Parallel Computing

Practicum / Home work

Week 3: OpenMP - part 2

Exercise 3: Computing 3.14159265358979323...

In this exercise we will compute an approximation of π using the identity

$$\int_0^1 \frac{4\,dx}{1+x^2} = 4\arctan(x)\Big|_0^1 = \pi.$$

We divide the integration interval $[0,1]$ into N chunks (N should be very large) and introduce $\Delta x = \frac{1}{N}$. The integral can be approximated numerically using

$$\int_0^1 \frac{4\,dx}{1+x^2} \approx \Delta x \sum_{i=0}^{N-1} \frac{4}{1+((i+0.5)*\Delta x)^2}.$$

The following code fragment performs this task

```
#define f(x) (4.0/(1.0 + (x)*(x))
....
dx = 1.0/N;
sum = 0.0;
for (i=0; i < N; i++)
{
  sum = sum + f(dx*(i+0.5));
}
sum *= dx;
```

Consider the program in the file `pi.c`. It is a sequential program that approximates π using the above code fragment.
(a) Make a parallel version of the program. Answer questions 1 and 2 on Nestor.
(b) Run the program for different problem sizes (N: 10^3, 10^6, 10^{10}, and for each value of N using 1, 4, 8 and 12 threads respectively) and answer questions 3,4,5 on Nestor. Since you are asked to run on different numbers of threads (that might exceed the number of cores of your device), it is required to use the Peregrine cluster.

Exercise 4: LUP decomposition

Suppose we want to solve the following set of linear equations

$$\begin{array}{rcrcrcr} & & 2x_1 & + & 3x_2 & = & 5 \\ 3x_0 & & & + & x_2 & = & 4 \\ 6x_0 & + & 2x_1 & + & 8x_2 & = & 16 \end{array}$$

In this simple case you probably see the solution at a glance: $x_0 = x_1 = x_2 = 1$. For larger systems with hundreds or thousands of equations it is not that obvious. It is convenient to write the system of equations in an equivalent matrix form

$$Ax = b, \text{ where } A = \begin{pmatrix} 0 & 2 & 3 \\ 3 & 0 & 1 \\ 6 & 2 & 8 \end{pmatrix}, \quad x = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \text{ and } \quad b = \begin{pmatrix} 5 \\ 4 \\ 16 \end{pmatrix}.$$

Such a matrix equation is particularly easy to solve if the matrix A is in a lower triangular form or an upper triangular form. For the matrix A at hand, this is clearly not the case. However, for any non-singular $n \times n$ matrix A it is possible to write it as a matrix multiplication of two $n \times n$ matrices L and U

$$A = LU,$$

where L is in lower triangular form and U is in upper triangular form. The process of splitting the matrix A into L and U is called *LU decomposition*. A variant of this process is called *LUP decomposition*, which computes three $n \times n$ matrices L, U and P such that

$$PA = LU,$$

where L is in lower triangular form with 1's on the diagonal, U is in upper triangular form and P is a permutation matrix, i.e. a matrix of zeros and ones that has exactly one entry 1 in each row and column. Since P is a permutation matrix it can be implemented in a 1-dimensional array. For the system at hand we have

$$A = \begin{pmatrix} 0 & 2 & 3 \\ 3 & 0 & 1 \\ 6 & 2 & 8 \end{pmatrix}, \; P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \; L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 1 \end{pmatrix} \text{ and } U = \begin{pmatrix} 6 & 2 & 8 \\ 0 & 2 & 3 \\ 0 & 0 & -\frac{3}{2} \end{pmatrix}.$$

It is easily verified that indeed $PA = LU$.
Such a decomposition can be used to solve the linear equation $Ax = b$ by multiplying both sides by P to get $PAx = Pb$ and thus

$$LUx = \tilde{b}, \text{ where } \tilde{b} = Pb = \begin{pmatrix} 16 \\ 5 \\ 4 \end{pmatrix}.$$

