CS 4444 – Report for Homework 2

Jerry Sun (ys7va)

1 Problem Description

The goal of this assignment is to write a shell script to optimize the runtime for a film rendering problem. In particular, we want to explore the performance of converting a sequential program into a high-throughput implementation using SLURM on Rivanna. More specifically, the input file we are given is Star-collapse-ntsc.blend, and we are required to render 250 individual frames using blender. After that, we will need to use ffmpeg to combine those individual frames to output an avi file.

2 Metadata

2.1 Software

SLURM: version 14.11

GNU bash: version 4.1.2

blender: version 2.70

ffmpeg: version 2.3.1

2.2 Hardware

10 core Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz with a cache size of 25600KB

3 Approach

For this assignment, we are required to parallelized the whole film rendering process. There are two main process in this problem. First is the frame rendering process using blender, and the second is to combine individual frames. Since the second process can only work on a single node, so our main focus is on parallelizing the blender rendering process.

First, I modified the *slurm.sh* using *srun* to ensure all process are running in parallel, and added $-j\ k$ flag in blender so now each single process will render each frame after jumping across k frames but only invoke *blender* only one time(detail explanation see code comments in appendix). By doing this, the workload in different tasks can be distributed evenly, because based on testing, we found that the rendering later frames seem to be more time consuming than that of the earlier ones.

Secondly, to automize the time testing based on number of process I use, I wrote *calculate.sh* which will automatically submit *slurm.sh* based on different number of processes, and record the time spent on each task correspondingly.

Finally, to confirm that different partitioning method did give significantly different speedup, I wrote the *partition.sh* to test a naive separate method, as I just evenly split the jobs into k parts and run all those tasks in parallel.

4 Performance

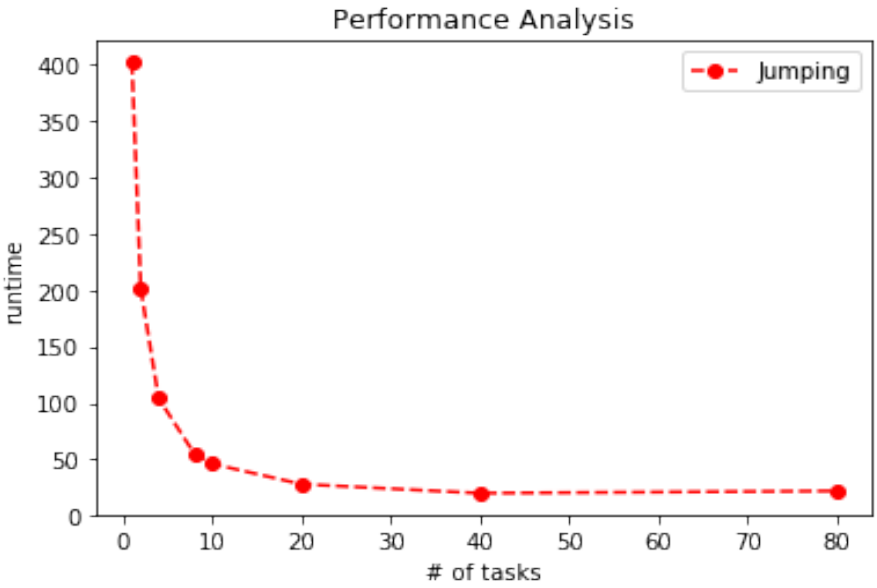
In this section, I will give a brief analysis for the data I retrieved from the testing above. Since the clock time of the program running on Rivanna heavily depends on the load of the certain nodes that the task is assigned, there are various 'noisy' data during experiment, and those apparent outliers are eliminated when recorded.

