ASSIGNMENT 4 SOLUTIONS
HPC1 Fall 2013

Due Date: ***Tuesday, November 5***
(please submit your report electronically to the instructor, in one PDF file, as *hw4-yourUBitname.pdf*)

Problem 1: Consider the Laplace equation in two dimensions (we will consider Dirichlet boundary conditions in the unit interval),

$$\nabla^2\phi = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\phi = 0. \tag{1}$$

Now take the case of a uniform square grid of M points in each direction, and using a simple two-point numerical derivative, it is not difficult to show that the optimum value of ϕ at the point $(i, j) = (x_i, y_j)$ is given by:

$$\phi_{ij} \simeq \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i,j+1} + \phi_{i-1,j} + \phi_{i,j-1}\right) \qquad (i, j) \in (1..M, 1..M). \tag{2}$$

The *relaxation method* consists of iterating this equation to obtain a new, improved solution from the previous iteration:

$$\phi^{n+1}_{ij} \simeq \frac{1}{4}\left(\phi^n_{i+1,j} + \phi^n_{i,j+1} + \phi^n_{i-1,j} + \phi^n_{i,j-1}\right) \qquad (i, j) \in (1..M, 1..M) \tag{3}$$

A simple summary of the technique (otherwise known as Jacobi iteration):

1. Apply a square lattice with uniform spacing - label the points (i, j).
2. Apply the fixed boundary condition values.
3. Make an initial guess for the interior points $\phi^0_{i,j}$.
4. Iterate until convergence, using the "cross" scheme in Eq. 3 (note that better schemes are available - feel free to derive one, or look them up in standard references).

a. Write a serial code to perform the above solution for the problem on the unit square $(0 \le x, y \le 1)$ with boundary conditions

$$\begin{aligned}\phi(x, 0) &= \sin(\pi x),\\ \phi(x, 1) &= \sin(\pi x)e^{-\pi},\\ \phi(0, y) &= \phi(1, y) = 0.\end{aligned}$$

You can verify that the analytic solution to this problem is $\phi(x, y) = \sin(\pi x)e^{-\pi y}$.

Solution:
Solution combined with part **c** with a code listing included at the end of these solutions.

b. How does the number of Jacobi iterations required for convergence in your code depend on the grid size? Show a plot of this behavior. Take a reasonable threshold for convergence, say to where the L2-norm of the solution difference between iterations reaches 10^{-5}.

Solution:

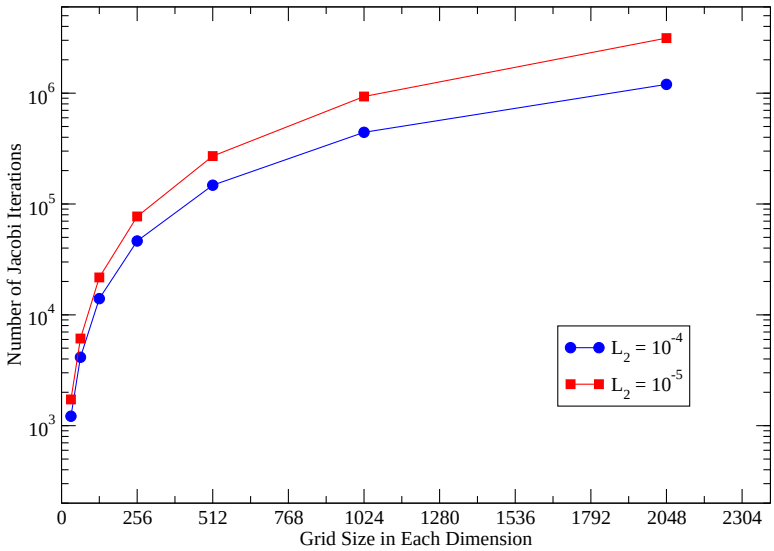

Figure 1: Number of iterations required as a function of the grid size for the simple Jacobi solver, using an L_2 norm of 10^{-4} and 10^{-5}.

Plotted in Fig. 1 is the number of Jacobi iterations as a function of the grid size, for a tolerance of 10^{-4} and 10^{-5} (a stricter tolerance will require more iterations) of the L2-norm between the solution iterates. Note the strong nonlinear growth (in fact proportional to the number of grid points squared) owing to the nature of relaxation methods - the solution actually propagates inward from the domain boundary, so as the mesh size grows, so must the number of iterations, roughly as the square of the number of mesh points. Note that the error, as measured from the analytic solution, is considerably larger than this iterative tolerance (to improve the error, reduce the tolerance).

You should verify, of course, that you are getting a "good" solution relative to the analytic one - see Fig. 2 below (a good solution will require an even stricter convergence criterion).

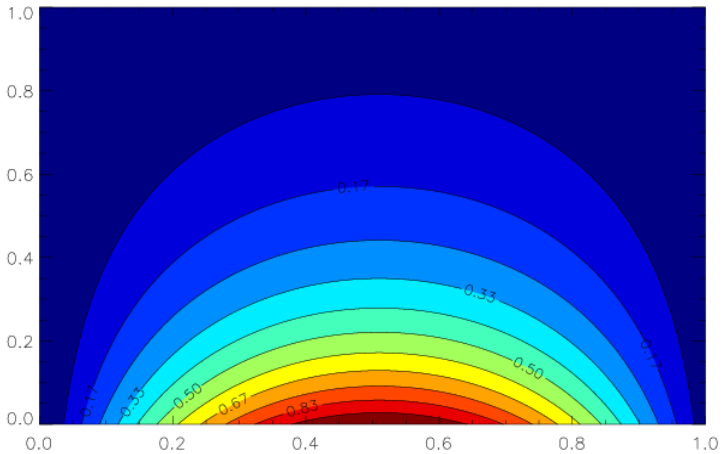

Figure 2: Color contour of solution using Jacobi solver, plotted using IDL.

c. Parallelize your solver using OpenMP.

