Group Members
- Ang Ze Yu (A0187094U)
- Choo Xing Yu (A0202132E)

Program Design

We will explain our program design from the 0%, then 100% case, then how we merge the two for the 50% case, showing how the addition of job knowledge influences our scheduling algorithm.

0% case

For the 0% case, we have no knowledge on the jobs and their sizes. Therefore, even though we can measure the elapsed time of each job, we cannot make any assumptions about the server capacity since we do not know the job size.

Therefore, our design here centers around a simple principle:

> *Assign whatever work there is to whatever resource is idle / available*

Specifically, we:

- limit the number of jobs processed by each server to **1** at any time
  - If all servers are busy, jobs are accumulated then "popped" once the servers are idle
- when a new job request comes in, we iterate through the servers and find **any first server that is idle**. We assign the job to this server.

In doing so, we **exploit the uncertainty and unevenness of server processing capacities**. This is because **faster servers** will **process jobs faster.** This leads to them becoming **"idle" faster, and more times than other servers**, which our algorithm naturally takes advantage of by assigning work more times to these faster servers.

100% case

Here, we have knowledge on all job sizes.

"Phase 1": processing capacity estimation (one-time)

We can therefore **estimate the processing capacities of servers** by **= job size / time elapsed**.

So, for the first **N (number of servers)** job requests, we assign the requests to servers whose capacities **are still unknown**.

When the servers complete these "probe" requests and reply, we calculate the time elapsed, and then obtain the processing capacity from the above formula.

“Phase 1.5”: job accumulation when no processing capacity estimates are available yet (one-time)

When we have sent out all the **N** probe requests but **none of them** (if even one of them completed already, we can proceed to **phase 2**. i.e., this phase is rather rare) has completed yet, it is possible that a new request arrives. In this case, we accumulate the job into a global queue for working on later.

Once the probe requests complete, we “pop” and assign these accumulated requests to the servers which completed the probe requests.

“Phase 2”: the scheduling algorithm (repetitive)

In this main phase, we exploit our server capacity estimations in the following way for each job (**whose size is known**):

1. For each server, we find a **response_time** estimate by
   - Calculating the **process_time_needed** for the current job (job size / capacity)
   - Calculate the **time to availability** (waiting time till the job can start) resulting from existing jobs being processed (if there is any). This timestamp is set in **step 4** below by the previous jobs.
   - Sum these together
2. To choose the server, we first prioritize servers which are **idle**. Our rationale here simply is that we should maximize resource usage in any case.
3. Then, to tie break amongst these servers, we choose the server with the best **response_time** and queue the job to be run here.
4. When doing this, we update the **time to availability** by adding on our calculated **process_time_needed** for the current job to the existing timestamp.

Some implementation details worth noting:

- We maintain our own queue abstractions. That is, each of our servers are only processing one job concurrently. The reasons for this are:
  - Our **time to availability** model requires that jobs are executed sequentially in the server one after another, and not sharing the resources amongst them.
  - The other more minor reason is that in this assignment, assigning jobs concurrently to the server does not utilize the server’s processing capacity any better.
    - This is not to say our solution is not generalizable outside of this assignment. (e.g., if each server has 4 physical CPU cores)
    - Because all we need merely do in this case is abstract each physical CPU core as a separate server.
- In reality, **process_time** is only **an estimate**. The processing might take slightly longer / slower due to e.g., **communication overheads**.
  - To tackle this, whenever a job is completed, we calculate the **actual** process_time (basically job completed time - start time). We calculate the difference between this actual value and our earlier process_time estimate, then add this difference into the **time to availability timestamp.** This allows us to maintain an accurate **time to availability**.

Rationale: This algorithm allows us to exploit our obtained server capacities by assigning jobs to servers that **minimize the response time** directly, which we define as the processing time needed + time to availability.

50% case

For the **50% case**, we try to take advantage of job sizes knowledge in the same way as the 100% case (since we don't know what test case our program is being run under), in that we estimate the **server capacities, time to availability, and process_time** as per the before procedure.

For job requests of **known sizes**, the same process as per before in the 100% case applies.

To tackle the case of job requests with **unknown sizes**, our algorithm dynamically applies a mix of our 0% and 100% case methods.

1. Find any **idle server** (same as 0% case) with the **highest server capacity** (now potentially available)
    a. This is the same as the 0% case except now we take advantage of **knowledge of server capacities** (once they are available)
2. If this **idle server is found**, go to **step 4**
3. If **all servers are busy** however, instead of accumulating the job, we apply the **100%** algorithm.
    a. we derive an **estimate for the job size** by using the **average of all known job sizes** encountered so far
    b. **Rationale:**
        i. From a program design perspective, this simply allows us to merge the logic for job requests of **known and unknown sizes** cleanly.
        ii. More importantly, this allows us to exploit the server capacities in the same way as the 100% case. While this is less accurate (due to us **estimating the job size**), we found that it still performs better.
        iii. We have mechanisms for dealing with the inaccuracy (implementation detail 2 in 100% case)
4. Like step 4 of the 100% case, we update the **time to availability** by adding on our calculated **process_time_needed** for the current job to the existing timestamp.
    a. This process_time in this case however, Is calculated from an estimate of the job size (average of all known job sizes)

Ergo, we take advantage of **some job size knowledge** by

- Applying the 100% algorithm for job requests of **known sizes**
- Applying the 0% algorithm for job requests of **unknown sizes**, enhanced by knowledge of server capacities. (choose the fastest idle server)
- Failing which (all servers are busy), applying the 100% algorithm for job requests of **unknown sizes**, by deriving **estimates of the job size**
    - This is less accurate of course but is the best choice in the 50% case.
    - We also have the above-mentioned correction mechanisms which by and large takes care of these inaccuracies along the way.
- Updating metadata for the 100% algorithm (**time to availability**) by using estimates

Results

(We obtained this on config_server/client_6 test files)

0%
*** Average completion time : 18.5519
*** 50th percentile time : 17.8000
*** 95th percentile time : 35.7000

50%
*** Average completion time : 17.7171
*** 50th percentile time : 15.9500
*** 95th percentile time : 36.4450

100%
*** Average completion time : 17.1046
*** 50th percentile time : 17.1250
*** 95th percentile time : 35.6950

As is to be expected, we can generally see that as our amount of knowledge increases, so too does the average completion time decrease =)