PHY905, section 4
Homework Assignment 6

Håkon V. Treider
MSU id: treiderh

April 20, 2016

Introduction

The tests have been run on Mac OS X (10.11.4) with a 2.5 GHz Intel Core i7.

In this assignment, we will look at openMP and see how a simple copy-operation can be sped up using multiple threads.

Assuming as always that the reader is familiar with the assignment text, the time per loop and rates can be seen in table 1 or visually in figure 1. Here I have used a vector with one million elements (4 byte `int`s) and timed the computation of copy 1000 times, and used the average. In a proper test, I would do this, say, 50 times and then report the minimum value of the fifty average values.

Table 1: Time and rates for copy-operation ($n = 1E6$)

Nt	Time/loop (ms)	Rate (MB/s)
1	6.51	12289
2	3.28	24390
4	1.84	43478
6	2.21	36364
8	1.73	46243
10	2.66	30075
12	2.69	29740
14	2.71	29520
16	2.67	29963

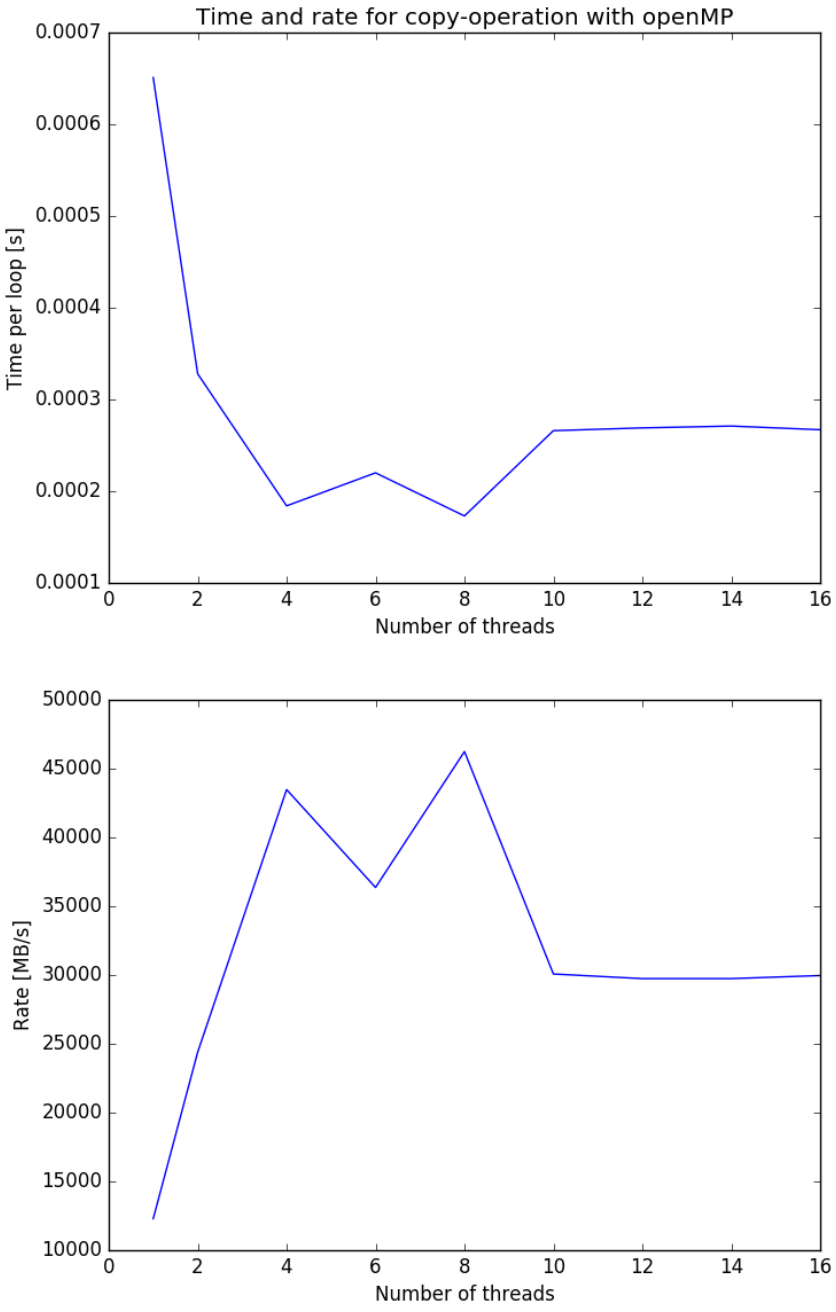

Figure 1: Experimental values for time and rates of copy with multiple threads

Disussion

As we have seen in the lectures, we get a linear speed up until we reach a regime where increasing the number of threads doesn't give any improvements at all. This constant regime starts at $N_t = 10$. The best performance is gained by using the same number of threads as cores (i.e. 4) or (even slightly better) the double of this (8 threads). I suspect this latter is because my CPU has hyper-threading and appears to have 8 logical cores, even though it just has 4 physical ones.

