PHY905, section 4
Homework Assignment 7

Håkon V. Treider
MSU id: treiderh

April 23, 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, I look into how we can use OpenMP to speed up the computation of the dense matrix-matrix multiply (algorithm) by means of parallelization.

I have run the serial code with and without OpenMP to check how much overhead it adds to the computation. Then I will use t threads from 2 to 8, which is the number of logical cores on my chip. The number of physical cores is 4. I ran the code for square matrices of the order $n = 20, 200, 1000, 1300$ and the results can be seen below in table 1 or in figure 1.

Table 1: Average time of matrix-matrix multiply, in seconds

Nt	OpenMP?	$n = 20$	$n = 200$	$n = 1000$	$n = 1300$
1	No	0.000008	0.0076	1.58	14.8
1	Yes	0.000014	0.011	2.08	14.7
2	Yes	0.000011	0.0057	1.05	7.44
3	Yes	0.000013	0.0041	0.72	5.06
4	Yes	0.000011	0.0032	0.57	3.85
5	Yes	0.000011	0.0049	0.72	3.90
6	Yes	0.000011	0.0049	0.86	3.89
7	Yes	0.000010	0.0045	1.09	3.92
8	Yes	0.000011	0.0045	1.61	3.86

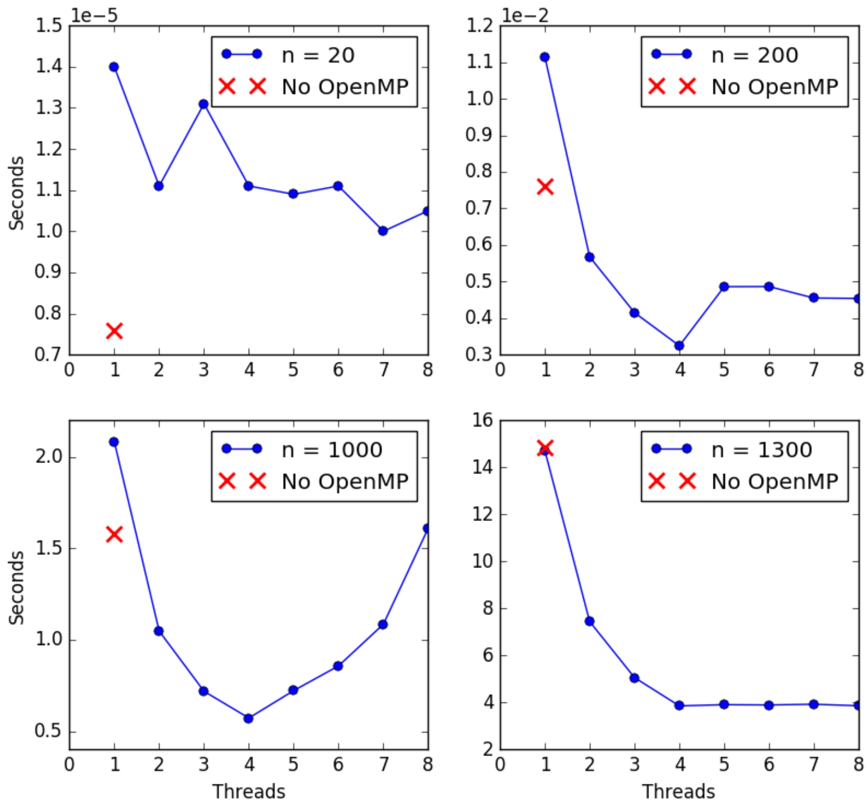

Figure 1: Computation time as function of number of threads used. Comparison with serial code.

Discussion

In the case of small matrix size (i.e. n=20), the overhead of using OpenMP is severe. So much in fact that using it, slows the computation by 20-100 %, depending on the number of threads. Going on step up on the magnitude ladder (n=200), we get a speed up when using more than one thread. This is of course because while the overhead stays roughly the same, the computation cost goes up by $\mathcal{O}(n^3)$ and since we can divide this work between multiple threads, the overall ratio goes down.

For reasonable matrix sizes, we have a distinct peak performance for *threads equal the number of processor cores*. This is best seen visually in the figure.

In homework assignment 6, we used a simple performance estimate where we assumed the total computation time scaled as $1/N_t$, up to a certain point where the computation is limited by memory bandwidth. At this point the computation time was almost constant for increasing number of threads. When we look at our result for the matrix $n = 1300$ we can see that this is also the case. The reason the three others doesn't seem to follow the same law, is because they are small enough to fit entirely in cache.

Code

```
#include <libiomp/omp.h>
#include <armadillo>

int main(void){
    const int n = 20; int Nt_max = 8; // Matrix size and max threads
    arma::mat A(n,n),B(n,n),C(n,n);
    double t0,t1,T,s;
    double T_end_tests = 0.5; // Seconds passed before quitting loop
    int test = 0;
    int max_tests_default = 300000; int max_tests = max_tests_default;

    // Fill A and B with numbers to check C is correctly computed
    for (int i=0; i<n; i++){
        for (int j=0; j<n; j++){
            A(i,j) = i+j+1;
            B(i,j) = 2*j+i+1;
        }
    }
    // One thread, NO openMP
    printf("Matrix %dx%d, No openMP,  ",n,n);

    for (test = 0; test<max_tests; test++){
        t0 = omp_get_wtime();          // Start timing
        for (int i=0; i<n; i++){
            for (int j=0; j<n; j++){
                s = 0.0;        // The sum for C(i,j)
                for (int k=0; k<n; k++){
                    s += A(i,k)*B(k,j);
                }
                C(i,j) = s;
            }
        }
        t1 = omp_get_wtime();           // End timing
        T += t1-t0;
        if (T>T_end_tests){max_tests = test+1;}
    }
    printf("Total t: %.2f, t/loop: %.7f, tests: %d, %f\n",T,T/max_tests,max_tests,C(1,1));

    // 1-8 threads, with openMP
    for (int Nt=1; Nt<Nt_max+1; Nt++){
        T = 0;                          // Reset timer
        max_tests = max_tests_default;  // Reset counter
        omp_set_num_threads(Nt);
        printf("Matrix %dx%d, threads: %d, ",n,n,Nt);

        for (test = 0; test<max_tests; test++){
            t0 = omp_get_wtime();          // Start timing
#pragma omp parallel for
            for (int i=0; i<n; i++){
                for (int j=0; j<n; j++){
                    s = 0.0;        // The sum for C(i,j)
                    for (int k=0; k<n; k++){
                        s += A(i,k)*B(k,j);
                    }
                    C(i,j) = s;
                }
            }
            t1 = omp_get_wtime();           // End timing
            T += t1-t0;
            if (T>T_end_tests){max_tests = test+1;}
        }
        printf("Total t: %.2f, t/loop: %.7f, tests: %d, %f\n",T,T/max_tests,max_tests,C(1,1));
    }
    return 0;
}
```