Group 6 - Included Eagles

Helena Schiøtz, s174279
Niels With Mikkelsen, s174290
Lasse Pedersen, s174253
Sebastian Arcos Specht, s164394
Felix Larsen, s174296
Nikolaj Geertinger, s153140
Matthew Sinnott, s191989

Table of Contents

Initial Setup (Milestone Submission)

Implementation:

- Spawn a new process for every request, using fork( )
- Reading the request directly to a packet.
- Calculate the hash.
- Send answer back to client.

Threads

Motivation: Threads are more lightweight

Implementation:

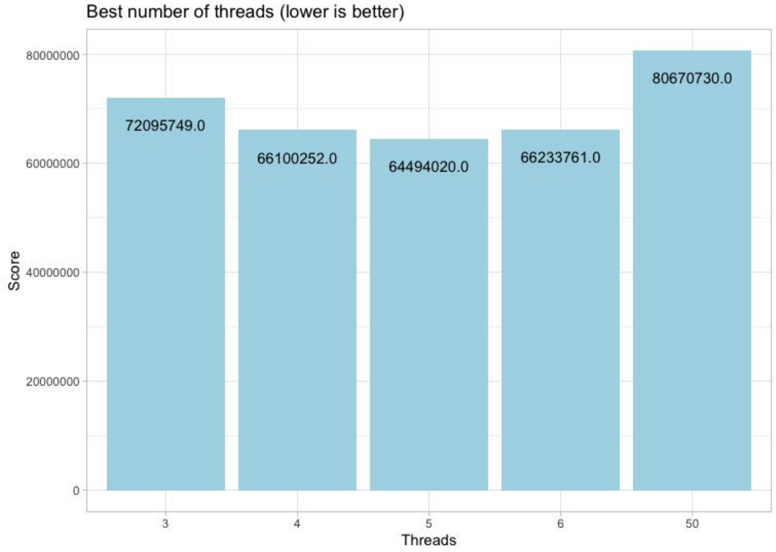
- Fixed number of threads (50 for this experiment)
- FIFO scheduling

Result: Noticeable speed boost, ~16.0% faster

Threads best number

Motivation: Avoiding Context switches

Result: Big speed boost with
5 threads, ~45.0% faster



Thread management

Run	Attempted improvemnt	Benchmark
First run	399,678,749	447,303,326
Second run	403,371,309	450,411,566
Third run	393,811,925	432,328,906
Fourth run	405,405,428	440,569,433
Fifth run	400,218,740	438,554,479
Mean	**400,497,230**	**441,833,542**

Motivation: Better utilisation of the CPU,
but more time wasted on managing threads

Implementation:

- Start new thread immediately upon exit
- Mutexes and shared memory to keep track of status for each thread

Testing: High difficulty

Result: ~10.3% faster

Optimized memcmp

Motivation: A lot of comparisons, Most comparisons are not equal, Inbuilt memcmp is CPU intensive

Implementation: Checking first four bytes before proceeding with inbuilt memcmp

Testing: Final Configuration, 100 requests

Result: Minor speed boost, ~6.0% faster

Manipulating Pthread nice values

Motivation: Tweaking inbuilt scheduler, High priority task -> Higher priority thread

Testing: Final Configuration, Altered priority lambda value, 100 requests

Result: No speed difference, ~0.3% faster

Linear vs Random search

Motivation: Can the Linear Search be improved by doing a random search?

Testing: Custom Client (due to bad computer):

- 1000 requests + 1000 difficulty
- 1000 requests + 100000 difficulty

Result: Same speed for 1000 difficulty, almost 10x slower for 100000 difficulty - Not Merged

Explanation: Worst case for linear search is $O(n)$, but for random search it is $O(\infty)$

Scheduling Based on a Cost Function

Motivation: Want to be “smarter” when scheduling packets

Tasks 1&2 $= 2t_1+2t_2$

Task 1 Task 2 $= t_1+2t_2$

Designed a cost function that takes into account both the size of the range as well as the priority of the request, and sorted the queued requests by this new cost.

Testing: All tests were done with the run-client-final.sh script
Result: Using the cost function improved the performance by 12.26% over the FIFO - merged
Notes: This change only makes a difference if there are more outstanding requests than the current number of available threads

Caching

Run	Seed	Without hash table	With hash table
First run	3435245	46.576.610	42.154.478
Second run	343524	58.094.685	43.041.318
Third run	34352	71.796.656	60.601.342
Median		**58.822.650**	**48.599.046**

Motivation: Can it improve the average response time to save the calculated hash values in memory?

Testing: 100 requests final-submission with and without implemented hash table

Results: **~17.4% faster**

Explanation: Implemented hash table to store calculated hash value. Optimized with different hash functions and different sizes of hash table.

Priority Queuing by Packet Priority

Motivation: Can we schedule the packets in order to compute the higher priorities first?

Implementation: Singly-linked list. 1 main thread, 3 workers (should probably have been 4). Main thread building the queue, and the 3 workers consuming the top of the queue.

Final Configuration

- Scheduling Threads with a Priority-based Cost Function
- Caching using Hashtable
- Multithreading