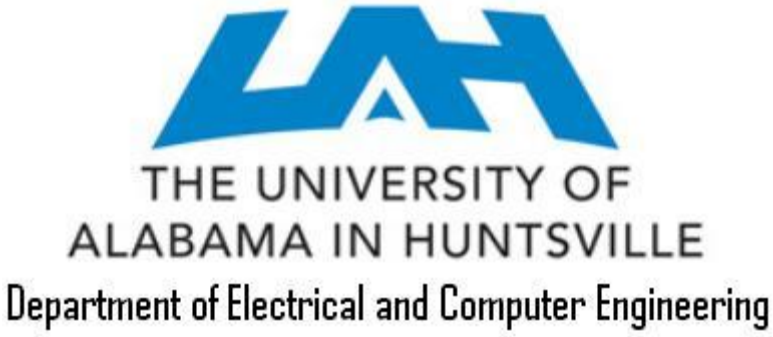

CPE 412/512

Fall Semester 2016

Exam 2 (Take-Home)

Name _Kyle Ray_____

INSTRUCTIONS: CPE 512 students must work all five (5) problems. CPE 412 students should work any four (4) problems clearly indicating which problem they will omit. In addition to electronic submissions on the UAH Canvas course administration system student are required to submit a complete hardcopy of this exam by its due date at the beginning of class on Monday November 13, 2017.

Statement of Compliance with UAH Academic Misconduct Policies

I _Kyle Ray_____ certify that I have worked independently of others on this test and the work that I am presenting is my own. I am familiar with the UAH academic misconduct policy as outlined in the UAH student handbook and have agreed to abide by the policies that are stated in this document.

Alternative Statement

I _____ am unwilling to sign the above statement of compliance because I cheated on the problems listed below:

_____
_____

Fully answer the following questions. Justify your answers by providing the page/section number reference from your text which supports your answer or by providing a complete citation of any external sources that you have used.

a) What OpenMP directive is used to provide mutual exclusion synchronization in sections of code? Give an example of its use and explain why it is needed.

b) Give the output of the following OpenMP code fragment if possible or discuss why this is not allowed. What is the effective scope of the sum variable? How could the answer be affected if the reduction clause is not used? Assume that there are 4 threads all together:

int i; double sum = 0.0; #pragma omp parallel for reduction(+:sum) num_threads(4) for (i=1; i <= 4; i++) sum = sum + i; cout << "The sum is " << sum << endl; c) For the following pthread code what is (are) the possible value(s) of num if there are no assumptions regarding the targeted system or the manner in which the pthread scheduler will operate? Justify your answer.

d) What is the difference between the ordered directive and the critical section directives in OpenMP? Give separate examples of the use of both constructs. Which constructs might be useful to support local synchronization? Which constructs are useful for or global synchronization? Explain.

e) What is the difference between the single and master section directives in OpenMP? Give separate examples of the use of both constructs.

f) Explain why static mapping of data blocks to processors may be bad for the Mandelbrot program that was discussed in one of the class lectures.

g) What is the basic idea of a Monte Carlo simulation. Describe how this technique can be used to solve the numerical integration problem? What makes this technique so attractive for parallel processing? What are the major issues that could affect its accuracy when Monte Carlo simulations are parallelized?

using namespace std; #include <iostream> #include <pthread.h> #include <stdlib.h> int num=1; void *thread1(void *dummy) { num = num + 1; } void *thread2(void *dummy) { num = num * 3; } void *thread3(void *dummy) { num = 123; } int main(int argc, char *argv[]) { pthread_t thread1_id,thread2_id,thread3_id; pthread_create(&thread1_id,NULL,thread1,NULL); pthread_create(&thread2_id,NULL,thread2,NULL); pthread_create(&thread3_id,NULL,thread3,NULL); cout << "num = " << num << endl; while(1); // infinite loop }

2. You have been provided with a sequential exhaustive search traveling salesman program that you are to parallelize using OpenMP, or pThreads. This sequential program which is named tsp_serial.cpp, can be found on the CPE 512/412 CanvasTM site (or can be copied from /cpe412/ exam2/tsp_serial.cpp on the UAH Jetson system).

a) Examine this program and modify it so that you can measure the execution time. Using the queuing system on the Jetson system, as discussed in class, record the

execution times for a 3,4,5,6,7,8,9,10,11,12, and 13 city tour. What is the time complexity of this algorithm? Develop an equation that can be used to estimate the execution time of the base sequential algorithm. Why do you think this problem is a challenging problem to solve, and inexact heuristics are utilized instead of exhaustively searching through all solutions as is done in this program?

b) Develop a general multi-threaded version of the sequential program in either pThreads or OpenMP that will effectively divide up the amount of work that is performed. Using a single node of the Jetson queuing system, measure the execution times for an 8,9,10,11,12, and 13 city tour on a 1, 2, 3, and 4 thread implementation. For each multi-threaded implementation show the speedup, efficiency, and cost as a function of the number of cities in the tour. 3. Expand the mm_mult_serial.cpp program on the CPE 512/412 CanvasTM site (or copy this program from /cpe412/exam2/mm_mult_serial.cpp on the UAH Jetson system) to create a hybrid multi-threaded/message passing implementation of a matrix/matrix multiplication program where the first matrix is of size lxm and the second matrix is of size mxn. Assume that the quantities being multiplied are of type float. Write the program in a general manner to allows the number of message passing processes and the number of threads per message passing process to be independently set by the user at run time. The program should be designed using the same row-wise decomposition method that was used in the third homework assignment. It should be written in a manner that will result in the total amount of computation to be divided as evenly as possible among the message-passing processes with the computation that is assigned to each messagepassing processes in turn being further divided as evenly as possible between the associated threads. You are to use a combination of MPI and either OpenMP or pThreads to complete this problem. The number of threads should be a command-line parameter while the number of message-passing processes should be set by the Jetson Queuing system when you specify the number of nodes that are to be employed. The l, m, and n dimensions should also be command line parameters that can be set at run time. In other words, if the executable is named mm_mult_hybrid then the syntax needed to execute the code should take on the general form shown below:

srun mm_mult_hybrid [No. Threads per Process] [dim_l] [dim_m] [dim_n])

where the number of MPI processes to be employed which is specified when the job is submitted to the Jetson queuing system.

a) Illustrate the correctness of your program for all possible combinations of thread numbers and number of MPI processes whose product is equal to 8 (i.e. 1 MPI process 8 threads per process, 2 MPI processes and 4 threads per process, 4 MPI process and 2 threads per process, and 1 MPI process and 8 threads per process) for cases where dim_l, dim_m, and dim_n take on distinct values.

b) Then perform a set of timing experiments under the same set of MPI process/thread combinations of part a (but with the output suppressed) for the cases where the dim_l=dim_m=dim_n=5,000. Record the execution time associated with each multi-threaded case and compare these times with that of the original serial program. What is the relative speedup and efficiency for each case. Is there a significant difference

between the various parallel implementations? If so give a possible explanation as to why the execution time was not the same for each case given that the total number of threads is the same in all cases and your code was designed to evenly distribute the workload among the threads.

4. Answer the following question for the code segment shown below assuming that we have a computing node that has 4 active cores that are dedicated to processing our problem and that were are utilizing OpenMP to parallelize a for-loop that initializes the upper triangle portion of a 100 × 100 matrix to the values returned by the function, init_element(x,y), which itself always executes in constant amount of time regardless of the values associated with its two arguments.

```
#pragma openmp parallel for schedule( ... )

for (i = 0; i < 99; i++) {

for (int j = i+1; j < 100; j++) {

a[i][j] = init_element(i,j);

}
```

} Notice that the arguments to the schedule() clause have been left undefined. Below are six example schedule clauses that could be used. Rank these clauses from slowest to fasted by closely examining the characteristics of this problem. To do this note that each iteration of the inner loop above does just one assignment and we can estimate the execution time by counting how many assignments each thread does. (Also note the total number of assignments the problem performs is exactly 4,950 assignments). For each schedule clause, estimate how long the parallelized loop will run. Explain how you arrived at your estimates.

schedule(static)

schedule(static, 10)

schedule(static, 1)

schedule(dynamic, 1)

schedule(dynamic, 10)

schedule(dynamic, 20)

5. Expand upon the bounded_buffer.cpp that is provided on Canvas (or copy this program from /cpe412/exam2/bounded_buffer.cpp on the Jetson system) to create two separate OpenMP implementations of the producer/consumer bounded buffer-problem discussed in class that utilizes a common shared memory (between threads) and counting semaphores to ensure proper synchronization. [Note you will also need to link this program with the util.o file that is also provided on Canvas or on the Jetson system at /cpe412/exam2/util.o] One implementation should use OpenMP's standard parallel and work sharing data parallel constructs to start up the separate consumer and

producer threads. The other implementation should utilize the OpenMP tasking model. Verify that both models function correctly and answer the following questions:

a) Carefully explain the characteristics of the producer and consumer bounded buffer problem? What are the major synchronization challenges that are involved.

b) Describe the function of the main, consumer, and producer module? What are the output files diary and con_? and prod_? reporting?

c) Execute both versions of the producer/consumer program with two producers, two consumers, number of messages of 100 and with a buffer size of 5. How many actual threads are generated when you run each version? Are they the same? How do they relate to the number of producers/consumers that are specified at run time?

d) How is it possible for the buffer size to be smaller than the number of data that is to be passed between producer and consumer? Explain how this is actually implemented in the program.

e) Experiment with different size buffers, messages, and consumer processes and producer processes. From the diary file and other files can you determine if the semaphore releasing strategy is fair? Explain your answer.