4.1 Runtime Analysis

Followed are all the inliers recorded during testing on Rivanna, and the average results for each number of tasks. Note that for all number of tasks smaller or equal to 20, all processes ran on different nodes, and for 40 tasks, they ran on 20 nodes, each got 2 tasks, and for 80 tasks all 20 nodes got 4 tasks.

iterations	1	2	4	8	10	20	40	80
1st run	401	203	105	55	42	29	20	25
2nd run	400	201	100	57	46	27	17	19
3rd run	402	198	103	58	46	29	21	20
4th run	405	200	105	51	43	30	22	22
5th run	406	199	107	52	45	28	21	22
6th run	401	205	110	54	46	27	18	24
7th run	399	205	104	59	49	27	20	19
8th run	400	201	103	54	50	29	22	20
9th run	405	200	100	54	44	30	17	23
10th run	400	201	102	55	43	29	23	20
average	402	201	104	55	46	28	20	22

Followed is the comparison graph given the average results above. We can see it is almost exponentially decreasing at first but when reaching more than 20 nodes, the increase in speed seems to shrink really quickly. This is mainly due to the overhead caused by communication and imbalance work load in different tasks.

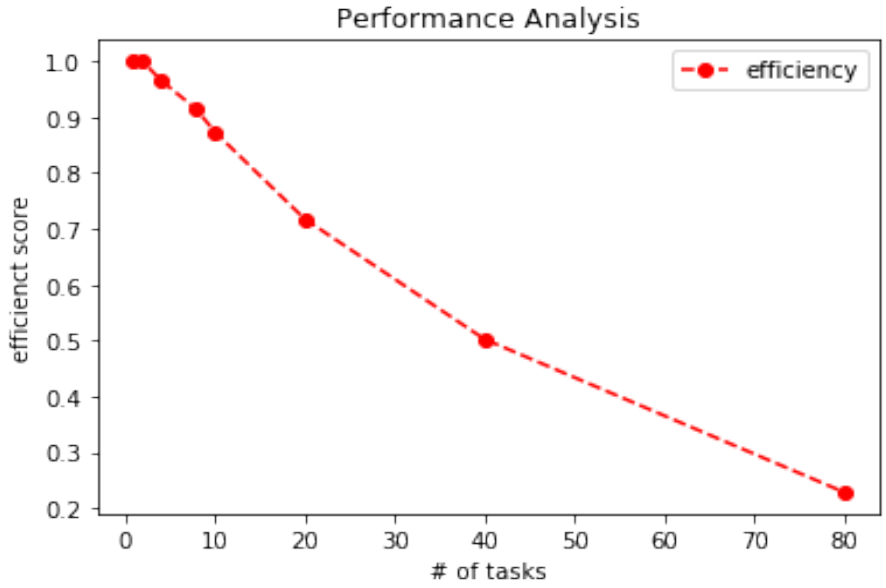


4.2 Efficiency Analysis

Here we also introduce another metric in parallel computing that is the efficiency. The equation is given below.

$$\text{Efficiency} = \frac{\text{Sequential Execution Time}}{\text{Number of Processors} * \text{Parallel Execution Time}}$$

And followed is the efficiency graph given the data above.



While we are looking for a high throughput strategy to faster the frame rendering process, we also want to keep the whole process relatively efficient. Therefore, based on the statistic above we would rather prefer 20 tasks in this particular problem to accelerate this process to avoid overhead as well as the efficiency loss.

4.3 Partition Analysis

As mentioned at the beginning, we also want to see if the 'jumping strategy' is truly an efficient way to divide the whole task. Here we also experiment with a linear partitioning strategy in which each process works on approximately equal number of continuous frames. The test was only done for 20 tasks.

	jumping	linear
1st run	29	45
2nd run	27	48
3rd run	29	54
4th run	30	60
5th run	28	55
6th run	27	56
7th run	27	56
8th run	29	49
9th run	30	50
10th run	29	50
average	28	53

Therefore, we can see a clear gap between the jumping strategy and the linear partitioning strategy, which is mainly because of the uneven workload distributed in different tasks.

