15418 HW4

Kathleen Fuh and Shreya Vemuri
kfuh@andrew.cmu.edu, shreyav@andrew.cmu.edu
March 25, 2016

Data Structures

In our web server, we started by fixing the problem of handling multiple requests by adding queues to the worker side. Below we discuss the various data structures we used throughout the process of optimizing our web server.

Structs:

- Within **MASTER:** We have a struct for the master state and a struct for each of the worker states. The worker states are kept in an array.

The `Master_state` struct holds information regarding the state of the master. Beyond the given starter code, we added variables `num_alive_workers` to indicate the number of workers booted and and being used at the time, and `num_to_be_killed` which indicates how many workers are to be killed (since once we indicate a worker should be killed we have to allow it to finish processing all its jobs). We explain this concept below when we talk about workers. These counts help us with our policies and with keeping track of the state of our workers at any point of time.

We additionally have 2 data structures within this struct. One is an unordered map that maps each tag to a `client_handle`, so that we know which client to send the response back to once we get a response. Once we receive a response, we remove the tag from the map. This way, our map doesn't grow to be too big. The second data structure is an array of of 4 worker states, so that we can keep track of the information for each worker. We next talk about our `Worker_state`. We never have more than `max_num_workers` worker nodes booted at any point in time. The handout states that this number will be max worker nodes = 4 for this assignment.

For our worker nodes, the `Worker_state` struct holds information about each of the worker nodes. We have a boolean variable `is_alive` indicating whether or not the worker is alive or not, a boolean variable `to_be_killed` which is a flag that helps us determine if a worker node needs to be killed because it is not doing much work. This flag helps us know not to add additional work onto this worker node if it to be killed soon. If we added work to a worker that needs to be killed, we would never be able to kill it because we must wait for all requests on a node to be processed before we get rid of the node. These flags additionally help us keep track of the number of workers in a particular state in our `Master_state`

We additionally have statistics indicating how many of each type of request we have. CPU intense requests include `wisdom` and `countprimes`. A cache intense request is `projectidea`. A non-intense request is `tellmenow`. We categorized these requests based on the descriptions given in the handout and based on what we observed. We use these numbers to help us schedule work on the worker nodes. We make sure that we add a request to the worker that has the least amount of that type of work. We tried writing individual traces to also calculate latencies of individual request types to gauge and check our understanding of how these requests were working.

We finally include the `worker_handle` for that worker in our struct which is used a unique identificaiton for that particular worker handle.

Caching: We decided to cache all of our requests except for the `compareprimes` requests which we're doing well on without caching due to parallism we added which is described in "Parallelism" section below. Our cache was implemented using an unorderedmap type in C++. We mapped from the request string to the response message object, so that before we send a request to the worker, we could check the cache and not do unnecessary work. When we get back a request from the worker, we add it our cache.

Other Maps for Request Information:

We make use of unordered maps in several situations. Two of them are for when we parallelize `compareprimes` which is described in the section about Parallism below.

Another map, maps each tag number to a request string. Conveniently, we make it such that each request and response have the same tag. We delete the key-value pair after we get the response back and send the response back to the client.

We additionally map each request tag to a string indicating the type of the request. We defined these types to be “cache," “cpu," and “non." Those types were described above when we talked about the statistics we included for each of our workers.

Whenever possible, we delete entries from our maps as we get responses back to make sure our maps do not grow to be too large.

- Within **WORKER:** we have worker queues which we talk about below:

Worker Queues: We optimized to have 3 different queues: one for `tellmenow` requests, one for `projectidea` requests, and one for all other requests. The reason we did this was that we wanted each of the requests to be processed immediately and not be held back by the other requests. Specifically, `tellmenow` requests are low in CPU intensity and also need to be processed within the strict 150ms latency requirement. `projectidea` requests also need to be processed quickly, and putting those requests in with other requests, would make the latency worse. `projectidea` is cache intensive and takes up 14MB working set of the L3 cache, which only has 15MB of space. Thus, we only want one `projectidea` request to run at a time within a worker node because we do not know on which of the two cores within a machine it will be scheduled, and we do not want two of them on one core which may lead to false sharing and lead to evictions. With 48 threads, we cannot specifically tell the OS which core within a worker node we want to schedule the `projectidea` request. We additionally do not have any locks in our queue because the given Worker Queue implementation is thread-safe. What's also nice about the Worker Queue (thanks for providing it) is that the threads can sleep if the queue is empty instead of spinning which would waste processor resources.

This layout allowed us to prioritize easily and run things as quickly as possible.

Parallelism

We now turn to talk about some of the axes of parallelism we took advantage of, which were along requests and within `compareprimes`.

pthreads and worker nodes: Within each worker node, we spawn 47 threads (so we have 48 threads in total including the main) as was recommended on Piazza. The reasoning for this was that some of the threads may be serving the OS, so we wanted to spawn enough. We assign one thread to the `tellmenow` queue, one to the `projectidea` queue, and the rest to the other requests queue. `tellmenow` requests have a small latency, so one thread is enough. Additionally, as explained above, we only want one project ideas running at a time.

We additionally launch up to a maximum of `max_num_worker` worker nodes as it's passed into the master's initializition function. We currently never have more than 4 worker nodes.

