Exercises for OpenMP

Shaohao Chen
Research Computing @ Boston University
Exercise 1: SAXPY

- **SAXPY**: \( s = a \times x + y \)
  
  adds a scalar multiple of a real vector to another real vector.

- Use OpenMP to parallelize the SAXPY codes.

```c
int i;
#pragma omp parallel for private(i)
for (i = 0; i < n; i++){
    y[i] = a*x[i] + y[i];
}
```
Exercise 2: Matrix Multiplication

• Matrix element

\[ c_{i,j} = \sum_{k=1}^{d} a_{i,k} \cdot b_{k,j} \]
• Use OpenMP to parallelize the matrix-multiplication codes.

Notes: 1. The three matrices are shared data, meaning that all threads can read and write them.
2. Distribute the works of the most outer loop to minimize overheads.

```c
#pragma omp parallel for shared(nra,ncb,nca) private(sum,i,j,k)
for (i = 0; i < nra; i++){
    for (j = 0; j < ncb; j++){
        sum = 0.0;
        for (k = 0; k < nca; k++){
            sum = sum + a[i][k] * b[k][j];
        }
        c[i][j] = sum;
    }
}
```
Exercise 3: Laplacian solver

- Two-dimensional Laplace equation: \( \nabla^2 f(x, y) = 0 \)

- Discretize the laplacian with first-order differential method and express the solution as:

\[
A_{k+1}(i, j) = \frac{A_k(i-1, j) + A_k(i+1, j) + A_k(i, j-1) + A_k(i, j+1)}{4}
\]

- The solution on one point only depends on the four neighbor points:
• **Jacobi iterative algorithm:**

1. Give a trial solution $A$ depending on a provided initial condition.
2. Calculate the new value for every element of the solution, that is $A_{\text{new}}(i,j)$, based on the old values of the four neighbor points.
3. Update the solution, i.e. $A=A_{\text{new}}$,
4. Iterate steps 2 and 3 until converged, i.e. $\max(|A_{\text{new}}(i,j)-A(i,j)|)<\text{tolerance}$.
5. Finally the converged solution is stored at $A$.

• **Use Jacobi iterative algorithm to solve Laplace equation.**
• Use OpenMP to parallelize the program for Laplacian solver.

```c
int iter = 0;
while ( error > tol && iter < iter_max ) { // iterate until converged
    error = 0.0;
    #pragma omp parallel for shared(m, n, Anew, A) private(i,j)
    for ( j = 1; j < n-1; j++ ) for ( i = 1; i < m-1; i++ ) {
        error = fmax( error, fabs(Anew[j][i] - A[j][i])); // calculate the maximum error
    }
    #pragma omp parallel for shared(m, n, Anew, A) private(i,j)
    for ( j = 1; j < n-1; j++ ) for ( i = 1; i < m-1; i++ ) {
        A[j][i] = Anew[j][i]; // Update the solution
    }
    iter++;
}
```
Exercise 4: Calculate the value of Pi

- Calculate pi with integration method

\[ \int_{0}^{1} \frac{4.0}{1 + x^2} \, dx = \pi \]

- Numerically, we can approximate the value of pi as the sum of a number of rectangles.

\[ \sum_{i=0}^{N} F(x_i) \Delta x \approx \pi \]
• Use OpenMP to parallelize the program for calculating pi.

Notes: Use reduction clause to avoid a data racing condition.

```c
#pragma omp parallel for default(shared) private(i,x) reduction(+:sum)
{
    for (i = 0; i < n; i++) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
}
```
Exercise 5: Matrix-vector multiplication (in Fortran)

• Calculate \( a = B \cdot c \), where \( a \) is an \( m \) dimension vector, \( c \) is an \( n \) dimension vector and \( B \) is an \( m \times n \) dimension matrix.

• Serial Fortran code

```fortran
do i = 1, m
   a(i) = b(i,1)*c(1)
   do j = 2, n
      a(i) = a(i) + b(i,j)*c(j)
   end do
end do
```
Use OpenMP to parallelize the code for matrix-vector multiplication (version 1)

```c
!$OMP PARALLEL DO DEFAULT(shared) PRIVATE(i,j)
  do i = 1, m
    a(i) = b(i,1)*c(1)
    do j = 2, n
      a(i) = a(i) + b(i,j)*c(j)
    end do
  end do
end do
!$OMP END PARALLEL DO
```
• Use OpenMP to parallelize the code for matrix-vector multiplication (version 2, only for Fortran)

```fortran
!$OMP PARALLEL DEFAULT(shared) PRIVATE(i,j)
  !$OMP WORKSHARE
  a(1:m) = b(1:m,1)*c(1)
  !$OMP END WORKSHARE
  !$OMP DO REDUCTION(+:a)
  do j = 2, n
    do i = 1, m
      a(i) = a(i) + b(i,j)*c(j)
    end do
  end do
!$OMP END DO
!$OMP END PARALLEL
```