Project 0: Getting Started with OpenMP

Behnam Saeedi
(Saeedib@oregonstate.edu)
CS575: Parallel Programming

Winter 2018

Abstract

This project is an introduction to OpenMP and how to set it up and use it. In this project we learn how to time, compute ratio of the computation per second, compare performance between single thread and multi-thread programming and compute the parallel fraction.

CONTENTS

1 REQUIREMENTS

1) Execution time results for 1 thread 5
2) Execution time results for 4 threads 5
3) Four-thread-to-one-thread Speedup 5
4) Parallel Fraction 10
5) Commentary 5
6) Potential Total 30

2 APPROACH

In order to solve this problem we will need to run the code 2 times. The first time around it is run with a single thread and the time is recorded. This time is stored in a a temporary file for later computation. Same process is done again with 4 threads an the time is recorded in another temporary file. The times are later used to compute both speed up and the parallel fraction.

3 UTILITY

The utility class provides the following functions:

```
class Utility{
public:
        Utility();
        bool isOpenMP();
        double megaMult(long op, float time);
        float speedUp();
        float pf(float speedup);
        float TS;
        float TM;
        float speedUp(float ts, float tm);
        void timerStart();
        void timerStop();
        double Time();
private:
        struct timeval startTime;
        struct timeval endTime;
};
```

This class allows us to time, find the multi process fraction, figure out speed up and compute the parallel fraction. Another portion of the utility is a bash file called ”run.sh”. This bash file computes the same values from an OS level allowing us to run the same code with different number of threads and adjust the number of threads in runtime rather than compile time.

```
#!/bin/bash
#echo "Compiling..."
#make
echo "Running the code with 1 thread..."
./out 1 s > tmp1
./out 1
echo "Running the code with 4 thread..."
```

```
./out 4 s > tmp2
./out 4
echo "extracting times..."
TS=`cat tmp1`
TM=`cat tmp2`
SU=$(echo "$TS/$TM" | bc -l)
#(4./3.)*( 1. - (1./S) )
AF=$(echo "4/3" | bc -l)
BF=$(echo "1/$SU" | bc -l)
CF=$(echo "1-$BF" | bc -l)
PF=$(echo "$AF*$CF" | bc -l)
echo "Speedup is $SU"
echo "parallel fraction returned $PF."
echo "cleaning up ..."
rm tmp1
rm tmp2
echo "Done!"
```

4 Time Trials

The first portion of the program runs 50000^2 computations on a single thread. This procedure takes about 6.23 seconds and at a peak rate of 406.5 Mega computations per second and an average performance of 402 computations Mega per second. The same number of computations ran in about 3.5 seconds at a peak and average performance rate of 1002 and 751.2 mega operations per second respectively.

5 Speed Up

The speedup is computed using the following function:

$$\frac{Time_{SingleThread}}{Time_{MultiThread}}$$

After running the code with 50000^2 computations in 6.23 seconds and 3.50 seconds on single and multi thread respectively it was computed that the speed up was by a factor of 2.39. The code ran roughly two and a half time faster with 4 threads running instead of a single thread.

6 Parallel Fraction

The parallel fraction is computed using the following function:

$$\frac{4}{3} \times (1 - \frac{1}{SpeedUp})$$
$$= \frac{4}{3} \times (1 - \frac{Time_{MultiThread}}{Time_{SingleThread}})$$

Using this function and the previously calculated 2.39 speed up factor we can compute that the parallel fraction is about 0.77.

7 RESULTS

```
g++ -fopenmp OMPUtil.cpp main.cpp -o out
./run.sh
Running the code with 1 thread...
Initiating the utility tools...
OpenMP is ready!
Using 1 threads
   Peak Performance =   406.48 MegaMults/Sec
Average Performance =   401.94 MegaMults/Sec
      Total time is = 6.22711.
Running the code with 4 thread...
Initiating the utility tools...
OpenMP is ready!
Using 4 threads
   Peak Performance =  1001.92 MegaMults/Sec
Average Performance =   751.21 MegaMults/Sec
      Total time is = 3.50434.
extracting times...
Speedup is 2.38518671659047925328
parallel fraction returned .77432748106224766161.
cleaning up ...
Done!
```

8 CONCLUSION

The same program running on multiple threads does indeed run faster than a single thread. However, an interesting finding in this setup is that 4 threads do not necessarily yield 4 times more performance or a speed increase of four times. This could be partially due to the overhead associated with increased number of threads.