Parallelizing Compare Primes: In order to make `compareprimes` requests run more efficiently, we broke the request up into the individual 4 `countprimes` requests and sent them as individual requests. We had to make sure that we knew which 4 `countprimes` requests belonged together and which request was which parameter as this was important to the calculation. In order to do this, we have two maps. When a `compareprimes` request came in, we assigned it an overall tag x. Then for each individual `countprimes` request for `n1`, `n2`, `n3`, `n3`, we assigned tags x, $x+1$, $x+2$, and $x+3$ respectively (this way of assigning tags allowed us to preserve the order of the parameters). The first map mapped the tag number of each parameter's `countprimes` request to the tag number of the entire `compareprimes` request. The second map mapped the tag number of the `compareprimes` request to a struct holding the `counts` array for the results as they come back and a counter telling us when we have gotten all of the 4 required responses. The first map helps us determine which index the parameter belongs to (a simple subtraction of the `countprimes` tag and the tag of the `compareprimes` request it belongs too - ie tag 62 for n3 - tag 60 overall = index 2). The second map tells us when we receive all necessary information back and stores the information to create the response for `compareprimes`.

Tick Time: We set the tick time to be 4 seconds. We did not want this number to be too short or too long relative to how long it takes to boot up a worker. We also wanted this number to be good enough such that we checked often enough when we need to add or delete workers. We chose this number after running multiple jobs and looking at the debugging messages we produced (showing us adding/ deleting of worker nodes and statistics) which gave us an idea about what our program was doing at each `handle_tick` and it seemed the best time to check the state of our server was 4 seconds to avoid too large of request buildups on individual nodes.

Policies Implemented

Elasticity Policy: In order to determine the right amount of worker nodes that we wanted to be running at a certain time, we implemented a policy to determine when we wanted to add and delete worker nodes. For each worker node we had a boolean `to_be_killed` flag. When a threshold (who's selection we'll discuss below) is reached, we mark the node as `to_be_killed` and we stop adding work to it. Once all its remaining jobs have completed, we kill the worker. Similarly, we also have a threshold for adding workers. If a node is marked as `to_be_killed` but has not been killed yet, we just set the flag to false again and it can be assigned work again. Otherwise, if we are under the `max_num_workers` value, we request a new onde. To determine if a machine was too loaded, we looked at the number of cache intensive and cpu intensive jobs.

- **Adding:** When the number of average cpu intense requests across the workers is greater than

$\frac{3}{4}$ the `NUM_THREADS` or we have more than an average of 1 cache intensive requests on a worker, we add a worker if we don't have more than 4 workers. We chose those thresholds because each thread picks up a request so using the number of threads in our policy is important and useful. We wanted to use $\frac{3}{4}$ because at that point, the worker is getting close to becoming overloaded and we want to get another node going before the worker gets overloaded, at which point it would be too late. Additionally, we do not want more than one cache intensive request per worker node. Using the average is okay here because our worker nodes are pretty balanced from our worker choosing policy (always selecting the worker with the minimum of that type of job); thus the average is representative of each worker node.

- **Deleting:** We delete a worker node when we have more than 1 worker node alive and when either of the following two conditions are met. (Please note, when we calculated the average work per node, we calculate it for one less worker to see what the average load would be if we were to delete a worker node, since we want to know if all the jobs can be well-distributed with fewer worker nodes.) Condition 1) there is an average cpu load of less than `NUM_THREADS` - 2 (2 because 1 thread is dedicated to projectidea requests and one to tellmenow requests). Condition 2) the average cache intensive load per worker node exceeds 1.

We also note that we do not check if we want to delete a worker node if we have just added a worker because having added a worker means that we were unable to efficiently schedule all jobs on the number of nodes we had at that point.

Find Best Worker Policy: We have another policy to determine which worker we want to send a request to. As mentioned above, we categorize each type of request as either cache intensive, cpu intensive, or non intensive. Thus, when we get a request that has not been cached, we send it to the worker with the least load of that type of request. In this way, we ensure that each worker does the same amount of work. We cannot weigh each request equally which is why we categorized each one (the latencies of each request is more important than the number of requests coming in).

Other Optimization Ideas: We also weighted each `countprimes` requests on a scale based on what its `n` a parameter was. We looked a the code for this request and read in the handout that this request was dependent on the input. We analyzed the traces to decide this. We additionally ran individual traces we wrote to see the differences in latency times based on changes in `n`. The weight of each the requests was its `n` itself to keep the scale as linear as possible.

These weights help us determine what worker to put a `countprimes` request on because we can use the total `countprimes` weight on a worker that we keep track of to find the worker with the min of this type of this request.

While the performance improvement was marginal from this optimization, we thought it best reflected the latencies of these types of requests.

To help us do this, we mapped a tag to its `n` value for easy lookup and for us to be able to decrement our weight counter for each worker when we came back from the worker with a response.

High Level Flow of Request to Response

Here, we give a high level picture of how all the cool things we discussed above work together to make our web server!

When the master gets a request, we first check the cache to see if we send a response right away. If not, we proceed to handle our request. If the request is a `compareprimes` request, we split it up into 4 countprimes requests. We additionally case on the other requests and update our counters within our statistics bookkeeping appropriately in addition to inserting into appropriate maps to keep things consistent and to help us throughout. We then choose the best worker to send our request to using the min work policy.

On the worker, threads assigned to a specific queue with a certain type of requests process requests and send the response back to the master. The remaining threads pull off of a shared work queue.

When the master get the response back, we check if the response is from a countprimes requests that belongs to a `compareprimes` request. If so, we make sure we get all 4 requests back and then do the logic for `comparepimes` before sending the correct response to the client. Otherwise we immediately send the responses back to our client. After this, we make sure to decrement any important counters to keep consistency, and delete any unnecessary data from maps. We finally check to see if the worker node was assigned the `to_be_killed` flag. If so and if its requests have reached 0, we kill it.

While all of this is happening, every 4 seconds, we call `handle_tick` which adds and deletes workers according to our policies described above to make sure that we do not have idle workers or workers that are overwhelmed.