        26 bytes

Concurrency Level:      5
Time taken for tests:   0.573 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      68000 bytes
HTML transferred:       26000 bytes
Requests per second:    1745.84 [#/sec] (mean)
Time per request:       2.864 [ms] (mean)
Time per request:       0.573 [ms] (mean, across all concurrent requests)
Transfer rate:          115.93 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.4      0      10
Processing:     0    3  11.3      0      83
Waiting:        0    2  11.3      0      83
Total:          0    3  11.3      1      83

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      1
  90%      1
  95%      2
  98%     56
  99%     78
 100%     83 (longest request)
```

Figure 6: 1000 client requests with 5 concurrent level



```
ev[redacted]e:~$ ab -n 100 -c 5 http://34.165.86.158:8080/
This is ApacheBench, Version 2.3 <$Revision: 1913912 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 34.165.86.158 (be patient).....done


Server Software:
Server Hostname:        34.165.86.158
Server Port:            8080

Document Path:          /
Document Length:        26 bytes

Concurrency Level:      5
Time taken for tests:   0.024 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      6800 bytes
HTML transferred:       2600 bytes
Requests per second:    4215.85 [#/sec] (mean)
Time per request:       1.186 [ms] (mean)
Time per request:       0.237 [ms] (mean, across all concurrent requests)
Transfer rate:          279.96 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.6      0      11
Processing:     0    0   0.1      0       1
Waiting:        0    0   0.1      0       1
Total:          0    1   1.6      1      11

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      1
  90%      1
  95%      1
  98%     10
  99%     11
 100%     11 (longest request)
```

Figure 7: 100 client requests with 5 concurrent level

Discussion

Increasing the thread count reduced the total time taken to handle the same number of multiple requests.

Higher thread counts also resulted in decreased latency for the same number of multiple requests.

Increasing the concurrency level in ApacheBench led to higher time taken and increased latency.

As the thread count increased, the server's throughput improved as well.

These are the main results from tests.

Conclusion

The test results demonstrated that the desired outcomes were achieved by increasing the thread count.

It significantly improved throughput and reduced latency for the same number of requests.

Higher concurrency levels in ApacheBench increased both total time taken and latency.

The optimized server design also showed improved scalabilit.

For future work, Adding support for advanced scheduling policies like priority-based task scheduling could further enhance performance.