5 conclusion

After the analysis above, we finally pick 20 tasks, and a so-called jumping strategy in parallelize this frame rendering process, which yields us a speedup ratio 14.3 and a efficiency score of 0.71.

6 Pledge

On my honor as a student, I have neither received nor given help on this assignment.

Signature: _____

7 Appendix

7.1 calculate.sh

```
#!/bin/bash
# Jerry Sun ys7va 2017-03-07
# calculate.sh
# used for automatically submit slurm.sh to SLURM and record the timing

# the first parameter given specified the number of iterations to run
iterate=$1
# the second parameter given specified the number of frame to render in blender
# 1 - 250 used for testing (use smaller value)
frameCounts=$2
# list of number of tasks that is going to be tested
numOfTasksList=(40 20 10 8 4 2 1)

# for loop to submit all tasks "iterate" times
for i in `seq 1 $iterate`;
do
# loop around all different number of tasks
for numOfTasks in "${numOfTasksList[@]}"
do
    # determine the task distribution
    # if numOfTasks is larger then 20, then there are in total 20 nodes, and each is ↩
        assigned
    # with corresponding number of tasks, namely numOfTasks/20.
    if [ $numOfTasks -gt 20 ]
    then
        let "tasksPerNodes=$numOfTasks/20"
        numOfNodes=20
    else
        # if numOfTasks is smaller or equal to 20, they are all distributed on different ↩
            nodes
        # and each node has only one task
        tasksPerNodes=1
        numOfNodes=$numOfTasks
    fi
    # endif
    # record the hyperparameter specified above into outputfile
    echo "Total Tasks: $numOfTasks, Nodes: $numOfNodes, Tasks per node: $tasksPerNodes" >>↩
         outputfile
    # submit the shell script to slurm specifying the required hyperparameters(detail see ↩
        slurm.sh)
    sbatch --nodes=$numOfNodes --ntasks=$numOfTasks --ntasks-per-node=$tasksPerNodes slurm↩
        .sh $frameCounts
    # check if output.avi has been produced, if not keep waiting, and recheck every 10 ↩
        seconds
    while [ ! -f output.avi ];
    do
        sleep 10
    done
    # grep the recorded timing in output of the slurm.sh into the outputfile
    grep 'Durations' script-output >> outputfile
    # remove the output.avi and ready for the next task
    rm -f output.avi
done
# endfor for
done
```

7.2 slurm.sh

```
#!/bin/bash
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=3000
#SBATCH --time=00:15:00
#SBATCH --partition=parallel
#SBATCH --account=parallelcomputing
#SBATCH --output=script-output
#SBATCH --exclusive

#Jerry Sun ys7va 2017-03-07
#slurm.sh
#used for submit parallel tasks onto slurm
#example submit format
#sbatch --nodes=1 --ntasks=1 --ntasks-per-node=1 slurm.sh 250
#three flags specify the number of nodes/tasks, and the task distribution
#the first parameter specify the number of frame to render

#time counter start
STARTTIME=$(date +%s)

#the total number of task = numOfNodes * numOfTasksPerNode
let "multi=$SLURM_NNODES * $SLURM_NTASKS_PER_NODE"

# the number of frames to be used to generate the video
frame_count=$1

#load the renderer engine that will generate frames for the video
module load blender

# -b option is for command line access, i.e., without an output console
# -s option is the starting frame
# -e option is the ending frame# -a option indicates that all frames should be rendered
# -j option is the number of frames to skip from the last generated frame

# run multi number of tasks all starting from different number of frames
# if multi=4 then 4 tasks will generate corresponding frames
# 1, 5, 9, 13 ...
# 2, 6, 10, 14 ...
# 3, 7, 11, 15 ...
# 4, 8, 12, 16 ...
for m in $(seq 1 $multi)
do
    # -n 1, -N 1 make sure each task take one whole process
    # & option push the last task to the backend and open a new process for the next task
    srun -n 1 -N 1 blender -b Star-collapse-ntsc.blend -s $m -j $multi -e $frame_count -a ↩
        &
done

# wait to make sure all processes finish before proceeding
wait

# need to give the generated frames some extension; otherwise the video encoder will not ↩
    worki
ls star-collapse-* | xargs -I % mv % %.jpg

# load the video encoder engine
module load ffmpeg

# start number should be 1 as by default the encoder starts looking from file ending with ↩
    0
```

```
# frame rate and start number options are set before the input files are specified so that the
# configuration is applied for all files going to the output
ffmpeg -framerate 25 -start_number 1 -i star-collapse-%04d.jpg -vcodec mpeg4 output.avi

# end clock
ENDTIME=$(date +%s)

# calculate the time spent for the whole process
let "DURATION=$ENDTIME-$STARTTIME"

# output the time spent into script-output
echo "Total Durations: $DURATION seconds"

# remove all the generated jpg files for future work
rm -f *.jpg
```