Answers:

1. a)

The critical directive is used for mutual exclusion.

#pragma omp critical (optional critical_section_name)

Usage:

```
// Some code
#pragma omp critical my_critical_section
{
    // Critical Section Code
}
```

b) Output: The sum is 10

The effective scope of the sum variable is shared. The answer will not be affected by removing the reduction clause. The reduction clause will create a private variable of sum for each thread which in turn will add the thread value of (i) to the private sum. Then at the end of the loop the reduction clause will add the private sums to the shared sum variable thus the reduction has just added another step. The for loop without the reduction clause will just add the value of (i) to the current shared value of sum, and because addition is commutative it will not matter which order this happens as we will get the same answer.

c) First, note that num is a global variable accessible by all. From my knowledge of pthreads, when a thread is created it will automatically start executing unless told to do otherwise which is not the case here. Without making any assumptions on the targeted system or the scheduler and considering that these functions are simple operations, with this setup I can only see num = 123 in all cases. Even with a simple addition and a multiplication I feel that threads 1 and 2 will finish very soon after creation, before thread 3 is created, which will leave the assignment from thread 3 to be the last operation on the global variable num, thus leaving it with the value 123.

d) **Ordered Directive** - The ordered construct will make it so the code for the structured block will execute sequentially.

Example.

```
#pragma omp parallel for schedule(static, 1)
{
    For (int I = 0; I < n; i++)
```

```
    {
        // Do something here first
        #pragma omp ordered
        {
            // Do these things in order
        }
        // Do some more stuff here
    }
    // Do something here first
}
```

Critical Directive – The critical directive will lock the structured block of code after the call to the directive until the current thread is finished, at which point it will release and allow the next thread in line to execute. This allows threads to execute code that might affect shared memory variables without entering in race conditions.

Example

```
// Some code
#pragma omp critical my_critical_section
{
    // Critical Section Code
}
```

The ordered directive is used to make sure that a block of statements is executed in sequential order, which could be a form of local synchronization.

The critical section is a good example of local synchronization as it will block a portion of code from the rest of the threads on the team until the locking thread is finished with the task at which point the previously locking thread can move on to the next set of instructions.

e) **Master Directive** – The master directive specifies that a piece of code should be executed by the master thread.

Example:

```
#pragma omp parallel num_threads(thread_count)
{
	// Do some stuff
	#pragma omp master
	{
		// Allow master to do something specific
		// Maybe output something to std out
	}
}
```

Single Directive – The single directive specifies that a piece of code should be executed by a single thread, but the thread is not required to be the master thread.

Example:

```
#pragma omp parallel num_threads(thread_count)
{
	// Do some stuff
	#pragma omp single
	{
		// Allow a single thread to do something specific
		// Note the thread doesn't have to be the master thread
	}
}
```

f) Because of the nature of the algorithm, different regions require a different number of iterations and time to complete. If this is statically mapped / scheduled, each iteration of the loop is assigned to a thread before it executes which can introduce load imbalance, where there can be some significant idle time and wasted resources when processing this problem i.e. if a thread finishes before the others then it has nothing left to do but wait. Better solution would be to use dynamic mapping so that a thread could request more work when it is finished with its current task.

g) The basic idea of Monte Carlo Simulation is to generate random inputs, belonging to a distribution, and evaluate the model's/program's risks and uncertainties.

In our text the author uses the Trapezoidal Rule for evaluating numerical integrals by approximating the region under a curve of a function as a trapezoid and finding the area for n trapezoids. This is a deterministic algorithm. The Monte Carlo method can also be implemented, non-deterministic algorithm, which would use number of samples in the distribution and evaluate the integral, this would be done however many times with however many samples and then the answers from the simulations would be averaged together to give the answer.

This technique is attractive to parallel programming because it is desired to thoroughly test a system; therefore, many test cases will need to be run which can be very time consuming. The use or parallel processing can divide the tasks up and hopefully complete the suite of simulations in a timelier manner.

The whole idea of the Monte Carlo Simulation is to test a model with random inputs to check risk and uncertainty. If using a pseudo random number generator, you could fall into the trap where each process/thread ends up with the same test variables and you just run the same test n times. While the results could be averaged to get a good assessment for that set of variables, there are many more combinations of tests that could have been run on the multiple processors/threads to give a better assessment of system under test.

2. a) Note: **See Appendix A for Source Code**

Time Complexity = O((n-1)!) approx. O(n!), the code is performing (n-1) permutations resulting in (n-1)! Tours.

Equation: Using line fit approximation -> $Y = (6e\text{-}9)e^{2x}$

This problem is one of the N = NP hard problems. It is a hard problem to solve because we are looking for the shortest path in a list of (n-1) paths. The only way to know for sure that we have the shortest path is by testing every path and comparing the costs. The number of paths that must be checked grows exponentially every time a city is added; therefore, finding a quick and correct solution to this problem is very difficult and one that many researchers and mathematicians have spent their entire careers trying to solve.

There have been some newer algorithms utilizing heuristics to quickly find a path, but this is exactly what happens the algorithm will find a short path, but it might not be the shortest path in the graph.

b) Note: See Appendix A for Source Code

Table 1: TSP Run Times Serially and Multi-threaded

NumberOfCities	Serial	NT = 1	NT = 2	NT = 3	NT = 4
3	0.00002100	0.00030867	0.00054250	0.00061733	0.00066841
4	0.00002600	0.00037083	0.00058716	0.00088074	0.00095524
5	0.00003800	0.00041466	0.00057000	0.00083925	0.00127965
6	0.00006900	0.00031175	0.00071074	0.00084258	0.00110049
7	0.00044900	0.00146774	0.00131216	0.00127966	0.00162181
8	0.00395100	0.01109520	0.00575688	0.00496014	0.00360471
9	0.03775100	0.08968780	0.04494200	0.03031980	0.02389500
10	0.41828700	0.84508500	0.43324100	0.28753200	0.22338700
11	4.99641000	9.39502000	4.84948000	3.14972000	2.37943000
12	64.78870000	113.70200000	56.69760000	37.77730000	28.88020000
13	905.26100000	1531.41000000	747.90600000	494.29500000	373.61600000

Table 2: TSP Relative Speed Up

RelativeSpeedup(NT = 1)	RelativeSpeedup(NT = 2)	RelativeSpeedup(NT = 3)	RelativeSpeedup(NT = 4)
0.06803492	0.03870989	0.03401752	0.03141794
0.07011299	0.04428072	0.02952050	0.02721832
0.09164046	0.06666725	0.04527874	0.02969562
0.22133332	0.09708137	0.08189143	0.06269934
0.30591249	0.34218388	0.35087445	0.27685117
0.35609994	0.68630925	0.79655010	1.09606598
0.42091567	0.83999377	1.24509396	1.57987027
0.49496441	0.96548341	1.45474938	1.87247691
0.53181473	1.03029809	1.58630291	2.09983483
0.56981144	1.14270622	1.71501669	2.24336050
0.59112909	1.21039409	1.83141848	2.42297171

Table 3: TSP Relative Efficiency

RelativeEfficiency(NT = 1)	RelativeEfficiency(NT = 2)	RelativeEfficiency(NT = 3)	RelativeEfficiency(NT = 4)
0.06803492	0.01935495	0.01133917	0.00785448
0.07011299	0.02214036	0.00984017	0.00680458
0.09164046	0.03333363	0.01509291	0.00742390
0.22133332	0.04854068	0.02729714	0.01567484
0.30591249	0.17109194	0.11695815	0.06921279
0.35609994	0.34315463	0.26551670	0.27401650
0.42091567	0.41999688	0.41503132	0.39496757
0.49496441	0.48274171	0.48491646	0.46811923
0.53181473	0.51514905	0.52876764	0.52495871
0.56981144	0.57135311	0.57167223	0.56084013
0.59112909	0.60519704	0.61047283	0.60574293

