OpenMP

Introduction

OpenMP stands for Open specification for Multi-Processing. It is a "standard" API for defining multi-threaded shared-memory programs. OpenMP is a portable, threaded, shared-memory programming specification with "light" syntax. OpenMP allows programmers to separate a program into serial regions and parallel regions, rather than concurrently-executing threads. OpenMP also hides the complexity of stack management and provides synchronization constructs for easy parallelization.

Unlike MPI, OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence. OpenMP is a small API that hides cumbersome threading calls with simpler directives. However, while MPI programs can run across many networked computers without modification, OpenMP is only designed to run on one PC – albeit one which multiple cores or processors.

General Syntax for OpenMP directives:

```
#pragma omp directive [clause…]
```

Parallel Syntax

```
#pragma omp parallel [clause…]
      structured_block
```

e,g.

```
#include <omp.h>
#pragma omp parallel
{
  printf("Thread rank: %d\n", omp_get_thread_num());

} // implicit barrier

Output: (T=4)
Thread rank:  2
Thread rank:  0
Thread rank:  3
Thread rank:  1
```

Note that it can be in any order.

Parallel For Syntax (combines parallel and for)

```
#pragma omp parallel for [clause…]
            for_loop
```

e.g.

```
#pragma omp parallel for shared(x)
for(i=0;i<x;i++) {
  printf("Hello!\n");
}
```

Exercise 1

Modify the following "Hello World" serial code to run multithreaded by using OpenMP

```
#include <stdio.h>

int main() {
           int i;
           printf("Hello World\n");
           for(i=0;i<6;i++)
                printf("Iter:%d\n",i);

           printf("GoodBye World\n");
}
```

Add `omp_get_thread_num()` to the print so that you can see what thread is running which iteration.

To compile:

```
$ gcc -fopenmp omp_hello.c -o omp_hello
```

Set environment variable for number of threads:

```
$ export OMP_NUM_THREADS=4
```

You can also use this to set it to the number of processors:

```
$ export OMP_NUM_THREADS=`nproc`
```

To run

```
$ ./omp_hello
```

Exercise 2

Given the following OpenMP partial code, describe the problem that will occur related to the code. Modify the code to solve the problem. To do this, Google for "Open MP reduction" to do the equivalent of the MPI_Reduce for OpenMP.

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

Exercise 3

The following C code loads a text file then counts how many of each letter there is.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <time.h>

#define ARRAYSIZE ('Z'-'A'+1)

int main(int argc, char *argv[])
{ FILE *infile;
  int i, count[ARRAYSIZE]={0};
  long int size;
  char *buffer;
  clock_t start, end;

  infile = fopen(argv[1],"r");
  if (infile)
  {   fseek(infile,0L, SEEK_END); // Find out the
      size = ftell(infile);       // size of the file.

      fseek(infile,0L, SEEK_SET); // Read the entire
      buffer = malloc(size);      // file into the
      fread(buffer,1,size,infile);     // buffer.

      start = clock();

      for (i=0; i<size; i++)
            if (isalpha(buffer[i]))
                  count[tolower(buffer[i])-'a']++;

      end = clock();

      printf("CPU time used: %f seconds\n",
            ((double) (end - start)) / CLOCKS_PER_SEC);

      for (i=0; i<ARRAYSIZE; i++)
            printf("%c %d\n",i+'A', count[i]);
  }
  else
        printf("Could not open %s for reading.\n",argv[1]);
}
```

A straightforward `#pragma omp parallel for` on `for (i=0; i<size; i++)` is problematic because the same array entry `count[tolower(buffer[i])-'a']` could be incremented by different threads at the same time.

Google "Array sum reductions in OpenMP 4.5" to see how you can do a reduction on the array so that this will work correctly, and modify the program accordingly.

The `clock()` function doesn't always work properly in multithreaded programs, so you will have to replace it with `omp_get_wtime()` instead in the OpenMP version of this program. This means changing `#include <time.h>` to `#include <omp.h>`, `clock_t start, end;` to `double start, end;`, and `printf("CPU time used: %f seconds\n",((double) (end - start)) / CLOCKS_PER_SEC);` to `printf("CPU time used: %f seconds\n",end - start);`

The datafile for this exercise, `lab07-3-data.txt`, which contains the text of *The Complete Works of William Shakespeare* is big enough that the time taken should be visible between the serial version and the parallel version. For example, on my computer, the serial version took 0.031175 seconds while the parallel version, running on 16 threads, took 0.003824 seconds. The time will vary, depending on how many background processes happen to be running at the time – I obtained the 0.003824 seconds number shortly after rebooting my computer when there were few other processes running.

Exercise 4

The `#pragma omp parallel for` requires that the loop variable is an integer. So what happens if the loop variable is *not* an integer, as in this example?

```
#include <stdio.h>
#include <math.h>

#define START (M_PI/2)
#define END (M_PI*2)

double f(double x)
{ return sin(x/2)+1;
}

int main(int argc, char *argv[])
{ double total = 0, x;
  int partitions;
  double slice;

  printf("How many partitions? "); fflush(stdout);
  scanf("%d", &partitions);
  slice = (END-START)/partitions;
  for (x = START + (slice/2); x < END; x = x + slice)
    total = total + f(x);
  total = total * slice;

  printf("The integration is %1.10f\n", total);
}
```

Convert this program into OpenMP.