

```
// file: simpson_cosint_openmp.cpp
//
// A quick and easy way to parallelize using a simple OpenMP command
//
// C++ Program to compute  $\int k^* \cos(k x) dx$  for  $x$  on  $[0, \pi/2]$ 
// (which simply equals  $\sin(k^* \pi/2)$ ), but where the integrand
// has been "sabotaged" with  $k x = \text{acos}(\cos(kx))$  in order to
// make the computation take much longer. We will try out some
// tricks to make the computation go faster -- assume that we
// are interested in the result of the integrand for many different
// k-values.
//
// Programmer: Chris Orban orban@mps.ohio-state.edu
//
// Revision history:
// 01/30/2011 original C++ version
// 03/05/2011 documentation and formatting edits by
// Dick Furnstahl (furnstahl.1@osu.edu)
// 01/02/2012 Improved clock timing and stripped down
// openmp for loop call by Chris Orban
// (orban@physics.osu.edu) with advice from
// Chris Daley (UChicago)
//
// Notes:
// * This is only useful if run on a machine that can share memory
// with multiple processors. For example, a single computer with
// more than one core.
// * To use g++, compile with
// g++ -fopenmp -o simpson_cosint_openmp simpson_cosint_openmp.cpp -lgomp
// This will work fine on a MacBook, for example.
// * If using OSU Physics Department computers you can also use the
// Intel C++ compiler. Load the Intel compilers with this command:
// module load intel-10.0.023
// With the intel compiler loaded, compile the program like this:
// icpc -openmp -o simpson_cosint_openmp simpson_cosint_openmp.cpp
//
// *****
//
// include files
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cmath>
#include <omp.h>
using namespace std;
#include <time.h>

// function prototypes
double my_integrand (double k, double x);
double simpsons_rule ( int num_pts, double k, double x_min, double x_max,
double (*integrand) (double k, double x) );
// *****

// A handy trick to write Pi to many digits
double Pi = 4.*atan(1.);

int
main (void)
{
    double start, end; // variables to check elapsed time
    ofstream out ("cosint.dat"); // open the output file

    double lower_limit = 0; // lower limit a
    double upper_limit = Pi/2.; // upper limit b
    double exact; // exact answer to the integral

    double kmin = 0.0001; // minimum k value to loop over
    double kmax = 50.0; // maximum k value to loop over
    const int numi = 1000; // number of i's (and k's) to loop over

```

```
double k[numi]; // array of k values from kmin to kmax
double result[numi]; // array of integral results

omp_set_num_threads(2); // Use this or specify the number of threads
// from the cmd line with OMP_NUM_THREADS

start = omp_get_wtime(); // get the start time
int i;
#pragma omp parallel for
for (i = 0; i < numi; i++)
{
    k[i] = kmin + i*(kmax-kmin)/(numi-1);
    result[i] = simpsons_rule (100001, k[i], lower_limit, upper_limit,
&my_integrand);
}
end = omp_get_wtime(); // get the end time

// output the elapsed time.
cout << "num_time(s) = " << end-start << endl;

for (i = 1; i < numi; i++)
{
    exact = sin(k[i]*Pi/2.);
    out.setf (ios::scientific, ios::floatfield); // output in fixed format
    out.precision (18); // 18 digits in doubles
    int width = 20; // set the width for output

    out << "numerical = " << setw(width) << result[i]
    << " exact = " << exact << endl;
}
out.close();

return 0;
// *****
double
my_integrand (double k, double x)
{
    return (k*cos(acos(cos(k*x)))); // just an integrand that takes a while
}

// Integration using Simpson's rule
double simpsons_rule ( int num_pts, double k, double x_min, double x_max,
double (*integrand) (double k, double x) )
{
    double interval = ((x_max - x_min)/double(num_pts - 1)); // called h in notes
    double sum = 0.; // initialize integration sum to zero

    for (int n=2; n<num_pts; n+=2) // loop for odd points
    {
        double x = x_min + interval * double(n-1);
        sum += (4./3.)*interval * integrand(k,x);
    }

    for (int n=3; n<num_pts; n+=2) // loop for even points
    {
        double x = x_min + interval * double(n-1);
        sum += (2./3.)*interval * integrand(k,x);
    }

    // add in the endpoint contributions
    sum += (interval/3.) * (integrand(k,x_min) + integrand(k,x_max));

    return (sum);
}

```