CS320/CSE302/ECE392: INTRODUCTION TO PARALLEL PROGRAMMING FOR SCIENTISTS AND ENGINEERS

Spring 1999

Machine Problem 3

Assigned: March 1, 1999
Due: March 13, 1999.

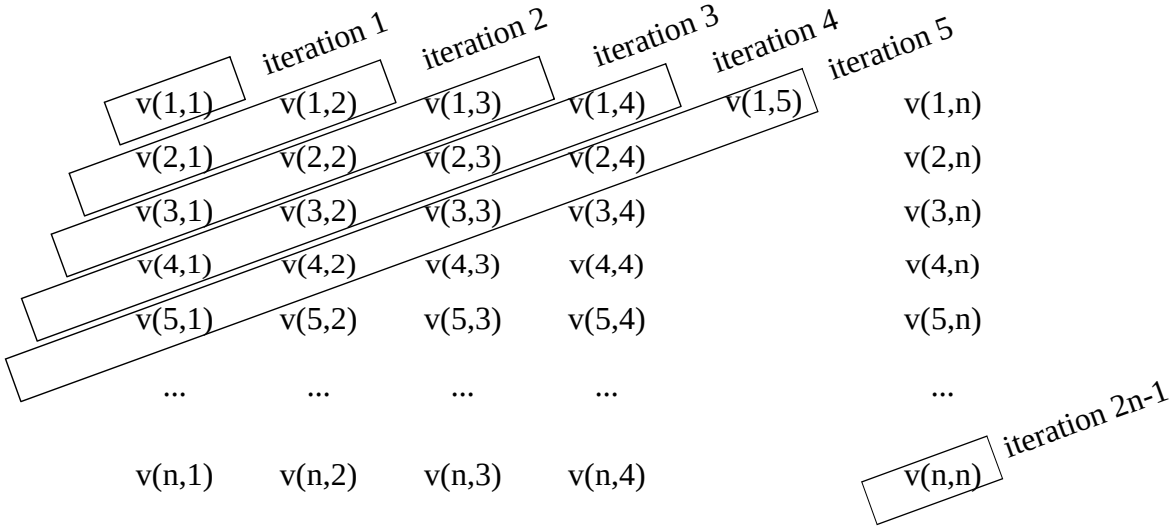
The objective of this MP is to implement a parallel OpenMp version of the following Fortran 77 code:

```
real v(100,100). max_change, close_enough
read *,n
read *,(v(1,i),i=1,n)
read*,(v(i,1),i=1,n-1)
read*,(v(i,n),i=2,n-1)
read *,(v(n,i),i=1,n)
close_enough = 1.0 e-5
max_change = close_enough + 1.0
do while (max_change .gt. close_enough)  ! Iterate until
   max_change = 0      ! The largest error we've seen. Initially 0
      do i=2,n-1       ! Do all rows, but not outer boundary rows
         do j=2,n-1    ! Do all elements in a row, but not boundary
            old_value=v(i,j)
            v(i,j)=(v(i-1,j)+v(i+1,j)+v(i,j-1)+v(i,j+1)/4
            max_change=max(max_change,abs(old_value-v(i,j)))
         end do
      end do
end do
do i=1,n
   print *,v(i,j),j=1,n)
end do
end
```

This code solves the LaPlace equation $\frac{\partial^2}{\partial x^2}f + \frac{\partial^2}{\partial y^2}f = 0$ implicitly on a square region using an *n* by *n* mesh. The solution will be in vector `v`, with the boundary values (`v(1,:), v(n,:), v(:,1), v(:,n)`) kept constant throughout the computation. As you can see the program proceeds by computing the value of each interior point in the mesh as the average of the four neighbors. The program stops when the maximum difference between corresponnding elements of two successive values of `v` is smaller than `close_enough`.

The assignment is to develop a parallel version of this program or two versions if you registered for one unit.

Parallel version 1 (to be implemented by everybody).
This version does not contain any race conditions and produce the same result as the code presented above. Basically, the idea is to compute the doubly-nested inner loops (the two loops inside the `do while`) as a sequence of iterations. This sequence of iterations can be represented graphically as follows:



Notice that the elements of v in each *iteration* can be computed using a `parallel do`. There will be no race conditions and the result will be the same as that of the sequential code as long as the iterations are computed in the order shown. Notice also that the value of `max_change` for each iteration will have to be computed as a parallel reduction.

Parallel version 2.
Those registered for one unit should also implement a parallel version that computes the values of v in two (parallel) iterations. The idea is to divide the elements of v into black and white elements which correspond to the squares in a checkerboard. The first iteration should compute the white elements and the second the black elements. Notice that the result will not be necessarily the same as that of the sequential program above, but it is still valid. Notice also that the value of `max_change` has to be computed as a parallel reduction for each iteration.