This system is easily solved. We permute b to get $\tilde{b}$, introduce the vector $y = Ux$ and solve

$$Ly = \tilde{b}.$$

This is easy, since L is in lower diagonal form. The method of solving such a system is called *forward substitution*. The solution is given by the recurrence

$$y[i] = \tilde{b}[i] - \sum_{j=0}^{i-1} L[i,j]y[j],$$

where we index in C-style (from 0 to $n-1$). The next step is to solve the system

$$Ux = y,$$

which is done using a similar process called *back substitution*. The solution is given by

$$x[i] = \frac{y[i] - \sum_{j=i+1}^{n-1} U[i,j]x[j]}{U[i,i]}.$$

Consider the program in the file `lup.c`. It is a sequential program that contains a routine for computing the LUP decomposition of a matrix and a routine to solve a linear system using the above method. The routine to compute the LUP decomposition of a matrix is given below

```
void decomposeLUP(int n, real **A, int *P)
{ /* computes L, U, P such that P*A=L*U */
  int h, i, j, k, row;
  real pivot, absval, *ptr;
  for (i=0; i<n; ++i)
  {
    P[i] = i;
  }
  for (k=0; k<n-1; ++k)
  {
    row = -1;
    pivot = 0;
    for (i=k; i<n; ++i)
    {
      absval = (A[i][k] >= 0 ? A[i][k] : -A[i][k]);
      if (absval > pivot)
      {
        pivot = absval;
        row = i;
      }
    }
    if (row == -1) {
      printf ("Singular matrix\n");
      exit(-1);
    }
    /* swap(P[k],P[row]) */
    h = P[k]; P[k] = P[row]; P[row] = h;
    ptr = A[k]; A[k] = A[row]; A[row] = ptr;

    for (i=k+1; i<n; ++i)
    {
      A[i][k] /= A[k][k];
      for (j=k+1; j<n; ++j)
      {
        A[i][j] -= A[i][k]*A[k][j];
      }
    }
  }
}
```

There are two important things to note about this code. First, the LU decomposition is computed in place, i.e., the matrix `A` is overwritten by its decomposition! The matrices L and U are encoded in the same matrix `A`, which saves storage. In other words, the output of the routine would in fact be the matrix $L + U - I$ (where I is the identity matrix). The other thing to note is that the permutation matrix `P` is implemented as a 1-dimensional vector to save storage. Clearly we can encode the matrix of the example given above by the vector `P=[2,0,1]`.

(a) Study the algorithm, and find opportunities to parallelize the algorithm. Implement this using OpenMP (work sharing and other directives).

(b) Solve the $n \times n$ system $Ax = b$ for $n = 10, 100, 1000$, and 5000. Choose for A and b

$$b[i] = 1 \text{ for all } i, \text{ and } A[i,j] = \begin{cases} -2 & \text{if } i = j \\ 1 & \text{if } j = i - 1 \\ 1 & \text{if } j = i + 1 \\ 0 & \text{otherwise} \end{cases}$$

The above matrix was obtained by discretizing the second order ordinary differential equation

$$\frac{d^2}{dx^2} f = -1.$$

Clearly, $f(x) = \frac{-x^2}{2} + cx + d$ is a solution of the differential equation, where c and d are arbitrary constants. You can verify your solutions by computing for each run the values of c and d and check your results.
Answer questions 6, 7 and 8 on Nestor. Run your program and include the requested output under question 9.
(c) For each non singular $n \times n$ matrix A there exists a unique $n \times n$ matrix A^{-1} such that

$$AA^{-1} = I,$$

where I is the $n \times n$ identity matrix. The matrix A^{-1} is called the *inverse matrix* of A. In the example above we have

$$A^{-1} = \frac{1}{18} \begin{pmatrix} 2 & 10 & -2 \\ 18 & 18 & -9 \\ -6 & -12 & 6 \end{pmatrix}.$$

Write a parallel routine `invert` that inverts a matrix. Hint: The equation $AA^{-1} = I$ can be considered as n separate linear systems. Paste the code of the routine under question 10.
(d) Check your solution obtained in (c) by writing a parallel matrix multiplication routine. Use the code you have written to answer question 11.

Exercise 5: Solving the wave equation

The wave equation is an important second-order linear partial differential equation that describes the propagation of a variety of waves, such as sound waves and water waves. It arises in fields such as acoustics and fluid dynamics.