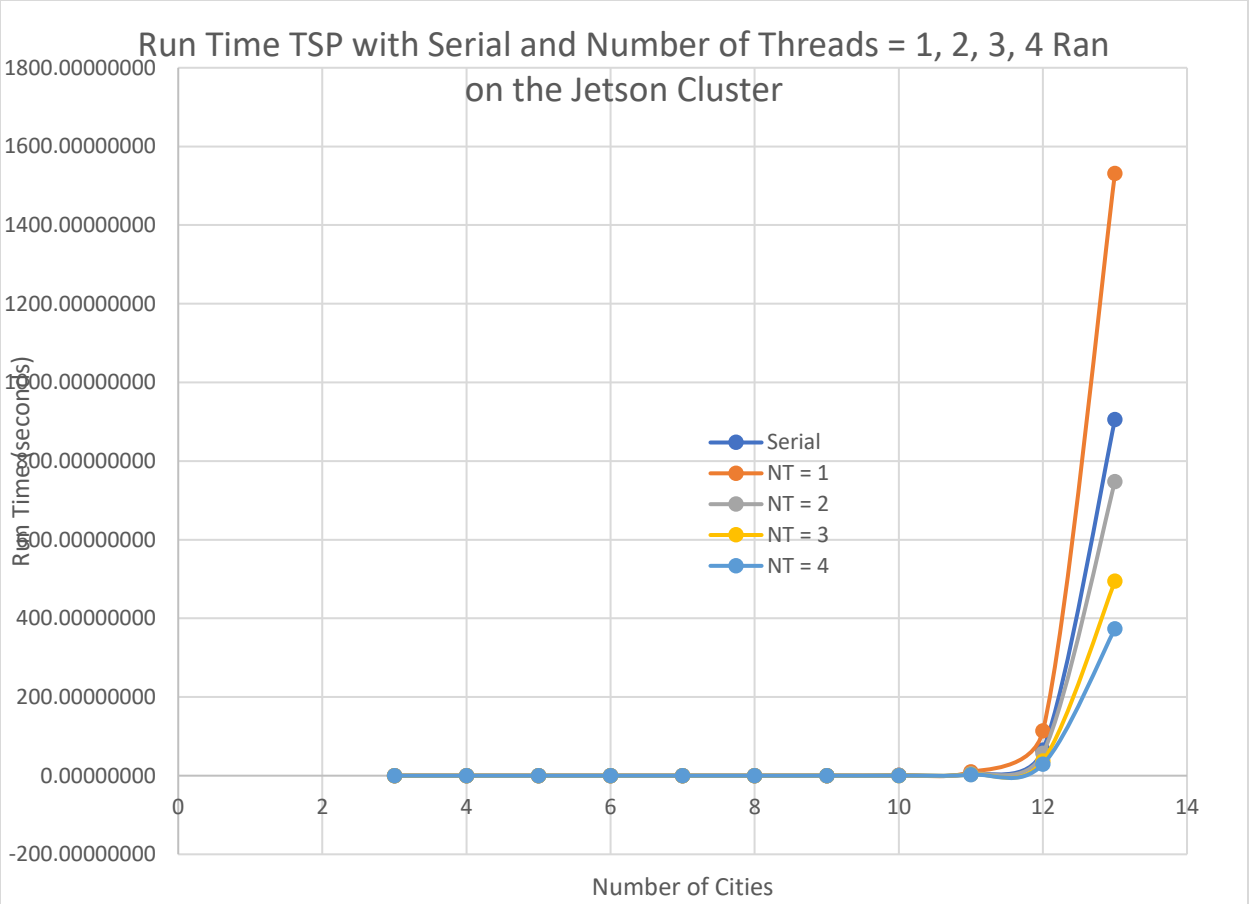

Figure 1: TSP Run Time Chart

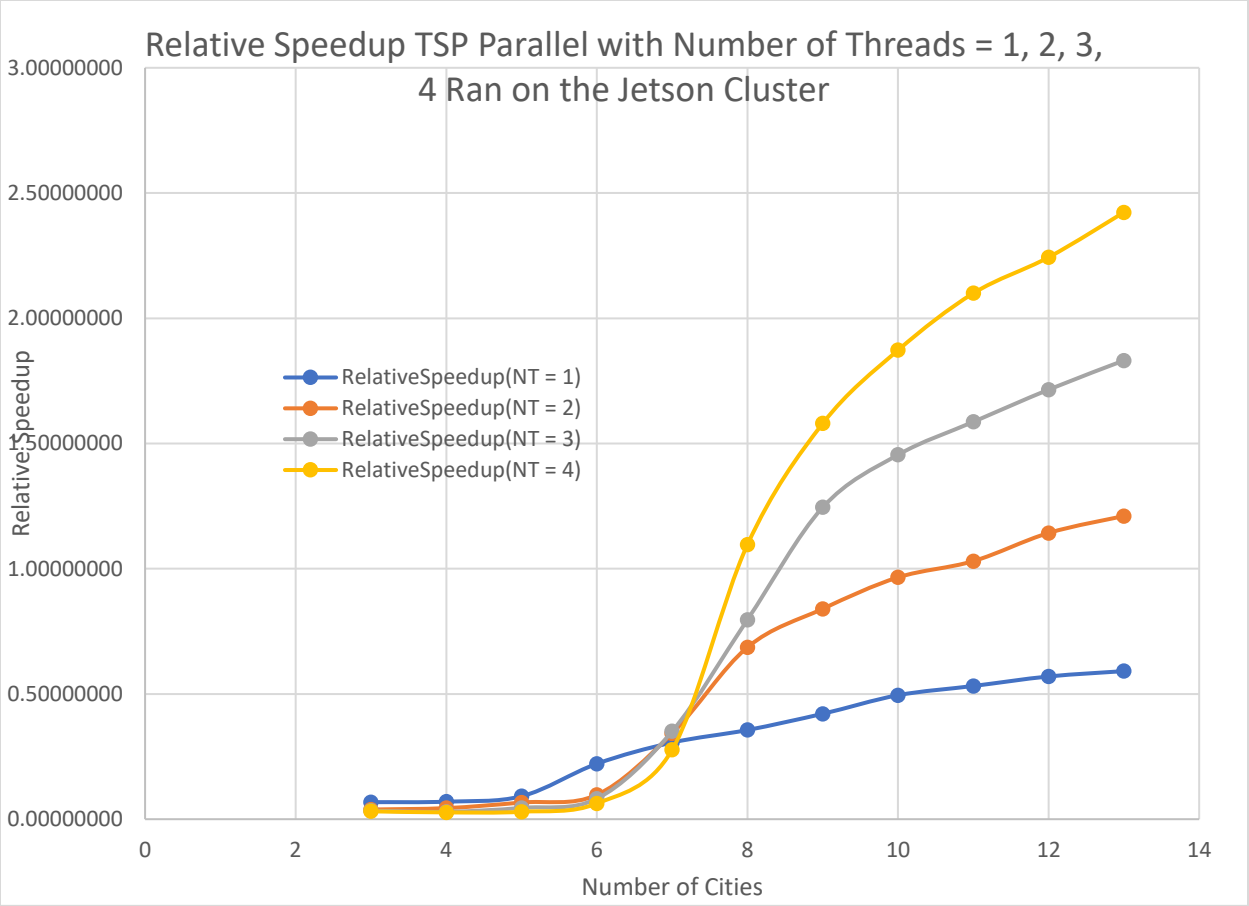

Figure 2: TSP Relative Speed Up

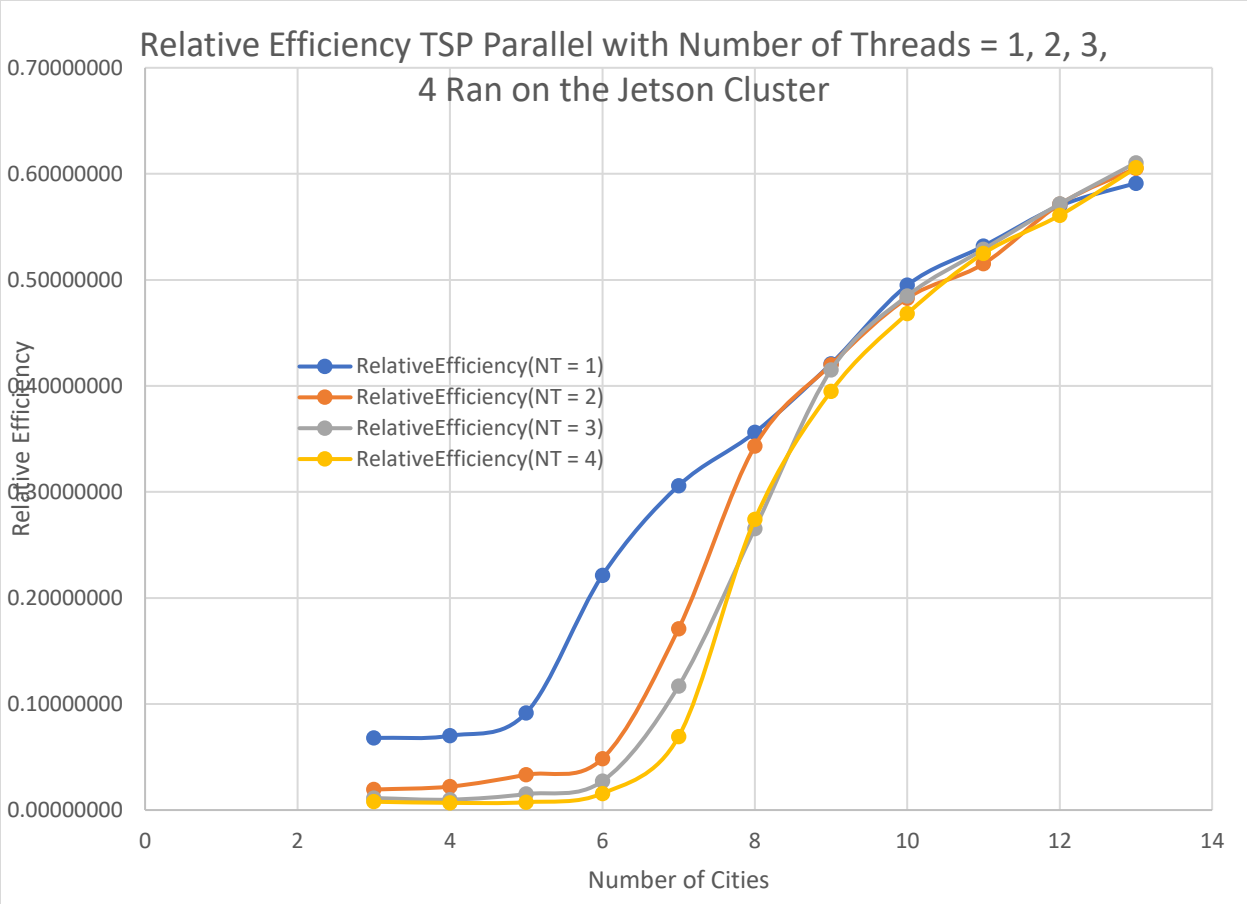

Figure 3: TSP Relative Efficiency

3. a) Note: **See Appendix B for Source Code**

Output Serial:

```
krr0010@jetson:~/jetson/Exam2/Problem3$ ./mm_mult_serial 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time=2e-06 seconds
```

