15-418 Parallel

Yiming Wu, Haibin Lin

Mar. 2016

1 Request Execution

1.1 Request management overview

Each worker node runs multiple concurrent worker threads to serve requests sent from master.

Requests are managed by several queues, from which worker threads fetch the next request to do. There're three queues: `tell_me_now_queue` contains the tell_me_now requests sent to this worker node, `project_idea_queue` contains the project_idea requests, `queue` contains all other kinds of requests. In our implementation, we used the `WorkerQueue` class to ensure safe concurrent operation.

Each thread in worker node is dedicated to serve the requests from one of the three queues. The current configuration, we have 2 threads serving project idea requests, 1 thread serving tell me now requests, 44 threads serving other requests. That's in total 48 threads.

1.2 Compare prime request execution

The original implementation of compare prime execution involves 4 serial `execute_work` calls. This is parallelized by splitting 1 compare prime requests into 4 sub-requests sent to different worker nodes, where each sub-request involves 1 `execute_work` call. Master node is responsible for gathering sub-request results and return the final response to client. Tag is used in master to identify sub-requests which corresponds to the same compare prime request.

1.3 Project idea request execution

The execution of a project idea request involves a large working set of size close to size of L3 cache. Given the hardware with 2 L3 cache, the most ideal assignment is 2 project idea requests per worker node, each running on a L3 cache. The master ensures each worker node serve no more than 2 project idea requests so that the workers don't suffer from cache conflict and cache thrashing.

1.4 Tell me now request execution

Tell me now requests have strict latency requirement. We have dedicated threads to serve this `tell_me_now` requests. This makes sure tell me now requests are not queue'd after other latency-insensitive requests.

1.5 Caching

We noticed that many repeated `count_prime` requests are tested. Such request is deterministic and computationally expensive, a cache for such request will be great to boost performance. In current implementation, `count_prime` are cache'd in master node.

2 Work Assignment

The work assignment policy we are using right now is assign the job to the least loaded worker except `project idea` jobs. For `project idea` job, the master node will assign it to the worker node with least `project idea` jobs.

3 Elastic Policy

In this section, we will introduce our Elastic Policy. In subsection 3.1, we will introduce our final version policy, which get 11/12 on nonuniform2 and full score on all others. It is based on the ratio of request number over capacity on worker nodes. At the same time, it is a general policy without any special design for traces. In subsection 3.2, we will introduce some other policy we tried and discuss why they do not perform well on this assignment.

3.1 Policy based on request number

Our submission version policy is based on the ratio of request number over capacity on worker nodes, a.k.a request ratio. Here we use the total number of request and total capacity that is 48 request per node times node number. We have an upper bound of 0.7 and lower bound of 0.4. Each time `handle_tick` is called, our master node will check whether the on going request ratio is out of range.

3.1.1 Policy to add nodes

There are two situations when we add a worker node. First one is when the request ratio exceeds the upper bound. This means that there are going to be too many requests on a node. It is time to add nodes to relieve the tense. To go deeper, this design is based on the observation that there is usually a peak of request coming in the future when the requests start to come more frequently.

The other one is when number of `project idea` tasks on any one worker node reaches 2. Here, we choose number 2 because CPU is 2-core with only 2 L3 caches. Apparently we donot want there are more than 2 `project idea` running on one worker node. On the other hand, we decide to add nodes once any one reaches 2 because it is reasonable to assume request comes in a peak.

3.1.2 Aggresive Adding

We use a policy called aggresive adding when adding nodes, where we add as many nodes as we can once the master decide to add nodes. This is also based on the observation that request always comes in a high peak rate in the future once it becomes more frequent. With this policy, our master node gets more elasticity when adding nodes. This ensures our performance.

3.1.3 Policy to kill nodes

When the request ratio is beneath the lower bound, we will find a node with least jobs and descide not to send new request to it, a.k.a try to shut it down. We also have to make sure that the node we decide to kill in the future does not have `project idea` running. Because sometimes there are only `project idea` requests.

Every time `handle_tick` is called, master node will check whether there is node that we tried to shut down that has no request going on. If so, the master will send the `kill` request.

3.2 Other Policy

One previous policy we tried is based on the latency of a request to decide whether to add/kill nodes. For example, if we found 10% of the `project idea` takes more than 3800ms to finish(grading is on 4100ms), we know that it is time to add worker nodes. This seems a more plaussable solution but it has some drawbacks under this assignment's conditions. Overall, it gets 11/12 on nonuniform1 and 12-3(because of too much worker)/12 on nonuniform3 and full scores on all others. The policy is in function `handle_tick1`

One main concern is the unreliable latedays machine performance. Running same task twice on latedays at almost same time usually turns out different latencies, which makes our sensitive policy create more worker nodes that are unnecessary.

The other one is that, based on observation, the latency of requests jumps from low level, e.g. 1600ms for `countprimes`, to danger level, e.g. over 2500ms in a very short period. And when we observe the rising latency, it is already more than 2500ms behind the requests peak comes into worker. At that time, adding more worker nodes will not help reliefing piles of requests in old worker nodes.

3.3 Extra Credit

To sum up, our scheduler is not trace-dependent, yet still achieves full score on all test cases :)