Homework 2: High Throughput Computing

Steven Stetzler

Pledge: On my honor as a student, I have neither given nor received unauthorized aid on this assignment.

1 Statement of Completion

I was successfully able to convert the single threaded process of rendering a movie into a high throughput process by rendering individual frames on separate computers. I found empirically that problems that can be solved with high throughput computing are infinitely parallelizable up to the number of computers that one has to work with. For example, if a movie has 250 frames, each can be rendered individually on 250 computers with little overhead. I found that once one tries to split a high throughput computing problem too finely over too small of a computing grid, there is some overhead that increases the expected run time of the program. For example, if a movie has 250 frames and one attempts to create 250 jobs to render each individual frame on a computing grid with only 100 computers, there will be pile-up as one cannot render all frames simultaneously given the resources at hand.

2 Problem Description

The problem was to render an animation containing 250 frames of a star exploding into a movie using the program `blender`. A simple single threaded approach to this problem would attempt to render each frame one at a time, until all frames are rendered and the movie can be made. However, since the rendering of each frame is a completely independent process (the rendering of any frame does not rely on any other frame being rendered), then this problem can be completely parallelized. Here, we attempt to parallelize this process by having several computers independently render different frames simultaneously. The parallelizability of this approach is limited only in the number of computers one has to work with.

3 Approach, Results, and Analysis

I wrote a `Python` script which generates and launches several jobs using the `SLURM` system on the University of Virginia Computer Science computing cluster. Throughout, I used the granger nodes in the cluster, which each have the following hardware specifications:

Processor: 4× Intel(R) Xeon(R) CPU E5-2623 v4 @ 2.60GHz

Cache: 10 MB of Intel SmartCache

Memory: 63 GB of RAM

The `Python` script I made created one or more `bash` scripts that each had one or more `blender` commands of the form

```
blender -t 1 -b Star-collapse-ntsc.blend -s <start> -e <end> -a
```

where `-t 1` indicates that each blender command should be single-threaded (to emulate each processor as a single computer), the `-b` indicates that the `blender` command should run in command-line mode, `-s <start>` indicates which frame to start rendering from the input file, `-e <end>` indicates to which frame the movie should be rendered, and the `-a` indicates that all frames between `<start>` and `<end>` should be

rendered. The input file `Star-collaple-ntsc.blend` contains animation information for a set a 250 frames which blender can pick out and render individually. This script was then run using the `SLURM` queuing system to run the `blender` commands on several nodes.

3.1 Approach

The challenge of this problem lies in choosing how to specify `<start>` and `<end>`. The simplest way is to follow a linear approach: if I have 2 computers at my disposal, I can have one node render the beginning (`<start> = 1`, `<end> = 125`) and one node render the end (`<start> = 126`, `<end> = 250`). If I have 250 nodes to work with, then for each node `i`, we would have `<start> = i` and `<end> = i` since each node would render exactly one frame. The benefits of this implementation is that for each computer, only 1 call to blender needs to be made, as we are rendering a set of adjacent frames on each computer. A downside to this approach is that this doesn't account for differences in the time it takes to animate each frame. For example, at the end of the movie, the animation gets much more complicated, and thus takes a longer time to render. Thus, with this approach, if we split the problem across 2 nodes one of the nodes will have to do more work than the other. This leads to a load balancing issue.

Load balancing can be solved in several ways. One solution is to interleave the animation schedule. For example, if you have N nodes at your disposal, you could have each node render the N^{th} frame in the movie. For example, with $N = 10$, one computer would render frame 1, 11, 21, 31, ..., and another would render frames 2, 12, 22, 32, ... covering all frames in an interleaved manner. This means that each computer will spend the same amount of time rendering frames at the beginning of the animation as at the end of the animation, thus balancing the load considerably. One issue with this approach is that all frames that a single computer should render are not adjacent. Thus, we are required to run `blender` $250/N$ times on each computer. This leads to some overhead in repeatedly starting up blender. Note that this method converges to the linear method described previously for $N = 250$, in which case each node is rendering exactly one frame.

