Making Minecraft Fast

Final Project Report

15-418, Fall 2019

Tony Lu, Julian Tutuncu-Macias

1 Summary

We improved the performance of Craft, a lightweight version of Minecraft, by using a quad-tree to parallelize the spatial filtering of player positions so as to reduce the overhead of unnecessary outgoing world updates to players.

2 Background

Minecraft is a popular sandbox game in which the world is composed of cube-shaped blocks. While the implementation of Minecraft is closed-source, Craft is a lightweight open-source implementation of an identical game design [1]. Craft supports few of Minecraft's many complex features beyond the same fundamental environment and mechanics, most notably the partitioning of the world into 32 by 32 block "chunks." While the majority of Craft itself is written in C, the multiplayer server is implemented in Python. Updates made to distinct chunks by multiple players interacting in a server are stored in an SQLite3 database as a delta against the initially generated world map. Modifications to the environment made in a player client are transmitted to the Craft server, where

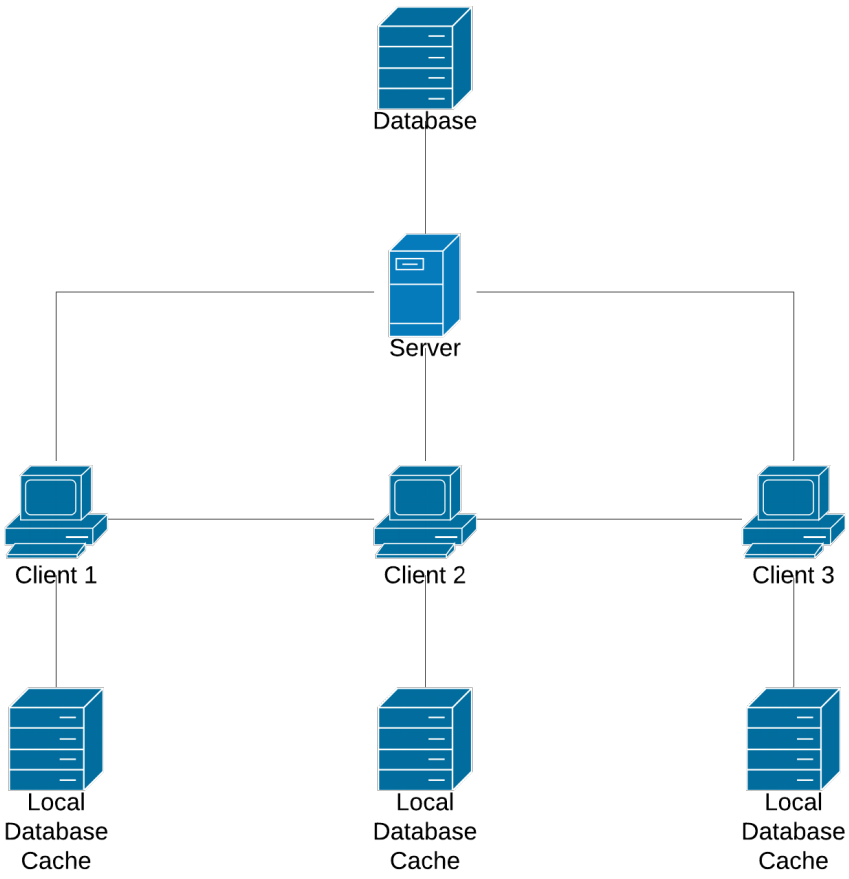

Figure 1: Server and database structure in multiplayer mode. There may be more than three clients.

the world database is updated with the modification.

In single player, the client edits the database directly, but in multiplayer, the client creates a smaller local copy of the database and stores changes there. It then requests chunks from the server's database if it doesn't have them, and uses a key based on update time to determine whether to write its changes to the server's database.

Craft also uses many queues to organize tasks. In single player and multiplayer, clients push user events, such as destroying blocks, onto a queue that is taken care of by a thread running database tasks. In multiplayer, the server employs many more queues. Client handlers each have a

