Compiling and Running the Program

The submission folder contains 3 sub-folders, 1 pdf file, and 1 env.output file. The table below explains the usage of each sub-folder.

Folders/Files	Comments
task1/	This folder stores all the source code for the serial program
task2/	This folder stores all the source code for the multi-threading program
env.out	Environment output for my development environment
report.pdf	The writeup of this assignment

Table 1, Top-tier directory

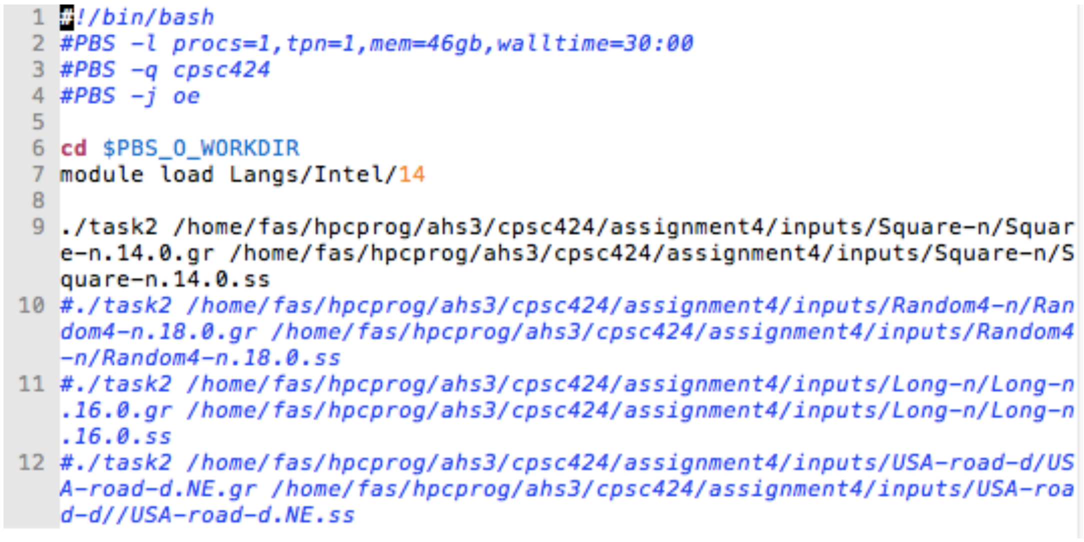
In each folder mentioned above, there are two scripts, *Assignment3BuildAll.sh* and *Assignment3RunAll.sh.* Using *task2* as an example here, to **compile** the program, one can do:

1. *Bash$ cd task2*
2. *Bash$ ./Assignment4BuildAll.sh*

Up to this point, one should be able to see a binary file, *task2*, generated. To **run** the program, one can execute:

1. *Bash$ qsub Assignment4RunAll.sh*

The command above submits a job to *Torque* and run the multi-threading program using the ***Square-n.14.0.gr*** dataset found in the ***/home/fas/hpcprog/ahs3/cpsc424/assignment4/inputs/Square-n/*** folder. I do not include the input data in my submission since it is rather large for a submission package. Please fetch the data directly from the directory above. In order to run the program with a different dataset, one can open *Assignment4RunAll.sh* (figure 1) and change the first and second input parameters to the *task2* binary file. Note that in Figure 1, I have commented out some of the examples from other datasets. One can simply uncomment these to run the program with the datasets. Once done changing, one can run the same *qsub* commands as shown above. Note that the total run time for larger dataset can take up to 20 minutes, for largest datasets, it may excess 30 minutes torque limitation.



```
#!/bin/bash
#PBS -l procs=1,tpn=1,mem=46gb,walltime=30:00
#PBS -q cpsc424
#PBS -j oe

cd $PBS_O_WORKDIR
module load Langs/Intel/14

./task2 /home/fas/hpcprog/ahs3/cpsc424/assignment4/inputs/Square-n/Square-n.14.0.gr /home/fas/hpcprog/ahs3/cpsc424/assignment4/inputs/Square-n/Square-n.14.0.ss
#./task2 /home/fas/hpcprog/ahs3/cpsc424/assignment4/inputs/Random4-n/Random4-n.18.0.gr /home/fas/hpcprog/ahs3/cpsc424/assignment4/inputs/Random4-n/Random4-n.18.0.ss
#./task2 /home/fas/hpcprog/ahs3/cpsc424/assignment4/inputs/Long-n/Long-n.16.0.gr /home/fas/hpcprog/ahs3/cpsc424/assignment4/inputs/Long-n/Long-n.16.0.ss
#./task2 /home/fas/hpcprog/ahs3/cpsc424/assignment4/inputs/USA-road-d/USA-road-d.NE.gr /home/fas/hpcprog/ahs3/cpsc424/assignment4/inputs/USA-road-d//USA-road-d.NE.ss
```