7.3 seq.sh

```
#!/bin/bash
#SBATCH --nodes=20
#SBATCH --ntasks=20
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=3000
#SBATCH --time=00:15:00
#SBATCH --partition=parallel
#SBATCH --account=parallelcomputing
#SBATCH --output=script-output
#SBATCH --exclusive

#Jerry Sun ys7va 2017-03-07
#seq.sh
#used for submit parallel tasks onto slurm for linear partition strategy
#example submit format
#sbatch slurm.sh 250
#the first parameter specify the number of frame to render

#start clock
STARTTIME=$(date +%s)
# the number of tasks to be used to generate the video
let "multi=$SLURM_NNODES*$SLURM_NTASKS_PER_NODE"

# the number of frames to be used to generate the video
frame_count=$1
# determine how many frames each task need to generate
let "step=$frame_count/$multi"

#load the renderer engine that will generate frames for the video
module load blender

# -b option is for command line access, i.e., without an output console
# -s option is the starting frame
# -e option is the ending frame# -a option indicates that all frames should be rendered

# number of iterations need to be run except last one
let "iterations=$multi-1"
# the startFrame for the first process
startFrame=1
```

```
# the endFrame for the first process
let "endFrame=1+$step"

# submit (iterations - 1) number of tasks
for m in $(seq 1 $iterations)
do
  # echo the start and the end frame for debugging
  echo "from $startFrame to $endFrame"

  # flags are similar to slurm.sh, but only involves -s and -e
  # used to specify the range of the frame to generate for this process
    srun -n 1 -N 1 blender -b Star-collapse-ntsc.blend -s $startFrame -e $endFrame -a &
  # determine the next startFrame is just the one after current endFrame
  let "startFrame=$endFrame+1"
  # the next endFrame should be the number of startFrame plus number of frames
  # each process is assigned
  let "endFrame=$startFrame+$step"
done

# Final process to make sure it stops at 250
let "startFrame=$endFrame+1"
echo "from $startFrame to 250"
srun -n 1 -N 1 blender -b Star-collapse-ntsc.blend -s $startFrame -e 250 -a &

# wait untill all processes are finished
wait

# need to give the generated frames some extension; otherwise the video encoder will not ↩
    worki
ls star-collapse-* | xargs -I % mv % %.jpg

# load the video encoder engine
module load ffmpeg

# start number should be 1 as by default the encoder starts looking from file ending with ↩
    0
# frame rate and start number options are set before the input files are specified so that↩
     the
# configuration is applied for all files going to the output

ffmpeg -framerate 25 -start_number 1 -i star-collapse-%04d.jpg -vcodec mpeg4 output.avi

# end clock
ENDTIME=$(date +%s)

# calculate the time spent for the whole process
let "DURATION=$ENDTIME-$STARTTIME"

# output the time spent into script-output
echo "Total Durations: $DURATION seconds"

# remove all the generated jpg files for future work
rm -f *.jpg
```