Output Parallel MPI = 1, NT = 8

```
krr0010@jetson:~/jetson/Exam2/Problem3$ mpiexec -np 1 ./mm_mult_hybrid 8 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time = 0.0003619194 seconds
```

Output Parallel MPI = 2, NT = 4

```
krr0010@jetson:~/jetson/Exam2/Problem3$ mpiexec -np 2 ./mm_mult_hybrid 4 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709
```

```
B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time = 0.0049779415 seconds
```

Output Parallel MPI = 4, NT = 2

```
krr0010@jetson:~/jetson/Exam2/Problem3$ mpiexec -np 4 ./mm_mult_hybrid 2 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
7979.16 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332

time = 0.012290955 seconds
```

Output Parallel MPI = 8, NT = 1

```
krr0010@jetson:~/jetson/Exam2/Problem3$ mpiexec -np 8 ./mm_mult_hybrid 1 4 6 3
A matrix =
48.3962 65.3245 15.0385 72.383 25.8898 46.0265
15.4881 50.6507 6.74602 71.0055 12.2209 77.5441
61.5452 31.5127 46.8515 89.4849 70.0342 57.3195
75.4144 83.5553 91.7832 7.74197 40.0845 11.1709

B matrix =
26.5416 83.9488 86.5328
51.0444 65.3442 85.2683
76.9977 49.0015 46.6826
12.2581 99.9706 40.1026
58.6347 47.2069 4.06732
37.0919 22.9082 82.6622

C matrix =
9889.42 18581 17272.7
9889.42 14392.3 15281.2
14179 23086.6 18811.4
16193.3 19210.5 19332
```

```
time = 0.069700956 seconds
```

b) Timing Analysis

Table 4: Timing Analysis with Dim_l = Dim_M = Dim_N = 5000

Num MPI Procs	Num Threads	Serial Run Time	Parallel Run Time	RelativeSpeedup	RelativeEfficiency
1	8	5184.78550000	2316.45900000	2.23823754	0.27977969
2	4	5184.78550000	1173.34980000	4.41878926	0.55234866
4	2	5184.78550000	865.67243000	5.98931573	0.74866447
8	1	5184.78550000	736.24347000	7.04221594	0.88027699

Yes, there is a significant difference between the various parallel implementations. Looking at the table above it is easy to see that the case with 8 MPI processes and 1 thread executed with the most efficiency. In the cases with more OpenMP threads than MPI processes, the threads must compete for shared resources such as cores and memory. However, we can see in the cases where the number of MPI processors is greater we achieve a higher efficiency. This is more than likely because we have evenly distributed the problem to a cluster of machines that can now each evenly distribute the workload amongst their own shared memory threads, hopefully reducing the strain on shared resources. Being curious, I went back and ran the test with the number of threads set to 1 and just varied the number of MPI processes.

Table 5: Timing Analysis with NT = 1

Num MPI Procs	Num Threads	Serial Run Time	Parallel Run Time	RelativeSpeedup	RelativeEfficiency
1	1	5184.78550000	6146.44490000	0.84354218	0.84354218
2	1	5184.78550000	2994.07120000	1.73168410	0.86584205
4	1	5184.78550000	1480.06930000	3.50306942	0.87576735
8	1	5184.78550000	737.10272000	7.03400674	0.87925084

Comparing both tables we can see that the light weight OpenMP threads do help to reduce the overall execution time, but it doesn't seem that, with my implementation, they are as efficient as distributing it across multiple MPI processes.

4. Assuming init_element always executes in a constant time (x). The parallel for will divide the first for loop based on the scheduling method used.

What I know about the schedule clauses:

The static schedule clauses perform the round robin assignment of the iterations to each thread before the loops execute; whereas, the dynamic schedule clause the iterations are assigned while the loop is executing and after a thread is finished it can request another “chunk” from the run-time system. The dynamic schedule clause does incur some extra overhead for its capability.

What I know about this problem:

1.) The machine can utilize 4 cores / threads.
2.) The call to init_element(arg1, arg2) executes in constant time T no matter the arguments given.
3.) Parallelization is applied to the outer loop, so the 99 iterations will be scattered amongst the cores based on the scheduling routine.
4.) The inner loop number of iterations decreases as the outer index variable (i) increases. This will cause some iterations to take less time.

What I think will happen:

1.) Schedule(static)
    a. In the first case (schedule(static)) the default chunk size which will be total_iterations(99)/threat_count(4). This means that each thread will get 25 iterations with one thread getting only 24 for this case (25, 25, 25, 24). Because the inner loop is dependent on (i) the number of inner loop iterations decrease as (i) increases and because of the default static schedule this means that the total number of assignments will be (2175, 1550, 925, 300) for each thread respectively. Note thread 1 gets the “first” 25 iterations which have the largest number of inner loop iterations. That means thread 1 will execute for approximately 2175*T time while thread 4 will execute for 300*T time. That means that thread 4 will be sitting idle doing nothing for a while waiting for the other threads to finish. This is load imbalanced.

2.) Schedule(static, 10)
    a. In this case we are given the chunk size of 10; therefore, each thread will grab 10 iterations in a round robin fashion until no more are left thus leaving with this distribution (30, 29, 20, 20). Applying the same logic above the threads will arrive at this distribution for the total number of assignments (1635, 1335, 1090, 890). While this is still load imbalanced it is better than the first schedule clause.

3.) Schedule(static, 1)

a. In this case we are given the chunk size of 1; therefore, each thread will grab 1 iteration from the pool in a round robin fashion until no more are left thus leaving with a similar distribution as the first case (25, 25, 25, 24). Applying the same logic above the threads will arrive at this distribution for the total number of assignments (1275, 1250, 1225, 1200). This is much more balanced than the first two scheduling options.

4.) Schedule(dynamic) / Schedule(dynamic, 10) / Schedule(dynamic, 20)

a. The distribution for these clauses is left up to the run-time system and could be different every time the program is ran. It is good to note that dynamic scheduling is good for situations where the loop iterations might not execute in the same time. In this case because the number of inner loop iterations decreases as a function of the outer loop index (i), it is safe to say that if each assignment happens in a constant time T then these loops will all execute with different executions times at a delta of T. This would lead to believe that dynamic scheduling would be a good candidate for this problem. I feel that in the chunk size for the dynamic case the case where there is no chunk size and it defaults to 1 there will be more overhead at run time because the threads will be grabbing data more often. If this overhead time is much greater than the assignment then a larger chunk size might be desirable; however, if the overhead isn't much larger than the assignment time a smaller chunk size might be better to help avoid the load imbalancing we are using dynamic clause for. So, I feel that the without running a large series of timing experiments that the chunk size of 20 would be the fastest of the three, I'm assuming the overhead time is more than the assignment, in this case followed by 10 and 1.

Ranking:

Based on the above I would rank the scheduling clauses as follows: Note I feel that the dynamic scheduling would be very close

Slowest: Schedule(static), most load imbalanced

Schedule(static, 10)

Schedule(dynamic)

Schedule(dynamic,10),

Schedule(dynamic, 20)

Fastest: Schedule(static, 1)

5. a) Note: **See Appendix C for Source Code**
The problem consists of a producer, consumer, and a fixed sized buffer or queue. The idea is that the producer will produce data and place it in the queue. The consumer will then read from the queue and consume the data. It's desired to have the producer and consumer be working in parallel, so the producer is producing, and the consumer is consuming at the same time. There are a few rules however, the producer can only put data into the queue as long as it isn't full, and the consumer can only read from the queue if it's not empty otherwise there will be wasted work. This is where the synchronization problems occur, especially with multiple producers and consumers. All producers need to have a way to know if the buffer is full so that they don't try and cram data into the queue. Also, the consumers need to know when data is available so that they can consume it right then. Counting semaphores as well as mutual exclusion locks are necessary to help solve this problem.

b) Main – The main module is responsible for initialization and kicking off the producer and consumer modules. Main retrieves the input data from the user regarding the number of producers/consumers to use as well as the number of data items to be sent and how big the buffer will be. It then proceeds to initialize the shared buffer, the mutual exclusion lock (used for placing / reading items on the shared buffer safely), initializing the counting semaphores full and empty, and cleaning up when finished.

Consumer – Sets up a diary for logging then proceeds to read data from the shared buffer when it is available and put it through the consume_item method from util.o.

Producer – Sets up a diary file for logging then proceeds to generate data, via the produce_next_data_item method from the util.o object, to place on the shared buffer if it's not full.

Diary – Reports all interactions from the producer and consumer modules as they happen. This is useful to see when and where producers are placing data and consumers are consuming data.

Con_ - This is the diary file written by each individual consumer (con_id) displaying which producers' data it is reading, which produced data item, and in what index it retrieved it from the shared buffer.

Prod_ - This is the diary file written by each individual producer (prod_id) displaying which item it produced and what index it is placed in the shared buffer.

c) Four threads were generated in each test scenario. The number of threads was the same. The number of threads is directly related to the number of producers and consumers that are specified, in this case two threads for producer and two threads for consumer = four threads total.

d) It is possible because this program is utilizing counting semaphores to block the producers access to the shared buffer until there is a spot open for the data. The consumer will alert the group of producer threads that there is a spot open so that the

producers can write to the buffer. This makes sure that we aren't reading and writing to memory that we shouldn't be.

e) It seems to be fair, the producers can grab the lock and put their data on the queue and producers that have been waiting longer, producers with a higher ID as they were spawned a bit later, seem to get a change to produce and place their data in a timely manner. This also seems to hold true for the consumers.