Another approach to load balancing is to simply assign frames randomly. For example, if we have N nodes, we would randomly choose (without replacement) $250/N$ frames to assign to each computer. This lets us approach the problem of load balancing without having any prior knowledge about the movie we are trying to render. For example, perhaps the animation gears up in complexity in the middle or the beginning instead of at the end and this information was unknown to us. It makes sense then to just randomly assign frames to each node to deal with spreading the workload evenly over the set of nodes we are using. Again, a downside to this method is that each node is required to run `blender` $250/N$ times, providing some overhead that might increase the run time.

3.2 Results

Contained in Fig. 1 is a plot of the total time required to render the movie when the job is spread across some number of nodes. The data is labelled by the frame distribution method used, either a linear assignment, an interleaved assignment, or a random assignment of frames. By observing the data in Fig. 1, we note that the run time as a function of the number of nodes is a power law of the form

$$T = A \times N^B.$$

where T is the run time, N is the number of nodes used, and A and B are fit parameters. In Fig. 2 we plot our data on a log-log scale to elucidate the form of this power law. The fits on the data points that followed this trend are:

$$\begin{aligned} T_{linear} &= 300 \times N^{-0.833} \\ T_{interleave} &= 442 \times N^{-0.888} \\ T_{random} &= 444 \times N^{-0.890} \end{aligned}$$

Note that a truly parallel process would give us a scaling of $T \sim N^{-1}$ since with N nodes, we expect each node to render $\frac{250}{N}$ frames taking $\sim \frac{1}{N}$ time to perform. Since each node renders frames in parallel, it would only take $\sim \frac{1}{N}$ time in total.

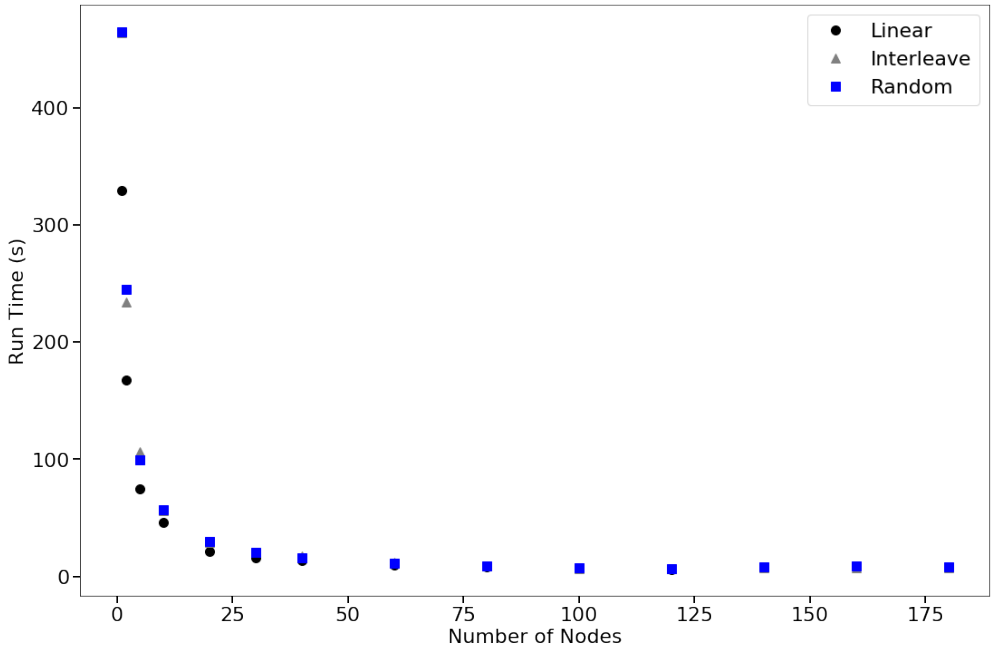

Figure 1: The time required to render the movie as a function of the number of nodes requested for the process.

