Samantha Rack

CSE 40822 Cloud Computing

Movie Rendering Service

Motivation

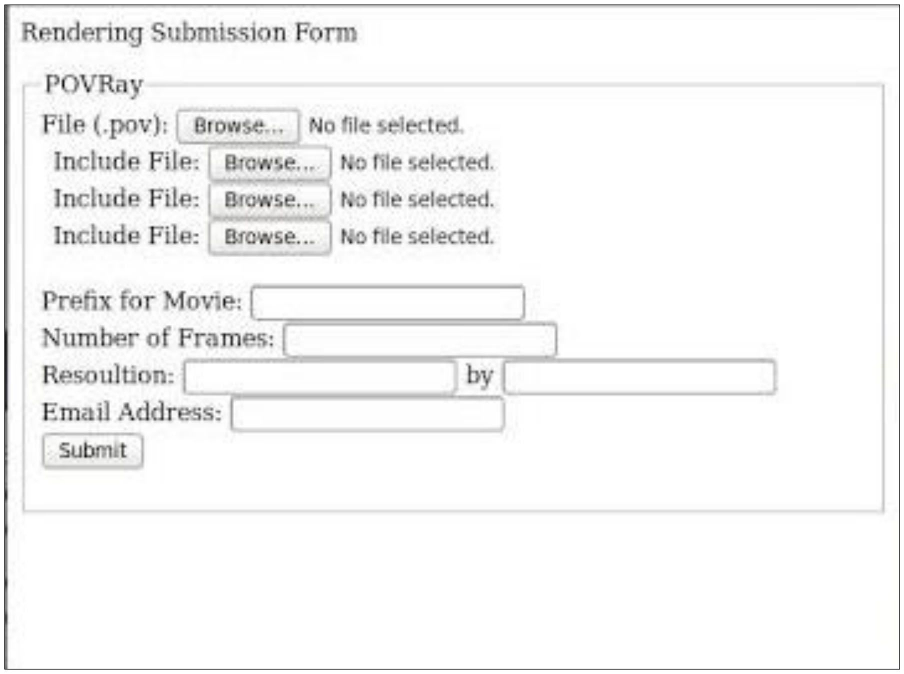
Frame rendering is a popular, computationally-heavy activity used to produce images for animations using software that can interpret specifications for each sequence in the animation. With animations running between 10 and 30 frames per second, there can be a huge number of frames that must be rendered for a production, with each taking a non-trivial amount of time to render. Fortunately, some frame rendering can be parallelized to reduce the total rendering time for a movie from days or weeks to hours. This potential for parallelization makes movie rendering an ideal candidate application to run in the cloud.

Users of frame rendering software are most focused on specifying the perfect frames via lighting, textures, object placement, and realistic movement of the objects. Sometimes, the users are artists with little knowledge of computers outside of the rendering software. Therefore, a portal to which users could straightforwardly submit rendering jobs without having to consider low-level details would be convenient. The users would not have to have an understanding of cloud computing or what an optimal job size would be because system could determine this for them.

For this project, I investigated two different rendering software systems, Autodesk's Maya and POVRay, an open source ray tracing program. Because the class had already worked with POVRay for an assignment, I initially aimed to focus on Maya, which is a software used by the Computer Science Department's Visual FX class. The frames that students render in the class take anywhere from seconds to hours to render, and their animation projects' rendering times increase linearly when they are run locally. I intended to build a web portal where students in the class could submit their Maya files to be run more effectively on Notre Dame's Condor cluster.

However, there were a number of obstacles in working with Maya rendering that are not present with POVRay. First, since Maya is commercial software, Notre Dame only owns about 30 licenses, with one license being in use either for a rendering job or for a student using the application to actively work on a project. Therefore, even if none of the students in the class were using the software, the maximum parallelization would be 30 machines. Second, because of the way Maya animation is calculated, parallel rendering is often not possible. Each frame depends on the values calculated in the frame preceding it. Thus, splitting the frames into groups to be rendered on different machines would cause redundant computing and would not give any performance improvements. Finally, Maya offers a large number of render options, causing two potential issues – (1) the user either being overwhelmed by the huge number of options on the interface or not satisfied if an option is missing; and (2) large potential for variance in the run times of each frame's rendering. The second issue is most debilitating for the service since the system aims to find optimal job times for the condor jobs by sampling one of the frames of the animation.

With these obstacles in mind, the project focuses only on POVRay rendering. There are a number of improvements introduced with this project that are not found in assignment one. The service offers an intuitive, simple web interface for users to submit their POVRay jobs without having to work with the terminal. Additionally, this interface does not ask how many jobs should be run in Condor cluster, but instead calculates that number which will give an optimal running time of approximately 30 minutes per job.   The interface can be seen in Figure 1 below.


