CS 4444 Parallel Computing Spring 2017

Homework 2: High Throughput Computing

Eamon Collins

When executed, my code successfully renders 250 frames from the given .blend file and produces a movie, output.avi. The best speedup I observed compared to a 1 processor run was at 10 processors, with a speedup factor of 8.13.

Problem Description:

The task in this assignment was to produce a movie from a blender file by distributing the rendering of each of the frames over a certain number of processors. Since each frame can be rendered completely independently of the others, this problem can be classified as “embarrassingly parallel”. After rendering the frames, a 10 second movie can be produced to quickly check if everything went smoothly during the rendering.

Approach:

To render all the frames in the least time possible, I split up the frames as equally as possible among all the processors. As recommended in the assignment, I used a stride when assigning frames to each processor equal to the number of processors, such that if there were 20 processors, processor 1 would be responsible for frames number 1, 21, 41, and so on. (Blender starts frame indexing at 1 and so all the indexes I used in the project also started at 1.)

This meant that each time a frame is rendered, a new call to blender had to be made as no processor was rendering any adjacent frames. I suspected that the extra overhead induced by this technique would be offset by the fact that the frames at the end took much longer to render than the frames at the beginning.

However, using this method of one blender call per frame produced horrible speedup, so I investigated and managed to put together a SLURM script that kept the striding of frames on each processor but only made one call to blender. I am still trying to make sense of what happened, so I present results from both methods below and compare them in my discussion.

After the blender jobs are done for each processor, the script tests to see if there are 250 images in the folder and if there are it collates them into output.avi, the movie. This means that whichever processor finishes last is the one to create the movie, so it is only done once.

Results:

All results are on the Rivanna system using master.sh to submit SLURM jobs. Results are measured from the start of execution until the movie is successfully made.

Below are the results for my first attempt, where a separate blender call is made for each of the frames.

1 blender call per frame

Number of Processors	Execution Time (sec)	Speedup
1	1282	1
10	820	1.56
20	711	1.80
40	683	1.88
80	633	2.03

The next table is from my second attempt, where I made only a single blender call per process but kept the striding.

1 blender call per processor

Number of Processors	Execution Time (sec)	Speedup
1	390	1
10	48	8.13
20	309	1.26
40	374	1.04
80	529	.74

Analysis:

For the first set of results, the trend is a steadily increasing speedup, although not approaching anywhere near what I would have expected. The speedup from using 80x the resources is barely above 2, which I would consider pretty pathetic. Seeing as this is an embarrassingly parallel problem and there is no communication between processors much less communication overhead to consider, what I presumed to be the problem was the memory bandwidth. If every processor is making a separate blender call for every frame, every frame it needs to retrieve the .blend file and load it into RAM. This many calls to get the .blend file from different processors could understandably bottleneck the system and offset the benefit received from more processors. While for this method 1 processor would have to perform the same amount of blender calls, the calls for the .blend file would not conflict and might not cause undue stress on the system. The submission script for this method can still be seen in render.slurm, although render2.slurm is the one now used by master.sh.

To try and alleviate this possible bottleneck, I found a way to make only one blender call per process and keep the stride, as seen in render2.slurm. This did in fact drastically decrease the execution time, although now there is an odd effect where increasing the processor count results in drastic speedups up until about 15 processors, (not listed above but 15 processors were able to complete the movie in about 37 seconds,) at which point it starts to drop off significantly. In fact, at 80 processors the execution time was actually significantly slower than on 1 processor. Looking at which frames are generated first at high processor counts, it seems that only very few of the processors were actually starting rendering when the process started, and they rendered all their frames and completed before

other processes even started, although SLURM lists their CPU time as well as the wall-clock time as the exact same.

In a further attempt to help with memory bandwidth issues, I created render3.slurm which creates a directory in /tmp for each job submitted and copies in the .blend file. Each job then uses its own copy of the .blend file in an attempt to make sure they weren't all trying to access the same file. (A last ditch attempt as common sense told me that each processor only needed to load the small file into RAM once at the beginning.) At the end of the job, each processor deleted its respective directory and .blend file to clean up after itself. This produced no noticeable change in execution time from render2.slurm, so currently master.sh uses render2.slurm but I kept render3.slurm around in case you wanted to check it out.

Another possible explanation is that this many jobs overload the SLURM capabilities and only allows some jobs to start. However, this does not seem likely as SLURM should be able to handle 80 jobs at a time, and certainly should not increase the execution time past that of 1 processor. This hits at the heart of why this is so confusing: the bottleneck posed by 80 processors seems to cause a failure so catastrophic it doesn't just limit the number of effective processors, it causes the system to actually function worse than simply running the job on one processor. This would suggest more problems than simple competition for resources, but in an embarrassingly parallel program like this I just think there's that much going on. Please run my jobs, get different results, and tell me so that at least I know I'm not insane.

Conclusions:

Even in situations like embarrassingly parallel problems, other considerations come into play when increasing the processor count, as there can be resource limitations other than the simple number of processors. Know your system well and run tests to determine if there is a "sweet spot" in the number of processors.