Let's compare the results with the following model:

$$r_t = \min(N_t\, r_1,\ \ r_{max}) \tag{1}$$

This basically predicts the scaling to be linear in *the number of threads,* N_t up to some limit r_{max} at which the rate stays constant. Selecting the constant ceiling to be the value of the maximum rate, we miss out on the details of the even higher rates that occur around 4 and 8 threads. Apart from that, it is almost a good fit, as can be seen in figure 2. To further show the linear region, I used the minimum only for values of N_t larger than 4, i.e. the number of cores on the CPU. This makes the model more valid for small number of threads, while still retaining the effect of high values of N_t.

A third model, which might be even better suited is to use the minimum of $N_t r_1$ and the actual r_{max} and then add a third variable $r_{N_t \to \infty}$ which takes over for $N_t > 10$ or whatever the limit is in a particular case.

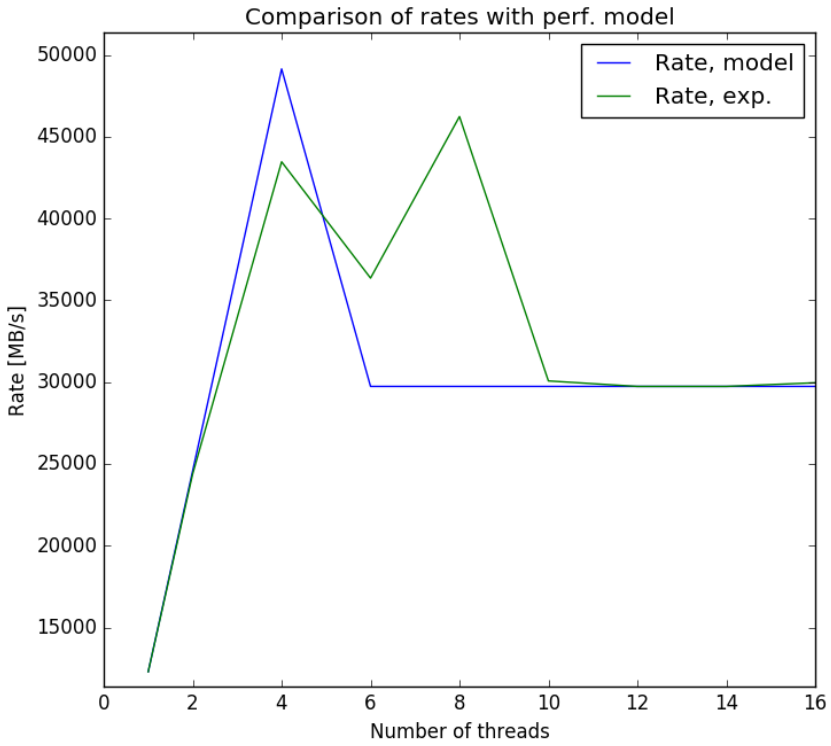

Figure 2: Actual performance vs. alternative perf. model of copy operation with multiple threads.

Code

```c
#include <libiomp/omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(void){

    int n = 1000000;
    int a[n],c[n];
    double t0,t1;
    int nk = 1000;

    // Loop over number of threads from 1,2,4,6...,14,16
    for (int Nt = 1; Nt<17; Nt += 2){
        omp_set_num_threads(Nt);
        t0 = omp_get_wtime(); // Start timing

        for (int k=0; k<nk; k++) { // Number of tests
#pragma omp parallel for
            for (int i=0; i<n; i++){
                c[i] = a[i];
            }
        }
        t1 = omp_get_wtime(); // End timing
        double tpl = (t1-t0)/nk; // Time per loop
        double rate = n*4*2 / (1e6*tpl);
        printf("%.2d threads, t/loop: %.5f sec, rate: %.0f MB/s\n",Nt,tpl,rate);

        if (Nt%2 != 0){Nt = 0;} // Fix step from 1->2

    }

    /*double sum = 0;
    int      N = 78;
    double   v[N];
    for (int i=1; i<N+1; i++){v[i-1] = i;}

#pragma omp parallel for reduction(+:sum)
    for (int i=0; i<N; i++){
        sum += v[i];
    }
    double cor_sum = N*(N+1)/2.0;
    printf("%f while correct sum is %f\n", sum,cor_sum);*/


    /*
    omp_set_num_threads(4);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        int np = omp_get_num_threads();
        printf("%d / %d\n",id,np);
    }*/

    return 0;
}
```