Appendix A: Traveling Salesman Problem Source Code

```
// Travelling Salesman Program -- Serial Version
// B. Earl Wells -- November 2017
//
// Travelling Salesman Problem: "Given a specified number of "cities"
// along with the cost of travel between each pair of them, find the
// cheapest way of visiting all the cities and returning to the first
// city visited."
//
// For an n city tour this program examines all (n-1)! tours and
// returns a tour with the least travel cost.
//
//
// Edits:
// Parallelized this program with the use of the OpenMP library
// Kyle Ray
// Exam 2
// CPE 512
// November 20, 2017
// Note: (Exam 2 Edit) comment placed before each change to make it easier to find.
//
// Experimented with parallelizing the computational for loops.  When applying this to
some
// of the assignment and small arithmetic loops, the overhead of forking and joining
num_threads
// seemed to cost more than the actual operations computing serially.
//

using namespace std;
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <sys/time.h>
//#include <time.h>

// Exam 2 Edit
// Include the OpenMP library for function calls
#ifdef _OPENMP
  #include <omp.h>
#endif

// Exam 2 Edit
// Making this global so that I don't have to change every function call
// to have it available, bad practice but using it for convenience.
unsigned int num_threads;

#define TIMER_CLEAR     (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec = 0)
#define TIMER_START     gettimeofday(&tv1, (struct timezone*)0)
#define TIMER_ELAPSED   ((long long) (tv3.tv_usec)+((long long) (tv3.tv_sec)*1000000))
#define TIMER_STOP      {gettimeofday(&tv2, (struct
timezone*)0);timersub(&tv2,&tv1,&tv3);}
//struct timeval tv1,tv2,tv3;

#define MX_CITIES 30       // maximum number of cities in tour
#define TRUE 1
#define FALSE 0
#define SEED (long) 178937 // random number seed
#define MAX_INT 0x7fffffff // maximum integer possible

// Defines so that I can compile the code in visual studio
//#define srand48(s) srand(s)
//#define drand48() (((double)rand())/((double)RAND_MAX))
```

```
// generate randomly a cost matrix that represents the cost of travel
// between each two cities. This generates a nonsymetric cost matrix
// which means the cost of travel between any two cities may be
// depend on which city is the source and which is the destination
void fill_cost_matrix(int cost_matrix[][MX_CITIES],int num_cities) {
   int i,j;
   srand48(SEED);
   if (num_cities<=MX_CITIES) {
      for (i=0;i<num_cities;i++) {
         for (j=0;j<num_cities;j++) {
            if (i==j) cost_matrix[i][j]=0;
            else cost_matrix[i][j]= (int) (drand48()*100);
         }
      }
   }
   else {
      cout << "Error: Too many cities -- increase MX_CITIES parameter" << endl;
      cout << " and recompile" << endl;
      exit(1);
   }
}

// This routine outputs to the screen the city cost matrix that has
// been generated
void print_cost_matrix(int cost_matrix[][MX_CITIES],int num_cities) {
   int i,j;
   for (i=0;i<num_cities;i++) {
      for (j=0;j<num_cities;j++) {
         cout << setw(5) << cost_matrix[i][j];
      }
      cout << endl;
   }
   cout << endl;
}

// This routine outputs the City visit order from left to right
// assuming that we are always starting our tour with City 0.
// Note: there is no loss of generality in this since any tour
// order can be rotated to make any of the cities the start and
// end point.
void print_city_visit_order(unsigned int city_order[],unsigned int num_cities_m1) {
   int i;
   cout << "[City 0]->";
   for (i=0;i<num_cities_m1;i++) {
      cout << "[City " << city_order[i]+1 << "]->";
   }
   cout << "[City 0]" << endl;
}

// routine that computes the cost of each tour where the tour is
// described in the city_order array that should have a valid
// city order premutation of the n-1 remaining cities after the
// first city, City 0, is assumed as the starting point. The
// tour starts at City 0 and goes to the first city in the city_order
// array. The tour is then processed in the sequence dictated by
// the city_order permutation after which the tour proceeds from the
// last city in this permutation back to City 0.
// Note: there is no loss of generality in this since any tour
// order can be rotated to make any of the cities the start and
// end point.
int tour_cost(unsigned int num_cities,unsigned int *city_order,
              int cost_matrix[MX_CITIES][MX_CITIES]) {
```

```
   int i,last_city,cost;
   last_city=city_order[0]+1;
   cost = cost_matrix[0][last_city];

   // Exam 2 Edit
   //#pragma omp parallel for num_threads(num_threads)
   for (i=1;i<num_cities-1;i++) {
      cost += cost_matrix[last_city][city_order[i]+1];
      last_city = city_order[i]+1;
   }
   cost += cost_matrix[last_city][0];
   return cost;
}

// Routine to save the current city_order permutation to another
// data structure so that it can be displayed later by the
// print_city_visit_order routine
void save_order (unsigned int *city_order_sv,unsigned int num_cities,
                 unsigned int * city_order) {
   int i;

   // Exam 2 Edit
   //#pragma omp parallel for num_threads(num_threads)
   for (i=0;i<num_cities;i++) {
      city_order_sv[i]=city_order[i];
   }
}

// data structure that is used to store the best city visit order
unsigned int best_order[MX_CITIES],best_cost=MAX_INT; //set to worst value poss.

unsigned long int num_perms(unsigned int num) {
   unsigned long int nm_perms = 1;

   // Exam 2 Edit
   //#pragma omp parallel for
   for (unsigned int i = 1; i <= num; i++) {
      nm_perms *= (unsigned long int) i;
   }
   return nm_perms;
}

// routine to determine a unique permutation ordering that is
// associated with the specified permutation number assuming
// the specified number of elements that are to be permuted
// i.e. num_symbols.
// See https://en.wikipedia.org/wiki/Factorial_number_system
// for more information concerning the method that was employed
void permutation(unsigned int *perm, unsigned long int perm_num,
                 unsigned int num_symbols) {
   unsigned int factoradic[MX_CITIES];
   for (unsigned int i = 1; i <= num_symbols; i++) {
      factoradic[i-1] = perm_num%(long) i;
      perm_num /= (long) i;
   }

   // compute lexical order
   {
      int perm_mask[MX_CITIES];
      for (int i = 0; i<num_symbols; i++) {
         perm_mask[i]=0;
      }
```

```
      // Exam 2 Edit
      #pragma omp parallel for num_threads(num_threads)
      for (int i = num_symbols-1; i>=0; i--) {
         int lex_cnt = -1;
         for (int j = 0; j<num_symbols;j++) {
            if (perm_mask[j]==0) lex_cnt++;
            if (lex_cnt==factoradic[i]) {
               perm_mask[j]=1;
               perm[i] = j;
               break;
            }
         }
      }
   }
}

void tour_search(unsigned int nm_cities,int city_cost_matrix[][MX_CITIES]) {
   // create a permutation matrix that list the cities in tour visit order
   // This one dimensional matrix is called perm_order and it holds the
   // index number of the nm_cities in the tour minus one because one
   // city is assumed to be held in place with the other city order being
   // permuted around it. Note that this program assumes that the city
   // held constant is city 0, and this city starts and ends the tour cycle.
   // This means that city 0 will preceed the city that is placed in slot 0
   // of the perm_order matrix and this city will be the final destination
   // city that is present in slot nm_cities - 2
   unsigned long int total_tours=num_perms(nm_cities-1);

   // Exam 2 Edit
   #pragma omp parallel for num_threads(num_threads)
   for (long int tour_nm = 0; tour_nm<total_tours; tour_nm++)
   //for (unsigned long int tour_nm=0;tour_nm<total_tours;tour_nm++)
   {
      unsigned int perm_order[MX_CITIES];
      // get next permutation
      permutation(perm_order, (unsigned long int)tour_nm, nm_cities-1);

      int tour_cst=tour_cost(nm_cities,perm_order,city_cost_matrix);
      if (tour_cst<best_cost) {
         save_order (best_order,nm_cities-1,perm_order);
         best_cost = tour_cst;
      }
      // uncomment to view all city tours
      // print_city_visit_order(perm_order,nm_cities-1);
      // cout << "tour cost = " << tour_cst << endl;
   }
}

// routine to obtain the number of cities from the user
// via the command line or by prompting the user for input
unsigned int get_city_number(int argc,char *argv[]) {
   // Exam 2 Edit
   unsigned int num_cities;
   if (argc == 3) {
     num_threads = atoi(argv[1]);
     num_cities = atoi(argv[2]);
   }
   else {
      if (argc==1) {
         // input number of threads and cities
         cout << "Enter number of threads:" << endl;
         cin >> num_threads;
         cout << endl;
```

```
         cout << "Enter number of cities:" << endl;
         cin >> num_cities;
         cout << endl;
      }
      else {
         cout << "usage: tsp_serial <number of cities>" << endl;
         exit(1);
      }
   }
   if ((num_cities<2) || (num_cities>MX_CITIES)) {
      cout << "Error: Number of Cities too large or too small" << endl;
      exit(1);
   }
}

int main(int argc, char *argv[]) {

   int city_cost_matrix[MX_CITIES][MX_CITIES];
   unsigned int num_cities;

   // get number of cities from the user
   // Exam 2 Edit
   num_cities = get_city_number(argc,argv);

   // fill city cost matrix
   fill_cost_matrix(city_cost_matrix,num_cities);

   // print city cost matrix
   print_cost_matrix(city_cost_matrix,num_cities);

   //TIMER_CLEAR;
   //TIMER_START;

   // Exam 2 Edit
#ifdef _OPENMP
   double startTime = omp_get_wtime();
#endif

   // search through all possible orderings of the
   // cities
   tour_search(num_cities,city_cost_matrix);

   //TIMER_STOP;
   // Exam 2 Edit
#ifdef _OPENMP
   double endTime = omp_get_wtime();
#endif

   // print city visit order
   cout << "Best City Tour:"  << endl;
   print_city_visit_order(best_order,num_cities-1);

   cout << "tour cost = "  <<
      tour_cost(num_cities,best_order,city_cost_matrix)
     << endl;
   //cout << num_cities << "  " <<  TIMER_ELAPSED/1000000.0 << endl;

   // Exam 2 Edit
#ifdef _OPENMP
   cout << num_cities << " " << endTime - startTime << " seconds" << endl;
#endif
   return 0;
```

```
}
```

