CS 475/575 -- Spring Quarter 2018

Project #0

Simple OpenMP Experiment

30 Points

Due: April 9

This page was last updated: March 15, 2018

Introduction

A great use for parallel programming is identical operations on large arrays of numbers.

Requirements

1. Pick an array size to do the arithmetic on. Don't pick something too huge, as your machine may not allow you to use that much memory. Don't pick something too small, as the overhead of using threading might dominate the parallelism gains.

2. Using OpenMP, pairwise multiply two large floating-point arrays, putting the results in another array.

3. Do this for one thread and do this for four threads:
   #define NUMT 1
   and
   #define NUMT 4

4. Time the two runs. Convert the timing results into "Mega-Multiplies per Second".

5. Review the Project Notes pages on "How Reliable is the Timing?"

6. What speedup, S, are you seeing when you move from 1 thread to 4 threads?
   S = (Execution time with one thread) / (Execution time with four threads)

7. If your 4-thread-to-one-thread speedup is **S**, compute the parallel fraction:

   ```
   float Fp = (4./3.)*( 1. - (1./S) );
   ```

Don't worry what this means just yet. This will become more meaningful soon.

8. Your written commentary (turned in as a PDF file) should include:
    1. Tell what machine you ran this on
    2. What performance results did you get?
    3. What was your 4-thread-to-one-thread speedup?
    4. Why do you think it is behaving this way?
    5. What was your Parallel Fraction, Fp?

The main Program

Your main program would then look something like this:

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

#define NUMT             4
#define ARRAYSIZE       ??      // you decide
#define NUMTRIES        ??      // you decide

float A[ARRAYSIZE];
float B[ARRAYSIZE];
float C[ARRAYSIZE];

int
main( )
{
#ifndef _OPENMP
        fprintf( stderr, "OpenMP is not supported here -- sorry.\n" );
        return 1;
#endif

        omp_set_num_threads( NUMT );
        fprintf( stderr, "Using %d threads\n", NUMT );

        double maxMegaMults = 0.;
        double sumMegaMults = 0.;

        for( int t = 0; t < NUMTRIES; t++ )
        {
                double time0 = omp_get_wtime( );

                #pragma omp parallel for
                for( int i = 0; i < ARRAYSIZE; i++ )
                {
                        C[i] = A[i] * B[i];
                }

                double time1 = omp_get_wtime( );
                double megaMults = (double)ARRAYSIZE/(time1-time0)/1000000.;
                sumMegaMults += megaMults;
                if( megaMults > maxMegaMults )
                        maxMegaMults = megaMults;
        }

        double avgMegaMults = sumMegaMults/(double)NUMTRIES;
        printf( "   Peak Performance = %8.2lf MegaMults/Sec\n", maxMegaMults );
        printf( "Average Performance = %8.2lf MegaMults/Sec\n", avgMegaMults );

        // note: %lf stands for "long float", which is how printf prints a "double"
        //        %d stands for "decimal integer", not "double"

        return 0;
}
```

Grading:

Feature	Points
Execution time results for 1 thread	5
Execution time results for 4 threads	5
Four-thread-to-one-thread Speedup	5
Parallel Fraction	10
Commentary	5
Potential Total	**30**