Files:

master.sh – the master script that submits as many slurm submission scripts as the command line argument tells it to.
Render.slurm – simple submission script that strides the frames but calls blender multiple times
render2.slurm – Improved submission script, calls blender only once per job.
Render3.slurm – creates a .blend file copy for each job, does not improve over render2, is around just for show
script-output – the output of the submission scripts
frames/ - directory my submission scripts place the frame jpegs

How to run my code:

make sure there's nothing in the frames directory
./master.sh [num procs]
go to frames directory and watch it fill up with *ls s*|wc -l*
squeue to see the time, or for better benchmarking, run *sacct –format=JobId,CPUTime*
after the jobs have been completed to see each one's cpu runtime.

Pledge:

On my honor as a student I have neither given nor received aid on this assignment.
-Eamon Collins

MASTER.SH

```
#!/bin/bash

NUM_PROCS=$1
CURR_PROC=1
while [ $CURR_PROC -lt $(($NUM_PROCS+1)) ];
do
        sbatch render2.slurm $NUM_PROCS $CURR_PROC
        ((CURR_PROC++))
done
```

RENDER.SLURM

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1096
#SBATCH --time=00:35:00
#SBATCH --part=eqa-cs6414
#SBATCH --account=parallelcomputing
#SBATCH --output=script-output

# the number of frames to be used to generate the video
NUM_PROCS=$1
PROC_ID=$2

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

#Attempt at putting the blend file local to each node so that memory access isn't a bottleneck
#srun mkdir /"$SLURM_JOB_ID"
#sbcast Star-collapse-ntsc.blend /"$SLURM_JOB_ID"/Star-collapse-ntsc.blend

# -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
iter=$PROC_ID
while [ $iter -lt 251 ];  #251
do
        if ( ls frames/star*$(($iter-$NUM_PROCS)) 1> /dev/null 2>&1 ) || [ $iter -le
$NUM_PROCS ] ; then
                blender -b Star-collapse-ntsc.blend -s $iter -e $iter -o //frames/star-collapse_ -t 1
-a
        fi
        if ls frames/star*$iter 1> /dev/null 2>&1; then
```

```
                iter=$(($iter+$NUM_PROCS))
        fi
done

if [ $(ls frames/star-collapse_* | wc -l) -eq 250 ]; then
        # need to give the generated frames some extension; otherwise the video encoder will
not work
        ls frames/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 frames/star-collapse_%04d.jpg -vcodec
mpeg4 output.avi
fi

#srun rm -rf /"$SLURM_JOB_ID"
```

RENDER2.SLURM

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1096
#SBATCH --time=00:35:00
#SBATCH --part=eqa-cs6414
#SBATCH --account=parallelcomputing
#SBATCH --output=script-output


# the number of frames to be used to generate the video
NUM_PROCS=$1
PROC_ID=$2

# 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

#create a string of -f arguments specifying each frame that this processor is responsible
#for rendering. This allows one call to blender per processor.
iter=$PROC_ID
frames=""
while [ $iter -lt 251 ];  #251
do
frames="$frames-f $iter "
iter=$(($iter+$NUM_PROCS))
done

#the blender call. $frames is the variable with the frames specified.
blender -b Star-collapse-ntsc.blend -o //frames/star-collapse_ -t 1 $frames

#if all frames are there, collate them into a movie. Only the last processor to complete
#will execute this, and then only if every frame has been successfully generated
if [ $(ls frames/star-collapse_* | wc -l) -eq 250 ]; then
	# need to give the generated frames some extension; otherwise the video encoder will
not work
	ls frames/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 frames/star-collapse_%04d.jpg -vcodec
mpeg4 output.avi
fi
```

RENDER3.SLURM

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1096
#SBATCH --time=00:35:00
#SBATCH --part=eqa-cs6414
#SBATCH --account=parallelcomputing
#SBATCH --output=script-output

# the number of frames to be used to generate the video
NUM_PROCS=$1
PROC_ID=$2

# 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
iter=$PROC_ID
frames=""
while [ $iter -lt 251 ];  #251
do
frames="$frames-f $iter "
iter=$(($iter+$NUM_PROCS))
done

#create a directory in tmp for each job and store a copy of the .blend file in it
srun mkdir /tmp/ec3bd
srun mkdir /tmp/ec3bd/"$SLURM_JOB_ID"
srun cp Star-collapse-ntsc.blend /tmp/ec3bd/"$SLURM_JOB_ID"/Star-collapse-ntsc.blend

#only access this processors copy of the blend file so they aren't all trying to open the same
file
blender -b /tmp/ec3bd/"$SLURM_JOB_ID"/Star-collapse-ntsc.blend -o
/scratch/ec3bd/HW2/frames/star-collapse_ -t 1 $frames

if [ $(ls frames/star-collapse_* | wc -l) -eq 250 ]; then
        # need to give the generated frames some extension; otherwise the video encoder will
not work
        ls frames/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 frames/star-collapse_%04d.jpg -vcodec
mpeg4 output.avi
fi
#cleaning up after myself
srun rm -rf /tmp/ec3bd
```