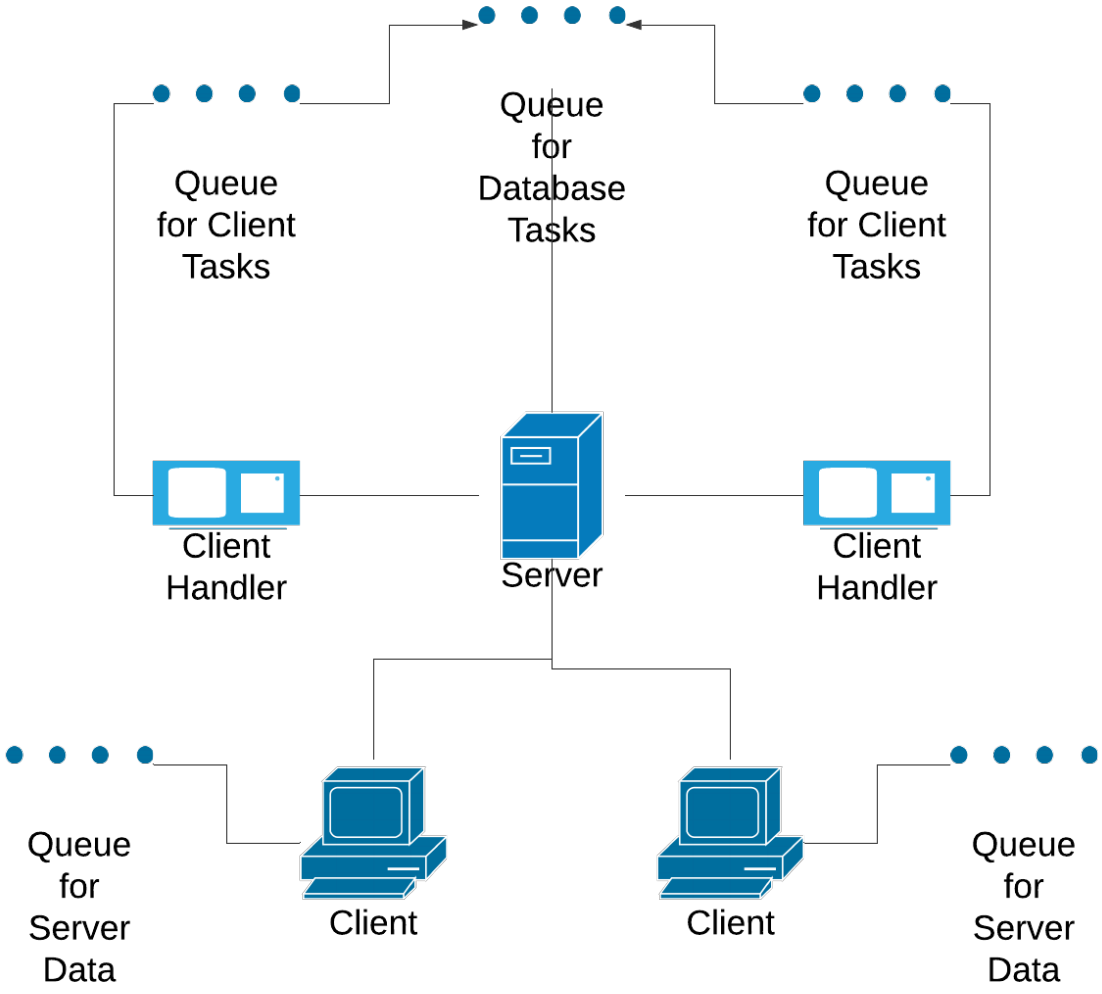

Figure 2: Usage of queues in multiplayer mode. There may be more than two client handlers and clients with their respective queues.

queue for handling tasks from their respective client. The server's database uses a queue for tasks that will edit the database, which are dequeued and parsed from the client handler queues.

To handle all this computation, Craft takes advantage of multithreading. In multiplayer, the server employs a thread for each client handler and one thread to handle edits to the database. The client employs multiple threads for loading chunks and one for receiving and queuing server data. The bulk of the computer's proessing resources are thus dedicated to loading chunks. This leaves only one thread to handle everything else, including rendering and handling server commands and user input. In single player, the client handler and server data threads are not used, and the database management thread is moved to the client.

In order to provide all clients with a synchronized world environment, the server needs to broadcast any updates to all the other connected clients to maintain synchronization. However, for the server to emit messages to every client for every update it receives is unnecessary. Clients have a configurable render distance between 1 and 24 chunks, past which the world is obscured with fog to enhance performance. Thus, when clients are sufficiently far apart from the source of the update, there is no need to relay changes made in chunks that are not currently loaded. In addition, the Craft server uses a single thread to send messages to clients. The presence of the Global Interpreter Lock in Python prevents parallel implementations from speeding up the code significantly and makes the `threading` module relatively ineffective beyond providing the queue-based task infrastructure to process these updates. By transitioning to the `multiprocessing` module, we can achieve true concurrency.