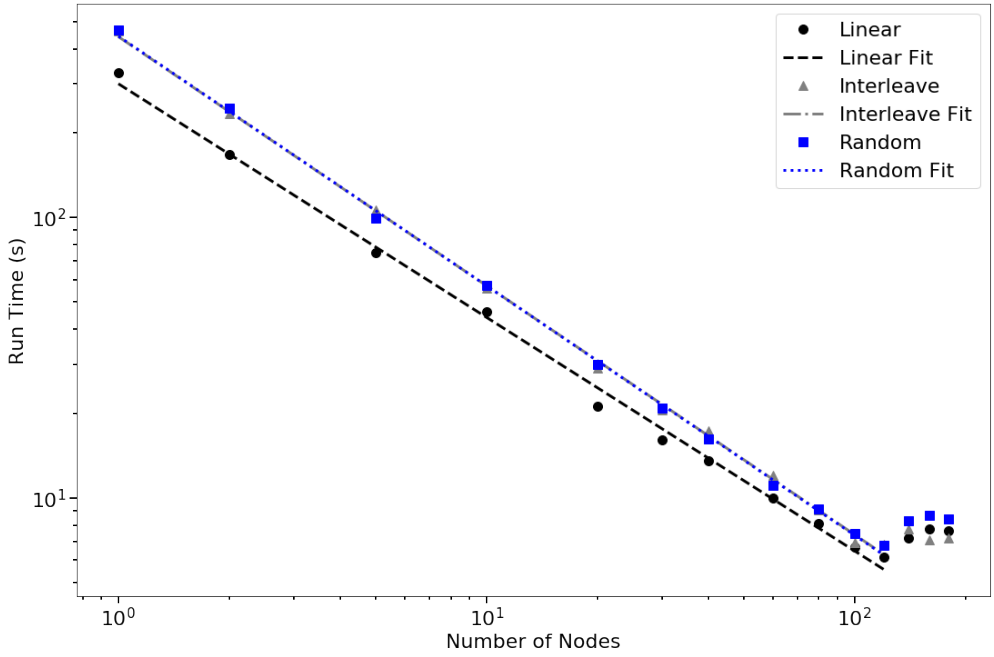

Figure 2: A log-log plot of the time required to render the movie as a function of the number of nodes requested for the process along with linear fits overlayed on the data for each of the three methods used to distribute tasks. Note the three points on the right which do not follow the trend that the other points do. This is due to requesting that the rendering be split over more nodes than were available on the computing grid we used.

The log-log scale also reveals systematic differences and similarities in the run time for each of the distribution methods described previously. We find that the linear distribution method performs better than both the interleaved and random distribution methods. This is due to the overhead incurred by having to run `blender` several times when using the interleave and random assignment method. We also note that as the number of nodes increases, the difference between the run times produced by each method shrinks, which is consistent with the theory that this difference is caused by the `blender` run overhead. For the interleave and random assignment, each node must perform $250/N$ calls to `blender`, which is proportional to the amount of overhead and which approaches 0 as N increases. Our theory predicts that the overhead should approach 0 as the number of nodes increases, and thus the results should approach that of the linear assignment, which is what is seen in the data. It is also interesting to note that the interleaving method did no better than simple random frame assignment, indicating that random assignment is just as good as using our prior knowledge that the end of the movie takes more time to render than the beginning. This is promising knowledge which tells us that it might be more beneficial for other problems to use a random assignment as well when we can't know a priori how to assign work to nodes.

We note that the run time scales very predictably up until we start distributing the problem over more than 120 nodes. This is because this was the maximum number of nodes that could be requested among the granger computers at the time of running. Once we run out of nodes to use, some processes have to wait for others to finish before they can run, causing some overhead. It is interesting to note that Fig. 2 indicates that run time is generally constant once this cap in the number of nodes is hit.

4 Conclusions

I found that I could parallelize rendering a movie very effectively using high throughput computing, in which a large problem made of independent sub-problems is split across several different computers. I found that for the problem of rendering a movie with `blender`, the overhead associated with multiple `blender` calls ruined attempts to perform proper load balancing of frame assignments to each computer used. I also found that simple random assignment of frames to computers was as effective as using prior knowledge about animation complexity throughout the movie to assign the frames. Finally, I found that the problem's running timing scaled favorably up until I requested more nodes than could be simultaneously provided by our computing grid. After that point, running time remained constant regardless of how many nodes were requested due to this bottleneck in the number of nodes in the scheduling system.