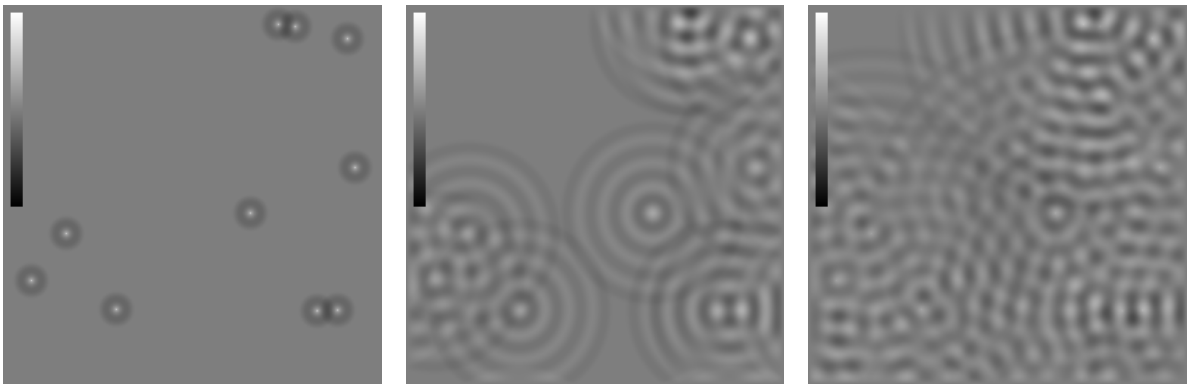

Figure 1: Time evolution of ten sources: left $k = 50$, middle $k = 250$, right $k = 500$

In its simplest form, the 2D wave equation refers to a scalar function $u(x, y, t)$ that satisfies

$$c^2 \left(\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} \right) = \frac{\partial^2 u(x,y,t)}{\partial t^2} \tag{1}$$

where c is a fixed constant equal to the propagation speed of the wave. The first partial derivative with respect to x can easily be approximated by

$$\frac{\partial u(x,y,t)}{\partial x} \approx \frac{u(x + \frac{\Delta x}{2}, y, t) - u(x - \frac{\Delta x}{2}, y, t)}{\Delta x}.$$

By repeating the above formula we get for the second-order partial derivative

$$\frac{\partial^2 u(x,y,t)}{\partial x^2} = \frac{\partial}{\partial x} \left(\frac{\partial u(x,y,t)}{\partial x} \right) \approx \frac{u(x + \Delta x, y, t) - 2u(x,y,t) + u(x - \Delta x, y, t)}{(\Delta x)^2}.$$

Now we introduce the discretization $u[i, j, k] = u(x_0 + i\Delta x, y_0 + j\Delta y, t_0 + k\Delta t)$ yielding

$$\frac{\partial^2 u(x,y,t)}{\partial x^2} \approx \frac{1}{(\Delta x)^2}(u[i+1,j,k] - 2u[i,j,k] + u[i-1,j,k]).$$

For the other second-order partial derivatives we do the same. We sample the grid with equal spatial resolution for x and y by introducing $\Delta x = \Delta y = \delta$. After substitution in equation (1) and some calculus we find the recurrence equation[1]

$$\begin{aligned} u[i,j,k+1] &= \lambda^2 \left(u[i-1,j,k] + u[i+1,j,k] + u[i,j-1,k] + u[i,j+1,k] - 4u[i,j,k] \right) \\ &\quad + 2u[i,j,k] - u[i,j,k-1], \end{aligned}$$

where $\lambda = \frac{c\Delta t}{\delta}$. In the literature on solving wave equations with the above discrete recurrence you can find that it is necessary that $|\lambda| \leq \frac{1}{2}\sqrt{2}$ in order to get a stable approximation. To start this recurrence it suffices to supply initial values $u[i, j, 0]$ and $u[i, j, 1]$ for all i, j.
Consider the files: `wave.c`, `README` and `makemovie`. The file `wave.c` is a sequential C program that solves the above equation for a number of time steps. For each time step an image is generated. With the script `makemovie` you can create a movie called `wave.avi`. Once you made the movie, you can play it with the command

[1] In the literature this recurrence is called the *FDTD (Finite Difference Time Domain)* wave equation.

```
mplayer -v x11 wave.avi
```

(a) Compile the sequential program, using

```
gcc -Wall -mcmodel=medium -O3 -o wave wave.c -lm
```

In the file `README` there are instructions how to run the program. Run the executable and create a movie with the script `makemovie`.

(b) Study the algorithm, and find opportunities to parallelize the algorithm. Implement this using OpenMP and answer questions 12-16 on Nestor.

(c) Perform speedup measurements of your program and make a table. Present your results under question 17 on Nestor.