Exercises in Parallel Programming in Fortran and C/C++

Dr. Sara Collins SoSe 2016

Sheet 6

Exercise 1: Integration routine with OpenMP **(3 Points)**

In exercise 4 from sheet 5 you modified your integration module to include the OpenMP directive `omp parallel`.

(a) Produce a new version of the integration routine using the OpenMP directive `omp do`.

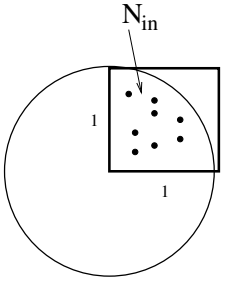
(b) Test that the routine produces the correct results and compare the wall clock times for the previous version (exercise 4 sheet 5) and the new one (using `omp do`).

Exercise 2: Monte-Carlo Integration **(8 Points)**

So far we have only considered 1-dimensional integrals. However, scientific problems usually require numerical computation of n -dimensional integrals, where n is in the range of $n = 2$ (in the simplest case) to n = *very very large*. Especially for the latter, the classical methods of integration (trapezoidal rule, rectangle rule, ...) are unsuitable and one uses so-called *Monte Carlo methods*. These methods use random numbers to estimated the (high-dimensional) integral. The precision of the result increases with the number of random numbers.

We first consider a simple example — the calculation of a quadrant of the area of a circle of radius R = 1.0 (also known as Monte Carlo calculation of the number π). One generates N (eg $N = 10000$) pairs $r_i = (x_i, y_i)$ (with $i = 1, \ldots, N$) of uniformly distributed random numbers x_i and y_i where $0 < x_i, y_i \leq 1$. One counts how many pairs are inside the quadrant of the circle ($R^2 \geq x_i^2 + y_i^2$), denoted N_{in}, and uses

$$\frac{N_{in}}{N} \approx \frac{A_{\text{Circle}}/4}{A_{\square}} \qquad (1)$$



where $A_{\square} = 1.0$.

Hint: Use the formula $A_{\text{Circle}} = \pi R^2$ in order to estimate π using this method.

(a) Write a program that calculates the area of a circle ($n = 2$) with the above described Monte-Carlo method. Display in a plot how the estimate of π approaches the exact result as N is increased.

(b) Generalise your program so that it works for n dimensions – calculating the volume of an n-dimensional sphere (A_V). Equation (2) is modified to

$$\frac{N_{in}}{N} \approx \frac{A_{\text{V}}/2^n}{A_{\square}} \tag{2}$$

Compare it with the exact result given by:

$$V = \begin{cases} \frac{\pi^k}{k!} & \text{für} \quad n = 2k \\ \frac{2^{k+1}\pi^k}{(2k+1)!!} & \text{für} \quad n = 2k+1 \end{cases} \tag{3}$$

wobei $(2k+1)!! = 1 \cdot 3 \cdots (2k-1) \cdot (2k+1)$

(c) Parallelise your program with OpenMP and test that it provides the correct functionality.

Exercise 3: sparse matrix vector multiplication **(10 Points)**

Many scientific problems involve matrix-vector multiplication: $\vec{w} = M\vec{v}$. We consider here the case of a matrix of size $n \times n$. If the matrix is sparse (i.e. mostly contains zeros) then the standard representation of the matrix (as a two-dimensional array) and the implementation of the matrix-vector multiplication (with two nested `do` loops) is very inefficient.
One possibility is to represent the matrix in compressed form using a vector which only contains the non-zero entries using the Compressed Row Storage (CRS) format. This requires three vectors - one holding the non-zero entries (`val`), the second storing the column indices of the non-zero entries (`col`) and the third (`row`) storing the indices of the elements in `val` which start a new row. For example,

$$\begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix} \tag{4}$$

$$\text{val} = (10, -2, 3, 9, 3, 7, 8, 7, 3, 8, 7, 5, 8, 9, 9, 13, 4, 2, -1) \quad (5)$$

$$\text{col} = (1, 5, 1, 2, 6, 2, 3, 4, 1, 3, 4, 5, 2, 4, 5, 6, 2, 5, 6) \quad (6)$$

$$\text{row} = (1, 3, 6, 9, 13, 17, 20) \quad (7)$$

If there are n_{zero} non-zero entries in M ($n_{\text{zero}} = 19$ in the example), then `val` and `col` are of length n_{zero} and `row` is of length $n+1$ (7 in the example), $\texttt{row}(n+1) = n_{\text{zero}}+1$. The matrix-vector multiplication can be performed using

```
do i = 1,n
   do j = row(i), row(i+1)-1
      w(i) = w(i)+val(j)*v(col(j))
   enddo
enddo
```

(a) Write a program which performs matrix-vector multiplication using the standard implementation (using a two-dimensional array for M). Extend the program to convert M into CRS format and perform the matrix-vector multiplication again. Check that the correct functionality is reproduced. You should initialise M to be a sparse matrix, for example, only containing 1% non-zero entries ($n_{\text{zero}} \approx n^2/100$).

(b) Parallelize your program for both cases using OpenMP. Your program should be able to compile with and without OpenMP.

(c) Measure the wallclock time for the multiplication (standard and CRS) for different values of n (from small to very large keeping $n_{\text{zero}}/n^2 \approx$ constant), with and without OpenMP and using a different number of threads. Display the results graphically as a function of n. Give also the performance (in Flops/s) and the efficiency as a function of n.

(d) Test different scheduling options for the do loops.