Here, I include the code I used to create the scripts that distributed the frames to separate nodes.

```
#
# CS 4444
# Steven Stetzler
# Homework 2: High Throughput Computing
#

import subprocess
import os
import argparse
import numpy as np

# Main function
def main():
    # Parse command line arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("host", type=str, help="The host to run on")
    parser.add_argument("num_nodes", type=int, help="The number of nodes to run on")
    parser.add_argument("--method", required=False, default="linear", help="The method to use for
        load balancing. Default=linear")
    args = parser.parse_args()
```

```
# Number of nodes to run on
num_nodes = args.num_nodes
# The host to request
host = args.host
# The load balancing method to use
method = args.method

# The blender file with frames to animate
file_name = "Star-collapse-ntsc.blend"

# Header information for sbatch scripts
header_lines = ['#!/bin/bash']
out_file = '#SBATCH --output={}'
error_file = '#SBATCH --error={}'
job_name = '#SBATCH --job-name="{0}-{1}"'
# The possible nodes these jobs could be run on
hosts = ['hermes[1-4]', 'trillian[1-3]', 'artemis[1-7]', 'qdata[1-8]', 'granger[1-8]',
    'nibbler[1-4]', 'slurm[1-5]', 'ai[01-06]']

# Pick out just the host we requested
exclude_list = [e for e in hosts if host not in e]

# Exlcude all hosts we didn't request
exclude_line = "#SBATCH --exclude={0}".format(",".join(exclude_list))

# Name of the script file to write commands out to
script_file = 'slurm_{0}_{1}_{2}.sh'

# Commands we want to run
time_command = 'date +%s%N'
hostname_command = 'hostname'
blender_command = 'time blender -t 1 -b {0} -s {1} -e {2} -a &> /dev/null'

# The total number of frames in the blender file
max_frames = 250

if method == 'linear':
    # Make a new directory for all of the scripts and output
    if not os.path.exists(os.path.join(host, "{}_nodes".format(num_nodes))):
        os.makedirs(os.path.join(host, "{}_nodes".format(num_nodes)))

    for i in range(num_nodes):
        # Assign frames to each node
        # Linear assignment: each node gets 250 / num_nodes frames, each in an adjacent chunk
            of frames
        lower = i * max_frames / num_nodes + 1
        upper = (i + 1) * max_frames / num_nodes

        # Add all lines to the script
        script_lines = [l for l in header_lines]
        script_lines.append(exclude_line)
        script_lines.append(out_file.format(os.path.join(host, "{}_nodes".format(num_nodes),
            "script_{}_{}_{}.out".format(i, lower, upper))))
        script_lines.append(error_file.format(os.path.join(host, "{}_nodes".format(num_nodes),
            "script_{}_{}_{}.err".format(i, lower, upper))))

        script_lines.append(job_name.format(lower, upper))
```

```
        script_lines.append(hostname_command)
        script_lines.append(time_command)
        script_lines.append(blender_command.format(file_name, lower, upper))
        script_lines.append(time_command)

        # Write out the script
        out = open(os.path.join(host, "{}_nodes".format(num_nodes), script_file.format(i,
            lower, upper)), 'w')
        for l in script_lines:
            out.write(l + '\n')
        out.close()

    # Start all jobs
    procs = []
    for i in range(num_nodes):
        lower = i * max_frames / num_nodes + 1
        upper = (i + 1) * max_frames / num_nodes
        procs.append(subprocess.Popen(['sbatch', os.path.join(host,
            "{}_nodes".format(num_nodes), script_file.format(i, lower, upper))]))

elif method == 'interleave':
    # Make a new directory for all of the scripts and output
    if not os.path.exists(os.path.join(host, "{}_nodes_{}".format(num_nodes, method))):
        os.makedirs(os.path.join(host, "{}_nodes_{}".format(num_nodes, method)))

    for i in range(num_nodes):
        # Specify the frames for each node
        # Interleave: the i^th node gets every i^th frame: for 10 nodes, node 1 gets frame 1,
            11, 21, 31, ...