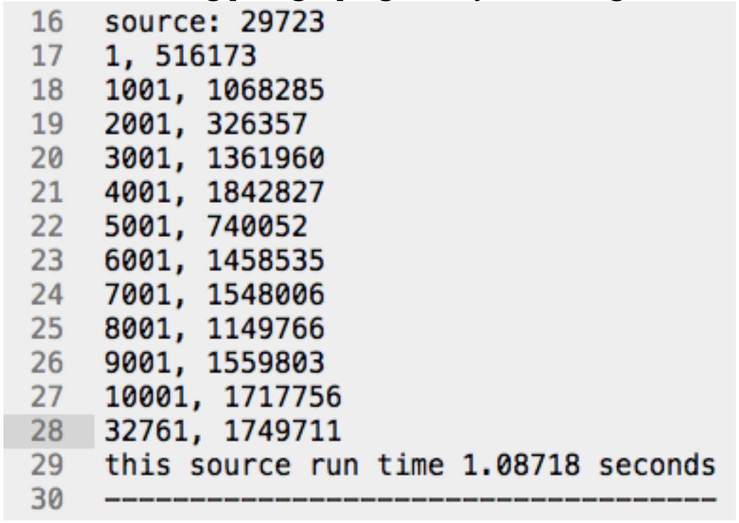
Figure 1, Assignment4RunAll contents

To see the pre-executed results, while in the *task2* directory, one can do:

1. *Bash$ cd results*

All my results are stored in this folder, and each sub-folder has the same name as the dataset. The exact procedure can be performed in task 1 folder for running and compiling the program.

All the numerical results in the *results* folder are identical to the original benchmark. Figure 2. Below prsents a result snapshot of one of the sources in Square 15. The following paragraph guides you through how to interpret the printouts.



```
16  source: 29723
17  1, 516173
18  1001, 1068285
19  2001, 326357
20  3001, 1361960
21  4001, 1842827
22  5001, 740052
23  6001, 1458535
24  7001, 1548006
25  8001, 1149766
26  9001, 1559803
27  10001, 1717756
28  32761, 1749711
29  this source run time 1.08718 seconds
30  ------------------------------------
```

Figure 2, Square 15, first source, result

In Figure 2, line 16 shows the current source vertex. Line 17 – 28 shows two unlabeled columns (my mistake here for forgetting to place label on top). The first column is the vertex numbers of the benchmark, and the second column is the distances results, which is identical to what was presented in the reference outputs. Line 29 shows the total run time for this source. Notice that the run time is much slower than the serial program. I will bring up more discussion on this behavior in the following paragraphs, especially in the performance section.

Task 1

My contribution in task 1 is simply implementing a serial program for the moores law and run a few test traces for verification purpose. The results has been prove to be matching, so does the program run time. Table 2 below debriefs what resides in the *task 1* folder. Since the major work focuses on the muti-threading program, I will move on the discuss the implementation choices in Task 2 section.

Folders/Files	Comments
Assignment4RunAll.sh	The qsub script for running either task 1 program. task 2 has its own qsub script.
Assignment4BuildAll.sh	The script to load module and run "make"
global.h	This file stores global constants and structs
Makefile	Make file for the task.
moores.c	Moores Law is implemented in this file
moores.h	Header file for moores.h
outputs/	Reference outputs folder provided by Dr.Sherman
results/	My results for the dataset I ran
task1.c	Main entry of the program, note that some of the code in this file are taken from Dr.Sherman
task1	Binary file for the task
***.o**	Object file from compilation

Table 2, task 1 directory

Task 2

In *task2* folder, the files' naming convention is the same as in *task 1* folder. The major difference comes from the moores law implementation, which uses OpenMP in task 2.