Rendering Submission Form

POVRay

File (.pov): Browse... No file selected.

Include File: Browse... No file selected.

Include File: Browse... No file selected.

Include File: Browse... No file selected.

Prefix for Movie:

Number of Frames:

Resoultion: by

Email Address:

Submit

Figure 1. Screenshot of the simple web interface for POVRay rendering job submissions.

In addition to the web interface, this project extends to allow for the management of multiple job submissions from multiple clients using the submission form. The system tests the job length of one rendered frame to determine an ideal number of frames to be submitted per condor job. Finally, in order to not overrun the host machine or the Condor cluster, the system caps the number of total processes that can be running on the machine and checks the availability of the Condor pool before submitting more jobs.

System Architecture

The movie rendering system is comprised on 4 key components: a server that receives client requests with POVRay files and configurations for the renderings, a database that tracks the status and data location of each submitted rendering job, a process periodically checking the database and the status of the Condor cluster to determine when new jobs should be run (check_jobs), and one or more processes that create the submit files and compile the frames into a final video when they are complete.

Below in Figure 2 is a diagram of the overall system with the components interacting together. I will reference this diagram during the detailed explanations of each system component throughout the rest of this section.

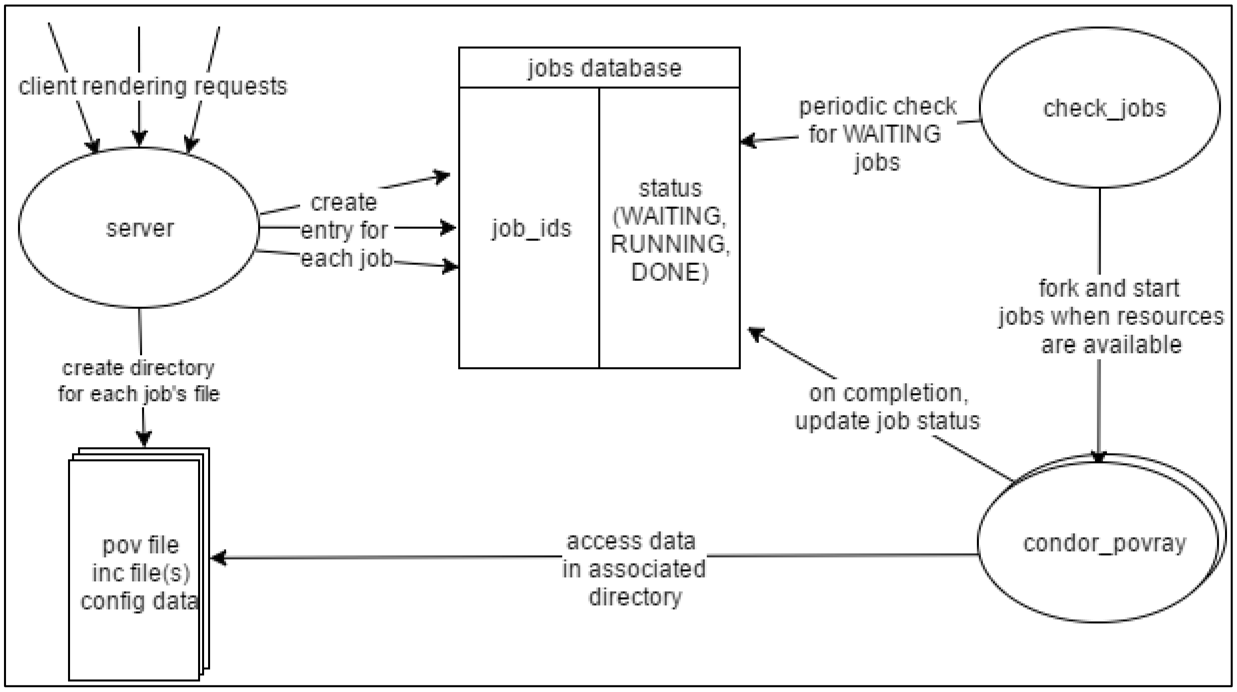

client rendering requests
server
create entry for each job
jobs database
job_ids
status (WAITING, RUNNING, DONE)
periodic check for WAITING jobs
check_jobs
fork and start jobs when resources are available
on completion, update job status
create directory for each job's file
pov file
inc file(s)
config data
access data in associated directory
condor_povray

Figure 2. Overview of the system architecture for the movie rendering service. Each circle indicates a process in the system, boxes indicate the organization of pertinent data on the machine, and arrows indicate interaction.