Appendix B: Matrix Multiply Hybrid Source Code

```
/*************************************************************/
/* Matrix Matrix Multiplication Program Example -- serial version */
/* September 2016 -- B. Earl Wells -- University of Alabama       */
/*                                    in Huntsville               */
/*************************************************************/
// mm_mult_serial.cpp
// compilation:
//   gnu compiler
//      g++ mm_mult_serial.cpp -o mm_mult_serial -O3 -lm
// Note: to compile a parallel MPI program version which is named
//   mm_mult_mpi.cpp
//   then execute the following command
//      gnu compiler
//         mpic++ mm_mult_mpi.cpp -o mm_mult_MPI_gnu -lm  -O3
/*
   This program is designed to perform matrix matrix multiplication
   A x B = C, where A is an lxm matrix, B is a m x n matrix and
   C is a l x n matrix. The program is designed to be a template
   serial program that can be expanded into a parallel multiprocess
   and/or a multi-threaded program.

   The program randomly assigns the elements of the A and B matrix
   with values between 0 and a MAX_VALUE. It then multiples the
   two matrices with the result being placed in the C matrix.
   The program prints out the A, B, and C matrices.

   The program is executed using one or three command line parameters.
   These parameters represent the dimension of the matrices. If only
   one parameter is used then then it is assumed that square matrices are
   to be created and multiplied together that have the specified
   dimension. In cases where three command line parameters are entered
   then the first parameter is the l dimension, the second the m, and
   the third is the n dimension.

   To execute:
   mm_mult_serial [l_parameter] <m_parameter n_parameter>

   Edits:
   Kyle Ray
   Exam 2
   CPE 512

   Added MPI process parallelism to distribute the problem evenly
   to multiple heavy weight processes, number of processes is given
   at run time by the user.  Also, added OpenMP parallel statements
   to divide the work of each process among a number of threads, number
   of threads available is specified by the user at runtime

   Compie:
   mpic++ -o mm_mult_hybrid mm_mult_hybrid.cpp -fopenmp

   Execute:
   mpiexec -np [num_MPI_processes] mm_mult_hybrid [num_threads_per_process]
[l_parameter] <m_parameter n_parameter>
*/

using namespace std;
#include <iostream>
#include <iomanip>
```

```
#include <sstream>
#include <stdlib.h>
#include <string.h>
//#include <sys/time.h>
#include <time.h>
#include <mpi.h>
#ifdef _OPENMP
#include <omp.h>
#endif

#define MX_SZ 320
#define SEED 2397           /* random number seed */
#define MAX_VALUE  100.0    /* maximum size of array elements A, and B */

/* copied from mpbench */
#define TIMER_CLEAR     (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec = 0)
#define TIMER_START     gettimeofday(&tv1, (struct timezone*)0)
#define TIMER_ELAPSED   ((tv2.tv_usec-tv1.tv_usec)+((tv2.tv_sec-tv1.tv_sec)*1000000))
#define TIMER_STOP      gettimeofday(&tv2, (struct timezone*)0)
//struct timeval tv1,tv2;

// Defines so that I can compile the code in visual studio
//#define srand48(s) srand(s)
//#define drand48() (((double)rand())/((double)RAND_MAX))


/*
This declaration facilitates the creation of a two dimensional
dynamically allocated arrays (i.e. the lxm A array, the mxn B
array, and the lxn C array).  It allows pointer arithmetic to
be applied to a single data stream that can be dynamically allocated.
To address the element at row x, and column y you would use the
following notation:  A(x,y),B(x,y), or C(x,y), respectively.
Note that this differs from the normal C notation if A were a
two dimensional array of A[x][y] but is still very descriptive
of the data structure.
*/
float *a, *b, *c;
float *group_a, *group_c;
#define A(i,j) *(a+i*dim_m+j)
#define B(i,j) *(b+i*dim_n+j)
#define C(i,j) *(c+i*dim_n+j)

/*
   Routine to retrieve the data size of the numbers array from the
   command line or by prompting the user for the information
*/
void get_index_size(int argc, char *argv[], int *dim_l, int *dim_m, int *dim_n, int*
thread_count, int rank) {
  if (argc != 3 && argc != 5) {
    if (rank == 0)
    {
      cout << "usage:  mm_mult_serial num_threads [l_dimension] <m_dimension
n_dimmension>"
        << endl;
      MPI_Finalize();
      exit(1);
    }
  }
  else {
    if (argc == 3) {
          *thread_count = atoi(argv[1]);
      *dim_l = *dim_n = *dim_m = atoi(argv[2]);
```

```
    }
    else {
          *thread_count = atoi(argv[1]);
      *dim_l = atoi(argv[2]);
      *dim_m = atoi(argv[3]);
      *dim_n = atoi(argv[4]);
    }
  }
  if (rank == 0)
  {
    if (*dim_l <= 0 || *dim_n <= 0 || *dim_m <= 0 || *thread_count < 1) {
      cout << "Error: number of rows and/or columns must be greater than 0 and the
thread count must be greater than or equal to 1"
        << endl;
      MPI_Finalize();
      exit(1);
    }
  }
}

/*
   Routine that fills the number matrix with Random Data with values
   between 0 and MAX_VALUE
   This simulates in some way what might happen if there was a
   single sequential data acquisition source such as a single file
*/
void fill_matrix(float *array, int dim_m, int dim_n)
{
  int i, j;
  for (i = 0; i < dim_m; i++) {
    for (j = 0; j < dim_n; j++) {
      array[i*dim_n + j] = drand48()*MAX_VALUE;
    }
  }
}

/*
   Routine that outputs the matrices to the screen
*/
void print_matrix(float *array, int dim_m, int dim_n)
{
  int i, j;
  for (i = 0; i < dim_m; i++) {
    for (j = 0; j < dim_n; j++) {
      cout << array[i*dim_n + j] << " ";
    }
    cout << endl;
  }
}

// Routine to get the base number of multiplies for a ceratin process
int getBaseMult(int num_mults, int numtasks, int rank)
{
  // Calculate the base number of multiplies for each task
  int base = num_mults / numtasks;

  // Calculate the extra if it is not an even distribution
  int extra = num_mults % numtasks;

  // If there are any extra assign them to the first tasks up to rank of extra
  if (extra != 0)
  {
```

```
    // If rank is less than the number of extra items, then this process gets an extra
multiply to process
    if (rank < extra)
      base = (num_mults / numtasks) + 1;
  }

  return base;
}

/* ONE-TO-ALL SCATTER ROUTINE
Routine to divide the number of rows to each process based on the number
of multiplies that each process must perform.  Each process will receive the
row up to number of multiplies in a sequential order.  This method will also keep up
with the current column slider and pass that to the process so it knows where to
start it's set of multiplications.
The partial arrays for each process are stored in the group_a array.
*/
void scatter(float* a, float* b, float *group_a, int root, int rank, int numtasks,
  int dim_l, int dim_m, int dim_n, int* start_column)
{
  MPI_Status status;
  int type = 234;

  // How many multiplies will the entire operation require?
  int num_mults = 0;
  if (dim_l == 1)
    num_mults = dim_n;
  else if (dim_n == 1)
    num_mults = dim_l;
  else
    num_mults = dim_l * dim_n;

  // Variables to keep up with what row and column we are reading from
  int begin_row = 0;
  int begin_column = 0;

  // Root process does all of the work
  if (rank == root)
  {
    // Loop over the MPI tasks
    for (int mpi_task = 0; mpi_task < numtasks; mpi_task++)
    {
      // Get the number of multiplies
      int base = getBaseMult(num_mults, numtasks, mpi_task);

      // Each task gets at least one row to work with
      int num_rows = 1;

      // Variables to keep up with navigating the matrices
      int count = base;
      int curr_col = begin_column;

      // Calculate the number of rows that we need to send to the process.
      while (count > 0)
      {
        // Get the current distance from the end of dim_n
        int diff = dim_n - curr_col;

        // If we have more multiplies to process then left for this row, we must add
another row
        if (count > diff)
        {
          num_rows++;
```

```
          count -= diff;
          curr_col = 0;
          continue;
        }

        count -= dim_n;
      }

      // If the current loop iteration doesn't correspond to the root then we must
send the column and number of rows to the process
      if (mpi_task != root)
      {