3 Approach

We began by forking Craft, a lightweight version of Minecraft written by Michael Fogleman. We started by devising predictive rendering schemes, although this fit more of a graphics theme than

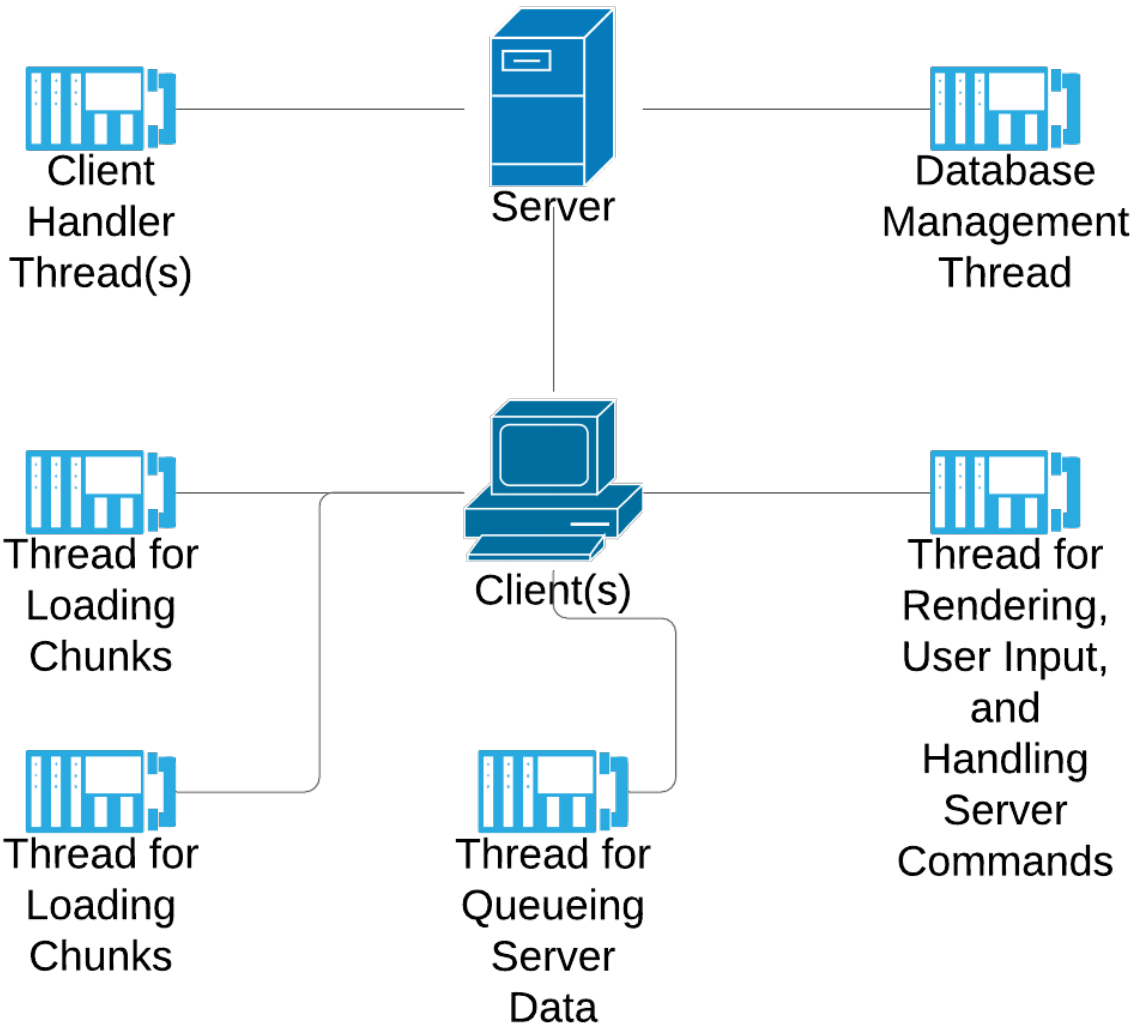

Figure 3: Usage of threads in multiplayer mode.

a parallel computing theme, so we pivoted to a different project. We then attempted to devise schemes to reduce the number of writes made to the database, since the current implementation uses one thread to handle database updates. We attempted to cache layers of database updates by chunk, and then commit those updates after the user had left the chunk. However, we found it difficult to devise a scheme that would let us remove locks around the cache layer, and we decided that database writes must be done sequentially. This resulted in us trying out our last approach, which was using the quad-tree to speedup broadcasting data to clients.

