Shared-memory programming using OpenMP

General description

This list of challenges was designed to "hands-on" the multithreaded parallel programming model. The term *thread* suggests a stream of control in a single process, and in a shared-memory program a single process may have multiple threads of control. To assess this model we will use OpenMP(for Open MultiProcessing), a library which is supported by most modern C/C++ compilers, including the GNU suite (gcc/g++)

OpenMP is not a stand-alone “language”; it is an API to an existing language. The OpenMP API consists of a set of compiler directives (called “pragmas”), runtime library routines and environment variables. The pragmas that are added to the original sequential program appear to be nothing more than comments to a non-OpenMP version of the compiler, and are simply ignored. However, by using an OpenMP-capable compiler (and activating that feature with the proper compiler switch or option), OpenMP can provide the compiler with additional information that can be used to parallelize the program.

For the following programs, turn in a printout (e.g., a screenshot) of the execution trace. In some cases, you may need to run several executions of the program. It is not necessary to turn in printouts of all executions, just a representative sample of them. It is also not necessary to show the entire trace of an execution. Some problems will generate output larger than what will fit in one screen of the terminal emulator (e.g., Putty). Just include what you feel is necessary to effectively answer the question.

1) OpenMP Model

OpenMP uses the fork-join model, which is essentially based on the same system fork command that was used in the previous projects. Just like the fork command, we can tell OpenMP to spawn a child process that will execute the exact same code that the parent process will execute. In OpenMP, the forking is handled

automatically, in a more transparent and convenient manner for the programmer. In addition, with a single pragma, multiple child processes (actually threads) can be created. A thread is a runtime entity that is able to independently execute a stream of instructions. In OpenMP, a Parallel Region is a block of code executed by all threads simultaneously. Each thread is not thought of as being a parent or child, but rather the group is referred to as a "team". The region of the program which should be run in parallel by the team is simply bounded as follows:

```
1. #pragma omp parallel
2. {
3.     <portion of code which team executes in parallel>
4. }
```

If the number of desired threads is not explicitly specified by the programmer, then OpenMP will set the number of threads by default to match the number of logical hardware cores in the system. Run the following program sequentially, that is, compile it without the -fopenmp option, then run it in parallel with the -fopenmp option specified during compilation. Explain what happens in each case. Recall that the Intel MTL is a 32-core machine, where each physical core is Hyper-Threaded.

```
1. #include <stdio.h>
2. int main(){
3.    #pragma omp parallel
4.      printf(" Hello ");
5.    printf("\n GoodBye -PUJPP- Exiting Program \n");
6. }
```

2) The team

Now, specify the number of threads in the "*team*" or group by using *omp_set_num_threads(nthreads)*. Although some compilers may not require it, note that you may also need to add the following include clause as well

```
#include <omp.h>
```

The program from (1) should now appear as follows for nthreads = 2:

```
1. #include <omp.h>
2. #include <stdio.h>
3.
4. int main(){
5.    omp_set_num_threads(2);
6.    #pragma omp parallel
7.      printf("Hello ");
8.    printf("\n GoodBye -PUJPP- Exiting Program \n");
9. }
```

Run the program above with the nthread parameter set to 2, 4 and 8. Verify that the output is as expected for the different number of threads created in the team. Is it possible to specify an odd number of threads? Is it possible to specify a number of threads greater than the number of logical cores (or physical cores if no HyperThreading is present) available in the hardware?

3) Thread identifier

The omp function call `omp_get_thread_num()` retrieves the ID of the thread.

Note that the master thread always has thread ID 0. That is, the numbering of the threads in the team starts from zero; not one. The other threads are given IDs of consecutively higher integers (1, 2, 3, etc.) up to nthreads-1.

By using `omp_get_thread_num()` we can have each thread print out its ID in addition to `" Hello thread = %d "`. Changing the single printf line with "Hello" in it from (2) above to get the following:

```
1. #include <omp.h>
2. #include <stdio.h>
3. int main(){
4.    omp_set_num_threads(2);
5.    #pragma omp parallel
6.      printf(" Hello thread = %d ", omp_get_thread_num());
7.    printf("\n GoodBye -PUJPP- Exiting Program \n");
8. }
```

Run the program for various values of nthreads. Perform several runs for each value of nthread and comment on the order in which the threads are run. Note that you can also specify an nthreads value of 1 to default back to the sequential, single thread case.

4) More often than not, a programmer is not just interested in having several threads all executing an identical section of code; (s)he would want to distribute the work among the threads so that it can be completed in less time and in parallel. With the *fork()* call, the programmer had to manually separate the work for parent vs. child by performing a test on the return value of the fork system call. For example, one branch of the IF represented parent executed code; the other branch represented the child executed code. OpenMP can do this partitioning of work among the available threads automatically.

A work-sharing construct of OpenMP divides the execution of an enclosed code region (typically a loop) among the members of the team; in other words: OpenMP splits the work between the threads. The OpenMP construct that does this for loop constructs is the keyword "for" appended onto the previously seen "parallel" pragma: ***#pragma omp parallel for*** Inserting this statement before a 'C' for loop will distribute the iterations of the for loop among the threads.