The first component involved when a client submits a POVRay job to be rendered is the server process. This process is a single threaded CherryPy server that provides the static HTML for the client's form and processes POST requests with the rendering's files and configuration information on submissions. Because there is no load balancing and only one server process with no threading,   it is imperative for this process to be fast in its processing so it can handle multiple clients. The server is not intended to scale like typical web applications, but it should be able to handle a few clients connecting and submitting jobs concurrently.

With this processing limitation, the server does very little with the data it receives. First, in a job-specific directory, it saves the POVRay and any include files that the client uploads for the job. In the same directory, it generates a configuration file that records all other pertinent information provided by the user – the number of frames to be rendered, resolution, the email at which to contact her on completion, and the name of the final movie to be created. Finally, the server creates an entry in the system's database to record metadata about the job. The primary purpose of the server is to act as a gatekeeper of the requests; it will accept as many rendering jobs as the clients request, but the jobs will only be started (by another process) when the system and Condor cluster is available to work on them.

The jobs database is a basic two-column database written in python to provide the functionality specific to this system. It tracks only two attributes of each job, the job id and the status of that job. The job id, however, also provides the location of the necessary files for that job to complete. The database also maintains a list of the queue of waiting jobs to be provided when computing resources become available for a new job to start running.

The most interesting components of the system are the two remaining processes: check_jobs and condor_povray. These are the components that handle job submission to Condor, scheduling for the submitted rendering jobs, and throttling of the number of jobs submitted based on the load on both the local system and the Condor cloud. The next process to be involved in the system after the job's information has been saved and an entry for it has been created in the database is check_jobs.

The check_jobs process is a script that runs periodically to check the database for waiting jobs and checks the local machine and the Condor cloud for the availability of resources. If both a rendering job is in the waiting state and there are resources available for that job to be run, then the check_jobs process forks a condor_povray process that will handle one specific job. The script first checks the availability of the Condor cluster using the ‘condor_status’ command. It filters out the machines that are in use or handling other jobs, and counts the unclaimed Linux machines to which it potentially could submit jobs. The Condor cluster is volatile with machines getting plugged in, machines getting unplugged, machines having their users become not idle, and jobs being submitted by other Condor users. Though the result of ‘condor_status’ produces a number that is stale as soon as it is received, it gives a best effort estimate of the status. It is more effective than not considering the Condor pool’s status at all. After checking the Condor cluster for availability, the check_jobs process determines the number of condor_povray jobs currently running locally to avoid overrunning the host machine. The number of simultaneous condor_povray jobs is capped at 10 for this system.

Following these resource checks, the check_jobs process obtains the next waiting job in the database’s queue. It then runs the first frame of this job (according to the resolution and the files provided by the client) to determine an estimated run time per frame of the job. Using this run time and the total number of frames requested by the user for the job, check_jobs determines the number of Condor submitted jobs that would be most effective for the rendering (with a cap of 250 Condor jobs so the cluster is not overwhelmed by just one job). Finally, the process compares this number to the availability of the Condor cluster and determines if the job can be effectively run. If the resources are not exhausted, then the check_jobs process will repeat this procedure with the next waiting job before it sleeps.

When it is determined by check_jobs that resources are available for a rendering job to be run, the process is forked to create a condor_povray job to handle the Condor job submissions for the given rendering. Since the resource management is handled by the check_jobs process, condor_povray is a straightforward process. It reads the configuration for the job it is assigned, generates Condor submit files with the optimal number of frames to be rendered with each job, and submits the files to the Condor pool. Finally, the process uses the 'condor_wait' command to block until all of the jobs have completed and then builds the .mpg file, producing the final animation. The user is then informed of the completion of her job via email and can access the resulting movie.

Evaluation of Correctness

The correctness of the system was evaluated via the output video and accompanying condor log files. The movies produced during testing were evaluated first for the length of time. The length of each of the movies produced matched the expected length that was input with the job information. In checking that the length was correct, the movies were watched to ensure that there were no unexpected jumps in the video that would indicate that frames may have been lost during the rendering.

Additionally, for one short length video, a side-by-side comparison was completed between locally rendered results and those obtained using the system. The one second movie that was used for this animation-to-animation comparison displayed no differences between the two movies. Because each of the other renderings tested on with the system using Condor would require more than 7 hours to render sequentially, these comparison tests were omitted. Instead, each resulting movie was watched to ensure the integrity of the frames comprising the rendered movie.