        frames = np.arange(i + 1, 251, num_nodes)
        # lower, upper only for uniquely naming each script
        lower, upper = frames[0], frames[0]

        # Add all lines to the script
        script_lines = [l for l in header_lines]
        script_lines.append(exclude_line)
        script_lines.append(out_file.format(os.path.join(host, "{}_nodes_{}".format(num_nodes,
            method), "script_{}_{}_{}.out".format(i, lower, upper))))
        script_lines.append(error_file.format(os.path.join(host,
            "{}_nodes_{}".format(num_nodes, method), "script_{}_{}_{}.err".format(i, lower,
            upper))))

        script_lines.append(job_name.format(lower, upper))

        script_lines.append(hostname_command)
        script_lines.append(time_command)

        # Create blender command for each frame
        for frame in frames:
            script_lines.append(blender_command.format(file_name, frame, frame))

        script_lines.append(time_command)

        # Write out the script
        out = open(os.path.join(host, "{}_nodes_{}".format(num_nodes, method),
            script_file.format(i, lower, upper)), 'w')
        for l in script_lines:
            out.write(l + '\n')
        out.close()
```

```
    # Start all jobs
    procs = []
    for i in range(num_nodes):
        frames = np.arange(i + 1, 251, num_nodes)
        lower, upper = frames[0], frames[0]
        procs.append(subprocess.Popen(['sbatch', os.path.join(host,
            "{}_nodes_{}".format(num_nodes, method), script_file.format(i, lower, upper))]))

elif method == 'random':
    # Make a new directory for all of the scripts and output
    if not os.path.exists(os.path.join(host, "{}_nodes_{}".format(num_nodes, method))):
        os.makedirs(os.path.join(host, "{}_nodes_{}".format(num_nodes, method)))

    # Generate a random order of frames
    frames = np.arange(1, 251)
    frames = np.random.permutation(frames)

    for i in range(num_nodes):
        # Assign frames to each node
        # Random: randomly assign 250 / N frames to the i^th of N nodes.
        # Since we performed a permutation on the list [1, ..., 250], we just need to grab some
            adjacent slice of this list to assign
        # frames to the node, so we use the same synatx as the linear case
        lower = i * max_frames / num_nodes
        upper = (i + 1) * max_frames / num_nodes

        # Add all lines to the script
        script_lines = [l for l in header_lines]
        script_lines.append(exclude_line)
        script_lines.append(out_file.format(os.path.join(host, "{}_nodes_{}".format(num_nodes,
            method), "script_{}_{}_{}.out".format(i, lower, upper))))
        script_lines.append(error_file.format(os.path.join(host,
            "{}_nodes_{}".format(num_nodes, method), "script_{}_{}_{}.err".format(i, lower,
            upper))))

        script_lines.append(job_name.format(lower, upper))

        script_lines.append(hostname_command)
        script_lines.append(time_command)

        # Get all assigned frames and make a blender command for each
        script_frames = frames[lower:upper]
        for frame in script_frames:
            script_lines.append(blender_command.format(file_name, frame, frame))

        script_lines.append(time_command)

        # Write out the script
        out = open(os.path.join(host, "{}_nodes_{}".format(num_nodes, method),
            script_file.format(i, lower, upper)), 'w')
        for l in script_lines:
            out.write(l + '\n')
        out.close()

    # Start all jobs
    procs = []
    for i in range(num_nodes):
        lower = i * max_frames / num_nodes
```

```
            upper = (i + 1) * max_frames / num_nodes
            procs.append(subprocess.Popen(['sbatch', os.path.join(host,
                "{}_nodes_{}".format(num_nodes, method), script_file.format(i, lower, upper))]))

# Run main only if this script was called directly and not imported
if __name__ == "__main__":
    main()
```