Enter this program to trace the 16 iterations of the loop and observe how the iterations are distributed among the threads as a function of the number of threads. Run the program for nthreads=2, 4, 8, 16 and 32. Explain what happens. What happens if the number of iterations in the for loop is not evenly divisible by the number of threads? For example, explain what happens in the program below if nthreads were set to 5 or 7, instead of to 2 as currently shown.

```
1. #include <stdio.h>
2. #include <omp.h>
3.
4.
5. int main() {
6.  int i;
7.  omp_set_num_threads(2);
8.  #pragma omp parallel for
9.  for (i = 0; i < 16; i++) {
10.    printf("Hello from thread number: % d Iteration: % d\ n",
11.      omp_get_thread_num(), i);
12.  }
13.  printf("\n GoodBye -PUJPP- Exiting Program \n");
14. }
15.
```

Notice that the iterations are split into contiguous chunks (rather than round robin fashion), and each thread gets one chunk of iterations. By default, OpenMP splits the iterations of a loop into chunks of equal (or roughly equal) size, assigns each chunk to a thread, and lets each thread loop through its subset of the iterations.

5) Degree of parallelization

Although OpenMP makes most variables shared by default and visible to all threads, this is not always what is needed to provide proper and correct behavior of the parallel version of the code. The code segment below produces the sum of integers from 0 up to (COUNT-1). When run in sequential mode, it produces the correct answer. However, when parallelized incorrectly as shown below, non-deterministic errors can occur. The errors are non-deterministic because they do not always happen – they only occur under certain conditions – the erroneous answers are different from run to run – and there is no apparent error or warning message, so it appears that everything is fine.

Run the code sequentially first for the three values of COUNT to produce the correct answer. Then run it for the following values of NTHREADS and COUNT. Suggested values for NTHREADS include 2, 4, 8, 32, and 64. Suggested values for COUNT include 10, 100, and 1000. Create a table similar to the one below, making at least ten runs at each of the 15 data points. For each of the 15 data points, estimate the probability of an incorrect answer. If you do exactly ten runs at each data point, then this probability will be easy to report. (You can optionally do a larger number runs at each data point to get a better statistical average) Why are some answers incorrect? What is causing the non-deterministic errors? What is the probability of obtaining an incorrect answer as a function of NTHREADS and COUNT? Is there a trend in the probability and what intuitive explanation might there be for it? Optionally, repeat the experiment on a single/dual/quad core machine, if you have access to one.

Percent of Incorrect Answers Generated by Flawed Parallel OpenMP Program

COUNT	NTHREADS=2	NTHREADS=4	NTHREADS=8	NTHREADS=32	NTHREADS=64
10					
100					
1000					

```c
#include <stdio.h>
#include <omp.h>


int main() {
 int i;
 int sum = 0;
 omp_set_num_threads(NTHREADS);
 #pragma omp parallel for
 for (i = 0; i < COUNT; i++) {
   sum = sum + i;
   printf("Thread number: % d Iteration: % d Local Sum: % d\ n",
     omp_get_thread_num(), i, sum);
 }
 printf("\n All Threads Done— Final Global Sum: % d\ n\ n", sum);
}

```

6) Private vs global scope

OpenMP knows to make certain variables private instead of shared, that is, visible only to individual threads. For example, the loop iteration variable, i, in the above code of (5) is automatically made private by OpenMP. OpenMP also allows a programmer to explicitly declare some variables as being private. What happens if the variable sum is made private with #pragma omp parallel for private(sum)

Make this small change to (5) and document and explain what happens. Does this solve the non-deterministic incorrect answers produced from (5) above? Explain why or why not.

7) Reduction

The reduction(+:sum) clause appended to the end of the *#pragma omp parallel* directive will solve the problem associated with (5) above. Run the program below with a few different values of NTHREADS and COUNT. Explain what reduction(+:sum) does.

```c
#include <stdio.h>
#include <omp.h>

int main() {
  int i;
  int sum = 0;
  omp_set_num_threads(NTHREADS);
  #pragma omp parallel for reduction(+: sum)
  for (i = 0; i < COUNT; i++) {
    sum = sum + i;
    printf("Thread number: % d Iteration: % d Local Sum: % d\ n",
      omp_get_thread_num(), i, sum);
  }
  printf("\n All Threads Done— Final Global Sum: % d\ n\ n", sum);
}
```

8) Compute PI

The program listed below computes the value of PI using iteration. Run the program sequentially first, taking a time measurement.

```c
#include <stdio.h>
#include <omp.h>
#include <time.h>

long long num_steps = 1000000000;
double step;

int main(int argc, char* argv[]){
        double x, pi, sum=0.0;
        int i;
        step = 1./(double)num_steps;
        for (i=0; i<num_steps; i++){
                x = (i + .5)*step;
                sum = sum + 4.0/(1.+ x*x);
        }
        pi = sum*step;
        printf("The value of PI is %15.12f \n",pi);
}

```

You can time the sequential program by typing time ./a.out See page 138 of the Chapman book on Using OpenMP for more information on using the time system function. Note that the

program does not need any user input and takes about 20 seconds of wall clock time on MTL to complete. Record the sequential time.

9) Parallelize the PI

Parallelize the PI program above, by including the following two OpenMP parallelization clauses immediately before the 'for loop'.

```
omp_set_num_threads(128);
#pragma omp parallel for private(x) reduction(+:sum)
```

In this particular case, adding just two more lines to the sequential program will convert it to a parallel one. Also note that omp_set_num_threads(NTHREADS) is not really necessary.

OpenMP will simply set the number of threads to match the number of logical cores in the system by default. So only one additional line consisting of an OpenMP #pragma omp parallel…. was really required to convert from sequential to parallel. We include the other one as well because we are interested in explicitly setting NTHREADS to different values as part of our experimentation. Time the parallel program below using various values of NTHREADS. Record and report your findings of Time vs. NTHREADS. Include test cases involving NTHREADS > 32, the number of physical cores, and NHREADS > 64, the number of logical cores in MTL.

Explain any observations. Optionally, repeat the experiment on single/dual/quad core machine(s), if you have access to these alternate hardware platforms.