To efficiently determine all clients within a given radius of an update, we used the quad-tree. Quad-trees partition the world space and allow efficient queries for the clients within a given distance from a point. When a block is placed or a player changes their position, we can compute the set of clients within a given radius and send an update to only those "relevant" clients. When we received a position or block update from a client, we would rebuild the quad-tree, write the update to the database, query the quad-tree to determine all clients within a given radius, and then send the block update to those relevant clients.

In the `Model` class definition in `server.py`, the functions `send_positions`, `send_position`, `send_block`, `send_light` and `send_sign` all shared the serialized approach of iterating through the full list of clients connected to the server and sending an update to those clients. We updated these functions to send block updates to relevant clients in parallel using the quad tree. The threads performing the parallel send used a task queue, so every available thread could perform a send if there were enough clients to send to. We focused development and testing on our personal laptops with relatively modest consumer hardware to better benchmark our implementation, as the base implementation of Craft was already performant on said machines.

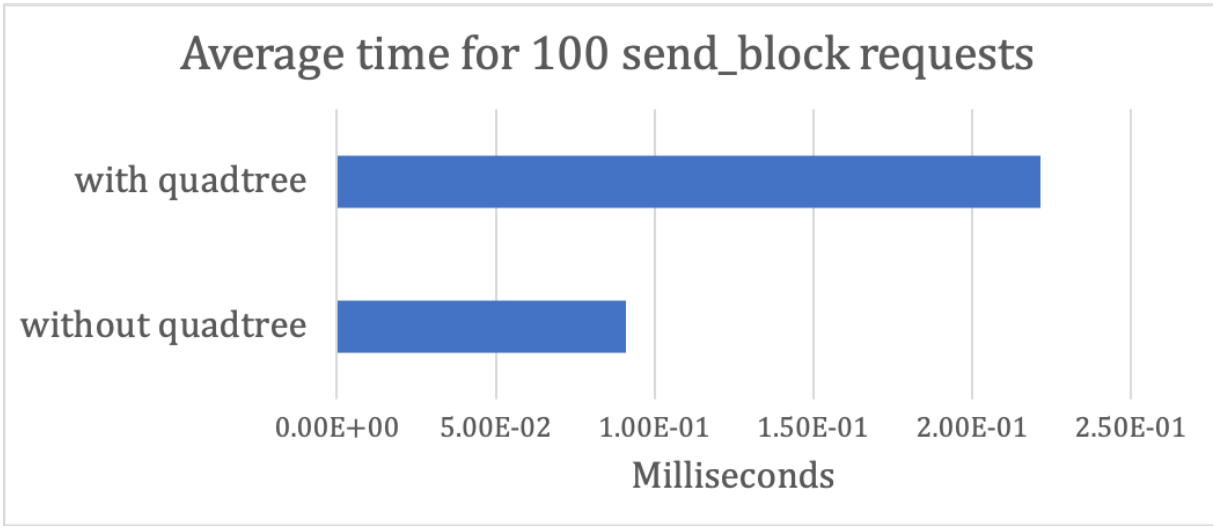

Figure 4: Average time for 100 `send_block` requests. Measured across two players who each made 50 requests.

4 Results

We did not achieve noticeable frame rate speedup with our quad-tree implementation. This is likely due to the fact that client sending and receiving is much less computationally intensive than other parts of the game pipeline, especially rendering and loading chunks. Even when we measured the time for broadcasting block updates alone, it did not achieve speedup. This is because we did not test on a server with enough players for it to have a nontrivial effect. Additionally, we ran the server and one of the clients on the same computer. This likely occupied all the threads on the machine, leaving little processing power for the parallel client sending.

It would have been a more rigorous test of our algorithm's scalability to host the server instance on a machine with a more powerful CPU, specifically one with more cores to more effectively parallelize the work of building the quad-tree upon each call. Since there is little computation involved, a machine with many CPU cores is more suitable than a GPU for this task. Ideally, we would have liked to measure speedup on a server with many players, since we speculate that our

algorithm will perform best in this scenario. The overhead of doing a linear search on all the clients to determine which are in the radius would then be higher than the overhead of constructing the quad-tree and querying it.

5 Distribution of total credit

50-50

References

[1] M. Fogleman. Craft, 2013. https://www.michaelfogleman.com/projects/craft/.