Shared Data Update

The shared data in my program include an end-queue counter, distance array , vertex queue, queue size, number of vertex, current source, and adjacent vertex list.

The end-queue counter increments when one of the thread fetchs a NULL in the queue, meaning that when the counter reachs 8 (assuming 8 threads in the pool), it is the end of the vertex queue, and therefore it reaches the end of the program. To make the increment atomic among threads, I use *#pragma omp atomic* clause as the simple increment is done via a "counter++" operation.

Vertex queue has the operations of enqueue and dequeue. In each enqueue operation (put_queue()), the queue size increment by 1, and the new vertex is appended to the end of the queue. The dequeue operation (get_queue()), reduces the queue size by 1 and removes the head of the queue (a.k.a, first element of the vertex array). That being said, both of the operations must be mutually exclusive. Only one is allowed to happen at the same time. Also since the operations changes the queue dimension, both of them are highly memory intensive, requiring reallocation of the entire memory chunk every time they are getting called upon. Therefore, the multi-threaded program is a few magnitudes slower than the serial program. Either way, to obtain the mutual exclusion between enqueue and dequeue, they must share a common critical region. To do that, I use *#pragma omp critical* clause for both operations without specifying the name for the critical region since this is the only critical region in my program.

Since the critical section handles the mutual exclusion, the shared queue size would not have collisions. The total number of vertex, current source and adjacent ertex list remains constant across the program, meaning no write operations are performed on these variables.

Termination

The termination of the program is determined by the end-of-queue counter as stated above. I am aware that just because the vertex queue is empty does not necessarily mean that the program is finished, since there may be ongoing computations that will add vertices to the queue. The counter is then acts as an flag. Once the counter reachs 8, meaning if all 8 dequeue(get_queue()) operations cannot get any vaild vertex, it is the end of the program. Since the dequeue and enqueue mutually exclusive, it is impossible to have a dequeue operation happening at the same time as enqueue, and therefore the vertex queue cannot be empty until the very end.

Correctness

It is not desirable to have multiple threads working on the same vertex, since such case will have multiple writes for the distance array. One may argue that the multiple writes can be handled by lock() and unlock(), yet it is hard to determine which is the latest write operation, and the values it writes are inter-dependent to each other (such as *newdist_vj < (*dist)[vj]) || ((*dist)[vj]==INF*). Therefore, the best way to handle the issue is to give each thread one vi, and each vi fetch vj in its adjacent list in sequential order.

Load Balance

Since each thread handles one entry in the vertex, it is easy to detect which of the threads fails to fetch a valid vertex, and therefore the failed threads stays idle while other threads is working on their vertex. In order to maintain the thread safe operation, I don't think it is a good idea to use the idle to keep polling the vertex queue for 2 reasons: 1. Vertex queue may change depends on the enqueue operation. 2. The enqueue and dequeue are protected in a common critical region, and breaking this scheme means break the whole thread safe semantics.

Performance

I am keeping my source code between the two tasks as close as possible for better comparison. In each task, I am keeping it on 1 process with 1 thread per node.

My *task1* performance is a lot slower than the reference outputs possibly due to two reasons: 1. My serial code is running on 1 process with 1 thread per process. 2. My enqueue and dequeue operations changes the queue size by reallocating the entire data chunk, which leads to heavy memory operation. One advantage of this implementation is that I can reduce the space complexity in contrast to using a very large constant queue size (N). I selected a few data points from *random* dataset to compare my serial and parallel program (table 3).

Dataset	Serial Run Time (seconds)	Parallel Run Time (seconds)
random_13	0.18082	0.81737
random_15	3.83267	8.89329
random_17	77.44162	122.71936
random_18	334.51530	504.15030

Table 3, performance comparsion

From table 3, I can see that the larger the data size the more likely to obtain the speedup in multi-threading program. The same can be said for all the dataset given. Note that using my program, some of the largest datasets (USA routes, largest datasets in each of long, random, and square) cannot be completed within 30 minutes in either serial or multi-threading program, and therefore torque job's 30 minutes limit is not long enough for these dataset. However, for all the smaller datasets, both of my programs guarantee the correctness.