        MPI_Send(&begin_column, 1, MPI_INT, mpi_task, type, MPI_COMM_WORLD); // send
the column slider location
        MPI_Send(&num_rows, 1, MPI_INT, mpi_task, type, MPI_COMM_WORLD); // send the
number of rows for correct sizing
      }

      // Local Buffer variables
      float* local_a = new float[num_rows*dim_m];

      // Reset our column check and multiply count
      curr_col = begin_column;
      count = base;

      // Row assignment
      for (int r = 0; r < num_rows; r++)
      {
        // Fill up the local matrix with values from the main A matrix
        for (int t = 0; t < dim_m; t++)
        {
          // If this is the root, go ahead and store it in the buffer
          if (mpi_task == root)
          {
            group_a[r*dim_m + t] = a[begin_row*dim_m + t];
          }
          else // Store it in the local that will be sent to the other processes
          {
            local_a[r*dim_m + t] = a[begin_row*dim_m + t];
          }
        }

        // Calculate the distance from the end of dim_n for this set of row
multiplications
        int diff = dim_n - curr_col;

        // If we still have more to process we must add the next row to this processes
variables
        if (diff <= count)
        {
          begin_row++;
          count -= diff;
          curr_col = 0;
        }
      }

      // Column Assignment
      // This logic will keep up with what column each process should start performing
calculations
      // Base is the updated base number of multiplies each process must perform
      int temp_column = base % dim_n; // Get the leftover after applying number of
multiplies
      begin_column += temp_column; // Update the current column index
```

```
      if (begin_column >= dim_n) // Account for wrapping around dim_n
        begin_column = begin_column % dim_n;

      // Send the data to the other processes
      if (mpi_task != root)
      {
        MPI_Send(local_a, num_rows*dim_m, MPI_FLOAT, mpi_task, type, MPI_COMM_WORLD);
      }

      delete[] local_a;
    }
  }
  else
  {
    // Calculate the base number of multiplies for each task
    int base = getBaseMult(num_mults, numtasks, rank);

    // I must send the number of rows as well
    int num_rows = 0;

    // Receive smaller arrays as well as the starting dim_n column from the root
process
    MPI_Recv(start_column, 1, MPI_INT, root, type, MPI_COMM_WORLD, &status);
    MPI_Recv(&num_rows, 1, MPI_INT, root, type, MPI_COMM_WORLD, &status);
    MPI_Recv(group_a, num_rows*dim_m, MPI_FLOAT, root, type, MPI_COMM_WORLD, &status);
  }
}

// All to one gather routine
// Each process will send their calculated sub matrix back to the root process
void gather(float* c, float* group_c, int num_mults, int root, int rank, int numtasks,
int dim_l, int dim_n)
{
  MPI_Status status;
  int type = 123;

  // Calculate the base number of multiplies for each task
  int base = getBaseMult(num_mults, numtasks, rank);

  if (rank == root)
  {
    int curr_ind = 0;
    // Piece back together the matrix
    for (int mpi_task = 0; mpi_task < numtasks; mpi_task++)
    {
      if (mpi_task == root)
      {
        for (int i = 0; i < base; i++)
        {
          // Copy what the root has
          c[curr_ind] = group_c[i];
          curr_ind++;
        }
      }
      else
      {
        // Receive from the processes

        // Calculate the base number of multiplies for each task
        int base = getBaseMult(num_mults, numtasks, mpi_task);

        float* temp = new float[base];
```

```
        MPI_Recv(temp, base, MPI_FLOAT, mpi_task, type, MPI_COMM_WORLD, &status);

        for (int i = 0; i < base; i++)
        {
          //cout << "Group_c Item " << temp[i] << endl;
          c[curr_ind] = temp[i];
          curr_ind++;
        }

        delete[] temp;
      }
    }
  }
  else
  {
    // Send the matrix to the root
    MPI_Send(group_c, base, MPI_FLOAT, root, type, MPI_COMM_WORLD);
  }
}

/*
   MAIN ROUTINE: summation of a number list
*/

int main(int argc, char *argv[])
{
  float dot_prod;
  int dim_l, dim_n, dim_m;
  int i, j, k;

  int num_mults, group_size, num_group;
  int numtasks, rank, num;
  int start_column;
  int thread_count;
  double start, finish;
  MPI_Status status;

  // Main Routine

  MPI_Init(&argc, &argv); // initalize MPI environment
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks); // get total number of MPI processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get unique task id number


  // get matrix sizes
  get_index_size(argc, argv, &dim_l, &dim_m, &dim_n, &thread_count, rank);

  // The root process fills the matrices and then passes them to the othe processes
  if (rank == 0)
  {
    // dynamically allocate from heap the numbers in the memory space
    // for the a,b, and c matrices
    a = new (nothrow) float[dim_l*dim_m];
    b = new (nothrow) float[dim_m*dim_n];
    c = new (nothrow) float[dim_l*dim_n];
    if (a == 0 || b == 0 || c == 0)
    {
      cout << "ERROR:  Insufficient Memory 1" << endl;
      MPI_Abort(MPI_COMM_WORLD, 1);
    }


    /*
```

```
     initialize numbers matrix with random data
   */
   srand48(SEED);
   fill_matrix(a, dim_l, dim_m);
   fill_matrix(b, dim_m, dim_n);

   /*
     output numbers matrix
   */
   cout << "A matrix =" << endl;
   print_matrix(a, dim_l, dim_m);
   cout << endl;

   cout << "B matrix =" << endl;
   print_matrix(b, dim_m, dim_n);
   cout << endl;
}
else
{
  b = new (nothrow) float[dim_m*dim_n];
}

if (rank == 0)
  start = MPI_Wtime();

MPI_Bcast(b, dim_m*dim_n, MPI_FLOAT, 0, MPI_COMM_WORLD);

// broad cast the data size, which is really the number of multiplies
if (dim_l == 1)
  num_mults = dim_n;
else if (dim_n == 1)
  num_mults = dim_l;
else
  num_mults = dim_l * dim_n;

int base = getBaseMult(num_mults, numtasks, rank);

// Each process has a local array set for local calculations
group_a = new (nothrow) float[dim_l*dim_m];
group_c = new (nothrow) float[dim_l*dim_n];


start_column = 0;

if (group_a == 0 || group_c == 0)
{
  cout << "ERROR:  Insufficient Memory 2" << endl;
  MPI_Abort(MPI_COMM_WORLD, 1);
}

// Scatter the Data
// The root process needs to scatter the correct amount of data to each process.
scatter(a, b, group_a, 0, rank, numtasks, dim_l, dim_m, dim_n, &start_column);

// Each process will start working on the data here
int startIndex = 0;
int row = 0;
int col = start_column;

// Split this loop with openmp
// Exam 2 Edit
#pragma omp parallel for num_threads(thread_count) schedule(dynamic)
for (int i = 0; i < base; i++)
```

```
  {
    // Exam 2 Edit
    // OpenMP splitting up the for loop, need to make sure that the loop row and
column
    // are dependent on the index (i) since that is how the threads are split

    // Get the column to work on
    int col = start_column + i;
    if (col % (dim_n) == 0)
    {
      col = 0;
    }
    else
    {
      col = col % dim_n;
    }

    // Get the row to work on
    int count = i;
    int row = 0;
    while (count > 0)
    {
      count -= dim_n;
      if (count >= 0)
        ++row;
    }

    group_c[i] = 0;
    float sum = 0;

    // Exam 2 Edit
    // This may not be helpful, maybe harmful depending on overhead
    //#pragma omp parallel for num_threads(thread_count) schedule(static, 1)
shared(row, col)
    for (int j = 0; j < dim_m; j++)
    {
      sum += group_a[dim_m*row + j] * b[j*dim_n + col];
    }

    group_c[i] = sum;
  }

  // Gather
  gather(c, group_c, num_mults, 0, rank, numtasks, dim_l, dim_n);

  /*
  Start recording the execution time
  */
  /*TIMER_CLEAR;
  TIMER_START;*/

  /*
     stop recording the execution time
  */
  //TIMER_STOP;

  if (rank == 0)
  {
    finish = MPI_Wtime();
    cout << "C matrix =" << endl;
    print_matrix(c, dim_l, dim_n);
    cout << endl;
```

```
    cout << "time = " << setprecision(8) << (finish - start) << " seconds " << endl;
    //cout << "time=" << setprecision(8) <<  TIMER_ELAPSED/1000000.0
         //<< " seconds" << endl;
  }

  if (rank == 0)
    cout << "Made it to the cleanup" << endl;

  // Clear out memory
  if (rank == 0)
  {
    delete[] a;
    delete[] b;
    delete[] c;
  }

  delete[] group_a;
  delete[] group_c;


  // Terminate MPI Program -- perform necessary MPI housekeeping
  // clear out all buffers, remove handlers, etc.
  MPI_Finalize();
}
```