Evaluation of Performance

For performance testing, clients submitted a range of movie lengths for high resolution frames. Each frame was rendered with a width of 1920 pixels and height of 1080 pixels. I chose these frame dimensions for evaluation to make the rendering sufficiently large computationally so that parallelizing the rendering using Condor would show benefits. Rendering 1920x1080 frames locally requires 258 seconds (4.3 minutes) for completion, so even short videos would necessitate a significant number of computation hours. Alternatively, clients could have submitted very long videos with lower resolution; this would provide an equivalent computational load but with more frames being rendered on each Condor machine.

The testbed of movies that were complete are all renderings of a Rubik's Cube moving in an empty setting. I considered performing renderings of different examples, but because the actual video result of the testing was unimportant to performance evaluation, I chose to only use the Rubik's cube animation. The videos were of the following lengths: one second, ten seconds, one minute, and five minutes. Each video created was rendered at ten frames per second. Table 1 below summarizes the results of each run.

Table 1.Results of frame renderings of varying lengths done for testing.

Frames	Movie Length	sequential (sec)	(hr)	parallel (sec)	(hr)	jobs submitted	speedup	efficiency
10	1 sec	2580	0.7167	1368	0.38	2	1.885965	0.942982
100	10 sec	25800	7.167	5032	1.3978	20	5.127186	0.256359
600	1 min	154800	43	9842	2.7339	120	15.72851	0.131071

To visualize these results, Figure 3 below plots the total time required for each rendering job for both the projected sequential time (based on 258 seconds per frame) as well as the empirically collected time using the system with Condor.

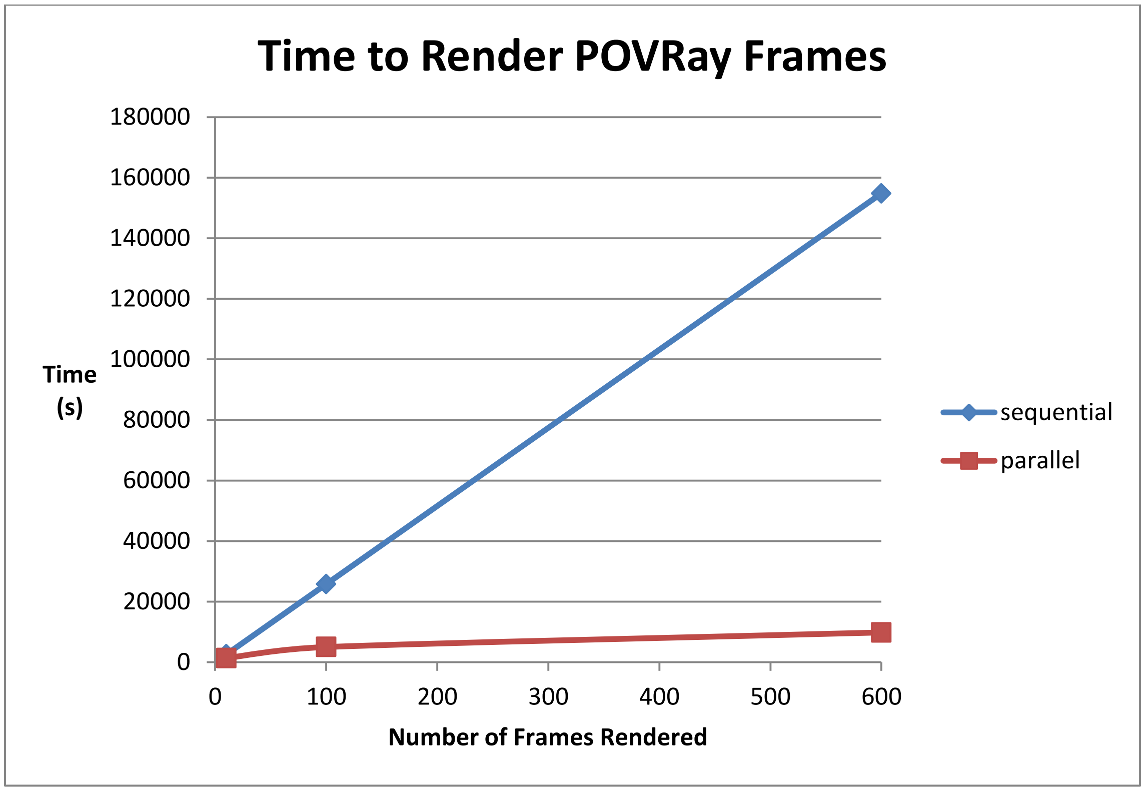


Figure 3. Graph demonstrating difference in render times of 10, 100, and 600 1920x1080 frames when they are rendered sequentially on a local machine and when they are rendered using the presented system.

