Systems Programming Final Project Report

Student name: Vakhid Betrakhmadov
Student number: 141044086

In this project, we were supposed to implement client-server based application with two different server strategies (thread-per-request and worker pool) and analyze the performance differences. The application is supposed to be of the following form. Server program is supposed to simultaneously handle connections from multiple clients, that is for each client generate random matrices A and b for the linear equation of the form Ax = b (P1), simultaneously solve this equation with 3 different methods (svd decomposition, qr decomposition and pseudo-inverse) using generated A and b matrices (P2), find the norms of the error terms for the generated solutions (P3), send all results to the requesting client and log result on the server side. Note that P1, P2 and P3 are also parallel. Clients program is supposed to generate q number of clients that race to connect to the server, that is send their ids and parameters of the matrices to the server, receive results from the server and log them to their own log files.

Usage:

1) Thread-per-request server implementation:
 In order to compile the whole system with thread-per-request server implementation just run **make** command in the folder with project.

Run either **./server** or **./server <port number>** in order to start the server program.

Run either **./clients <rows> <columns> <clients number> <hostname>** or **./clients <rows> <columns> <clients number> <hostname> <port number>** in order to start the clients program.

Press **CTRL+C** in order to terminate the server or clients program.

2) Worker pool server implementation:
In order to compile the whole system with worker pool server implementation, first, uncomment **TH_POOL = -DTH_POOL** line in the makefile,second, run **make** command in the folder with project.

Run either **./server <pool size>** or **./server <port number> <pool size>** in order to start the server.

Run either **./clients <rows> <columns> <clients number> <hostname>** or **./clients <rows> <columns> <clients number> <hostname> <port number>** in order to start the clients program.

Press **CTRL+C** in order to terminate the server or clients program.

Performance analyses:

All measurements were taken in terms of miliseconds, ‘rows’ and ‘columns’ parameters were chosen the same for all test cases. The results obtained from the test cases for both server implementations and different number of clients and ‘clients’ programs running simultaneously are listed in the tables bellow (merged cells represent multiple ‘clients’ programs running simultaneously) .

Thread-per-request server implementation test cases results:

Number of clients	Average connection time (milisec)	Standard deviation
5	0.09	0.11
10	0.15	0.16
20	0.12	0.16
50	0.10	0.15
60	0.33	0.38
70	58.05	236.85
100	372.65	499.16
5 5	0.08	0.07
	0.10	0.09
10 10	0.17	0.15
	0.16	0.19
20 20	0.10	0.12
	0.26	0.17
50 50	60.89	242.81
	768.37	436.087
100 100	1185.982	679.470
	1506.77	1346.23

Worker pool server implementation test cases results:

Number of workers	Number of clients	Average connection time (milisec)	Standard deviation
5	5	0.63	0.30
5	10	0.24	0.24
5	20	0.10	0.09
5	50	0.15	0.29
5	60	0.094	0.096
5	70	58.32	237.72
5	100	392.48	443.24
20	10	0.16	0.21
20	20	0.18	0.17
20	50	0.08	0.11
20	70	0.10	0.15
20	100	276.76	457.04

50	5	0.15	0.17
50	10	0.06	0.02
50	20	0.05	0.04
50	50	0.13	0.20
50	70	0.21	0.37
50	100	0.12	0.13
50	50	0.16	0.18
	50	0.23	0.37
50	100	552.50	501.81
	100	441.52	496.09
100	100	123.13	333.22
	100	358.13	488.40

As can be seen from the tables above, average connection time for the thread-per-request server implementation starts to rise tremendously after 60 clients requesting connection at the same time, and average connection time results almost match results obtained for the worker-pool server implementation with just 5 workers. After we increase number of workers in the worker-pool server implementation from 5 to 20, we can clearly observe average connection time dropping. Thus at number of 50 workers average connection time even at connection request burst of 100 clients stays considerably small. We then double connection requests and consequently average connection time increases again, we then double workers and observe average connection time decreasing.
From this observation we can make a conclusion that worker-pool based servers after some point perform much better then thread-per-request based servers, but pool size must be carefully balanced with expected number of simultaneous requests. Flexible implementations may dynamically adjust the number of threads in the pool to maintain system balance.
Such performance difference might be explained by the new threads creation overhead in the thread-per-request based servers, which after some particular point starts to affect the whole system run time. Also, performance of a worker-pool based server is critically tied to the size of the pool and does not differ too much from thread-per-request based server performance for a small number of workers and considerably small number of simultaneous connection requests.