Appendix C: Bounded Buffer OpenMP Source Code

```
// General Bounded Buffer Producer/Consumer Problem
// This program is to dynamically spawn a user defined set
// of producer threads and consumer threads which communicate
// with one another using a circular buffer where the
// producer thread places its id.
// FILE: bounded_buffer_omp.cpp
//
// Edits:
// Kyle Ray
// Exam 2
// CPE 512
// November 20, 2017
//
// Added OpenMP worksharing construct as well as
// OpenMP task implementations.

using namespace std;
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <string.h>
#include <omp.h>
#include <semaphore.h>
#include <unistd.h>


/* Compilation on the UAH ECE Jetson Cluster (blackhawk.ece.uah.edu)

  first set up environment by typing from the command line the
  following two module load commands

   module load intel
   requires linking with util.o file at compile time which should
   be placed in the same directory as bounded_buffer_omp.cpp file

   to compile/link the program with routines in the util.o file type

      g++ bounded_buffer.cpp -o bounded_buffer_omp util.o -fopenmp

   to run

     ./bounded_buffer_omp

*/

// producer/consumer headers for routines
// that are in util.o file
void init_producer(int);
void init_consumer(int);
int produce_next_data_item(int);
int consume_item(int,int);

// Global Counting Semaphore Declarations
sem_t empty, full;

// Global MUTEX Lock
omp_lock_t mutex;

// producer and consumer slot indices
int con_index=0, prod_index=0;
```

```
// Pointer to Shared Buffer used by
// Producer/Consumer Threads
int *shared_buf;

// major runtime parameters to be obtained from user
int num_producers; // number of producer processes
int num_consumers; // number of consumer processes
int num_items;     // number of data items
int N;             // number of data slots in circular buffer

// SHARED_BUF(x,y) macro
//    where indicies
//       x = 0 -> id of producer; x = 1 -> data that was produced
//       y = next location in finite bounded buffer
// This macro is used to represent the dynamically declared region
// of memory pointed to by the globally defined shared_buf varible
// to represent a two-dimensional memory space where the first
// dimension is of size 2 and the second dimension is equal to the
// number of entries in the circular buffer as defined at
// execution time.
#define SHARED_BUF(x,y) (shared_buf[y*2+x])

// global file descriptor for diary file
ofstream diary;

// routine to ask the user at run time for the number of producer and
// consumer threads, total number of data items to be produced/consumed
// (to be evenly divided among the producer/consumer threads),
// and the number of entries in the communication buffer
void init_data(int * producers, int * consumers, int * items, int *N) {
   do {
      cout << "Enter Number of Producer Threads to be Spawned:";
      cin >> *producers;
   } while (*producers < 1);

   do {
      cout << "Enter Number of Consumer Threads to be Spawned:";
      cin >> *consumers;
   } while (*consumers < 1);

   do {
      cout << "Enter the Total Number of Items to be Produced/Consumed:";
      cin >> *items;
   } while (*items < 1);

   do {
      cout << "Enter the Number of Entries present in the Circular Buffer:";
      cin >> *N;
   } while (*N < 1);

   // if present erase file named diary.txt
   diary.open("diary.txt");
   diary.close();

   // then open the diary.txt file
   // again in append mode
   diary.open("diary.txt",ios::app);

}

// utility function to convert integers into old style C strings
void int_to_C_string_conv(char * C_str_num, int num) {
```

```
   std::string s;
   std::stringstream out;
   out << num;
   s = out.str();
   strcpy(C_str_num,s.c_str());
}

// macro that computes the number of items in the list associated
// with a particular consumer or producer thread that has the
// specified thread id. It assumes that the data items that
// are to be processed by the set of threads in each category
// should be as eqaul as possible (most heavily loaded thread
// has at most one more data item than the least heavily loaded
// one).
// inputs:
//    id:          thread id within the group
//    nm_items:    total number of items in list
//    nm_threads:  total number of threads in the category
// outputs:
//    if (nm_items%nm_threads) return nm_items/nm_threads + 1;
//    else return nm_items/nm_threads;
#define group_sz(id,nm_items,nm_threads)
(nm_items/nm_threads+(id<(nm_items%nm_threads)))

void producer(int id) {

   // Create a Separate Producer diary file named prod_{id}.txt
   // for the particular producer that is being referenced
   char id_string[10]; // C style string rep of the producer id
   char prodfile[20];   // C style file name string
   strcpy(prodfile,"prod_");
   int_to_C_string_conv(id_string,id);
   strcat(prodfile,id_string);  // append producer id to file name
   strcat(prodfile,".txt");     // give it a *.txt extension
   ofstream prod_file(prodfile); // open prod file stream for writing
   // initialize the producer function produce_next_data_item(id) so that
   // it produces consecutively numbered data items for each thread
   // starting at 1 and varies in terms of its execution time in a
   // threadsafe pseudorandom manner.
   init_producer(id);

   for (int i=0;i<group_sz(id,num_items,num_producers);i++) {
      // produce the next data item -- not in critical section
      // This could take varing amounts of time
      int new_data =  produce_next_data_item(id);

      // add semaphore synchronization
      // enter your code here!!!
      // Decrement full count and block if full
      sem_wait(&full);
      /*
      int value = 0;
      sem_getvalue(&empty, &value);
      cout << "empty val = " << value << endl;
      sem_getvalue(&full, &value);
      cout << "full val  = " << value << endl;
      */

      // set omp lock variable -- enter critical section
      omp_set_lock(&mutex);
      // Critical Region of Code where produced item is placed in
      // next slot of buffer and global producer index is
      // incremented by one in a modulo manner
```

```
      cout << "Hello from Producer " << id << endl << flush;

      // place data into shared buffer
      // data includes producer ID & data
      SHARED_BUF(0,prod_index)=id;       // producer ID
      SHARED_BUF(1,prod_index)=new_data; // produced data
      cout << "Hello from Producer " << id << endl << flush;
      // log activity in common diary file
      diary << "Producer" << id << " placed data [" << SHARED_BUF(1,prod_index) <<
            "] in buffer slot(" << prod_index << ")" << endl << flush;

      // also log activity in local prod file
      prod_file << "Producer" << id << " placed data [" << SHARED_BUF(1,prod_index)
             << "] in buffer slot(" << prod_index << ")" << endl;

      /* increment prod_index to point to next slot */
      prod_index = (prod_index+1)%N;

      // reset omp lock variable
      omp_unset_lock(&mutex);

      // add appropriate semaphore synchronization
      // enter your code here!!!
      // Unclock the empty if it is locked because we just put something in the buffer
      sem_post(&empty);
   }
   // close the producer file
   prod_file.close();
}

void consumer(int id) {

   // Create a Separate Consumer diary file named con_{id}.txt
   // for the particular consumer that is being referenced
   char id_string[10]; // C style string rep of the consumer id
   char confile[20];   // C style file name string
   strcpy(confile,"con_");
   int_to_C_string_conv(id_string,id);
   strcat(confile,id_string);  // append consumer id to file name
   strcat(confile,".txt");     // give it a *.txt extension
   ofstream con_file(confile); // open con file stream for writing

   // initialize the consumer function consume_item(data) so that
   // it varies in time in a threadsafe pseudorandom manner.
   init_consumer(id);
   for (int i=0;i<group_sz(id,num_items,num_consumers);i++) {

      // add semaphore synchronization
      // enter your code here!!!
      // Is the buffer empty, if so then wait
      sem_wait(&empty);

      omp_set_lock(&mutex);
      // Critical Region of Code where produced item is consumed from
      // next slot of buffer and global consumer index is incremented by one
      // in a modulo manner

      cout << "Hello from Consumer " << id << endl << flush;

      // get data from shared buffer
      // data includes producer ID & data
      int prod_id  =SHARED_BUF(0,con_index); // producer ID
```

```
      int prod_data=SHARED_BUF(1,con_index); // produced data

      // log activity in common diary file
      diary << "Consumer" << id << " Received Producer" << prod_id <<
            "'s data item [" << prod_data << "] via the buffer slot ("
            << con_index << ")" << endl << flush;

      // also log activity in local con file
      con_file << "Consumer" << id << " Received Producer" << prod_id <<
               "'s data item [" << prod_data << "] via the buffer slot ("
               << con_index << ")" << endl;

      // increment con_index to point to next slot in buffer
      // wrapping back arround to 0 at the end of the buffer region
      con_index = (con_index+1)%N;

      // reset omp lock variable
      omp_unset_lock(&mutex);

      // add semaphore synchronization
      // enter your code here!!!
      // Unlock full, because we just consumed an item from the buffer
      sem_post(&full);

      // while outside of the critical area consumer thread
      // consumes the data item in a meaningfull way
      // This could take varing amounts of time
      consume_item(prod_data,id);
   }

   // close the consumer file
   con_file.close();
}

int main (int argc, char *argv[]) {

   // prompt user for number of producers, consumers,items,
   //  and buffer size
   init_data(&num_producers, &num_consumers, &num_items,&N);

   // initialize shared memory region
   shared_buf = new int[N*2];


   // initializing standard OpenMP MUTEX Variable
   omp_init_lock(&mutex);

   // initalize Counting Semaphores, full & empty
   // enter your code here!!!
   // enter your code here!!!
   sem_init(&empty, 0, 0);
   sem_init(&full, 0, N);

   // Spawn Consumer/Producer Threads
   int test = 2; // 1 = Work Sharing, 2 = OpenMP Tasks
   switch (test)
   {
     //  Case 1) using simple work sharing data parallel constructs
     //          enter your code here
     // (KRR)
     // Want both producer and consumer to kick of in parallel
     // Use semaphores to lock the producer putting something in
     // the buffer is full.  Block the consumer if there is nothing
```

```
  // to read from the buffer.
  case 1 :
  {
    #pragma omp parallel sections
    {
      #pragma omp section
      {
        #pragma omp parallel for
        for(int i = 0; i < num_producers; i++) {
          producer(i);
        }
      }

      #pragma omp section
      {
        #pragma omp parallel for
        for(int i = 0; i < num_consumers; i++) {
          consumer(i);
        }
      }
    }
    break;
  }
  case 2 :
  {
    //  Case 2) using OpenMP's tasking model
    //          enter your code here
    // (KRR)
    // Kick off the tasks using a single thread
    #pragma omp parallel
    {
      // Have a single thread kick everything off
      #pragma omp single
      {
        // Place Producers in a Task
        for(int i = 0; i < num_producers; i++) {
          #pragma omp task
          producer(i);
        }

        // Place Consumers in a Task
        for (int i = 0; i < num_consumers; i++) {
          #pragma omp task
          consumer(i);
        }

        #pragma omp taskwait
      }
    }
    break;
  }
}

// destroying standard OpenMP MUTEX Variable
omp_destroy_lock(&mutex);

// destroy semaphores full & empty
// enter your code here!!!
// enter your code here!!!
sem_destroy(&empty);
sem_destroy(&full);

// close diary.txt file
```

```
    diary.close();
}
```