Looking at the results in Table 1, the maximum speedup obtained with these tests was about 15 times. The graph of Figure 3 shows a promising trend for the parallel execution times when rendering more and more frames. However, referring back to Table 1, there is also a trend of greatly decreasing efficiency as the movies get longer.

I hypothesize that the decreasing efficiency found from this testing was a result of the stragglers in the Condor cluster. Stragglers are the final few Condor jobs that finish significantly later than the majority of the jobs because of slow machines or eviction. To further investigate the stragglers, I used the Condor Log Analyzer to produce the graphs in Figure 4 below of the jobs for the 10 second movie and 1 minute movie.

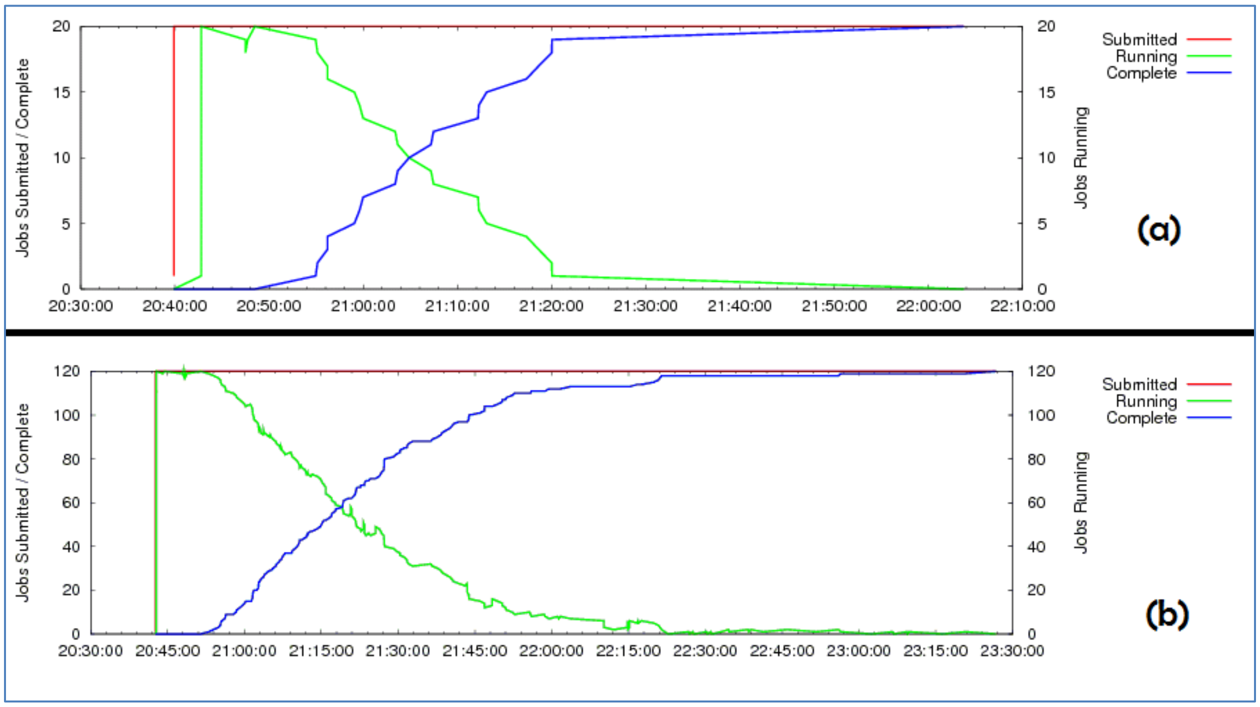


Figure 4. Graphs of data found in the Condor log files for (a) the 10 second movie rendering completed using 20 submitted jobs, and (b) the 1 minute movie rendering completed using 120 submitted jobs.

It is evident in the graphs of Figure 4 that stragglers are an important issue to be considered with the overall time of one rendering job's completion. Approximately halfway through each of the Condor submissions, about 95% of the jobs were completed (and therefore frames were rendered). Stragglers are unavoidable, but to mitigate their effects on turnaround time, the system could be more aware of their existence and submit duplicate jobs.

Because the aim of this system is to effectively use the resources available in Condor for multiple jobs and not to reduce the turnaround time for one job, stragglers are not a primary performance issue for this application. When a few stragglers are consuming cluster resources, there are still a large number of resources that have been used and freed by that job. Those resources could be allocated for a new waiting job while the older job finishes.