Solution:

See the program listing at the end of the solutions.

d. Take a grid size that is significant enough to require some time (say 1000 or so, but feel free to innovate), and do a study in parallel scalability - how well does your parallel program scale with additional processors?

Solution:

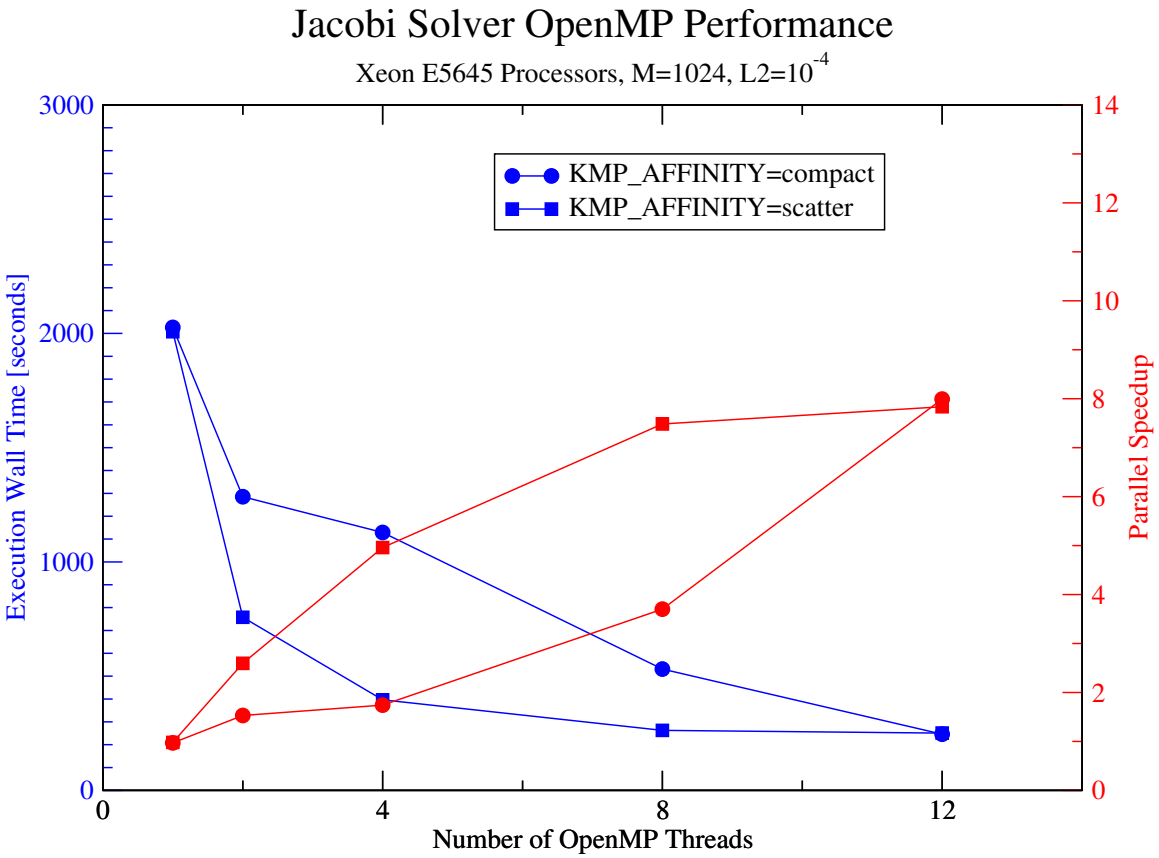

Figure 3: OpenMP parallel performance of the simple Jacobi solver on the **UB/CCR** nodes (12-core systems).

The performance shown in Fig. 3 is not bad - a speedup of about 8 on all 12 cores. Note the variation shown by adjusting the affinity option (which in the Intel compiler is done by setting the `KMP_AFFINITY` environment variable) such that the OpenMP threads are placed on adjoining cores (`compact` setting), or filled in using a scatter approach (`scatter` setting) on the other processors (recall that these 12-core nodes have two 6-core processors, with a shared L3-cache on each processor). What we can see from the difference between the `compact` and `scatter` approaches is there is very significant contention for memory in this code (since our matrices are 1024x1024, each one takes up about 8MB of memory, so the two matrices together can not fit into the L3 cache). Using the `scatter` approach for thread placement gives a performance boost for intermediate values of `OMP_NUM_THREADS`, as the second processor's L3 cache comes into use, but once the node is full, a `compact` strategy will actually work best. What happens if you fit both matrices into the L3 cache? You can see from Fig. 4 the

results from running the same grid size on one of the Intel 32-core nodes, which have 24MB of L3 cache for each of the four (8 cores each) processors. If you increase the problem size

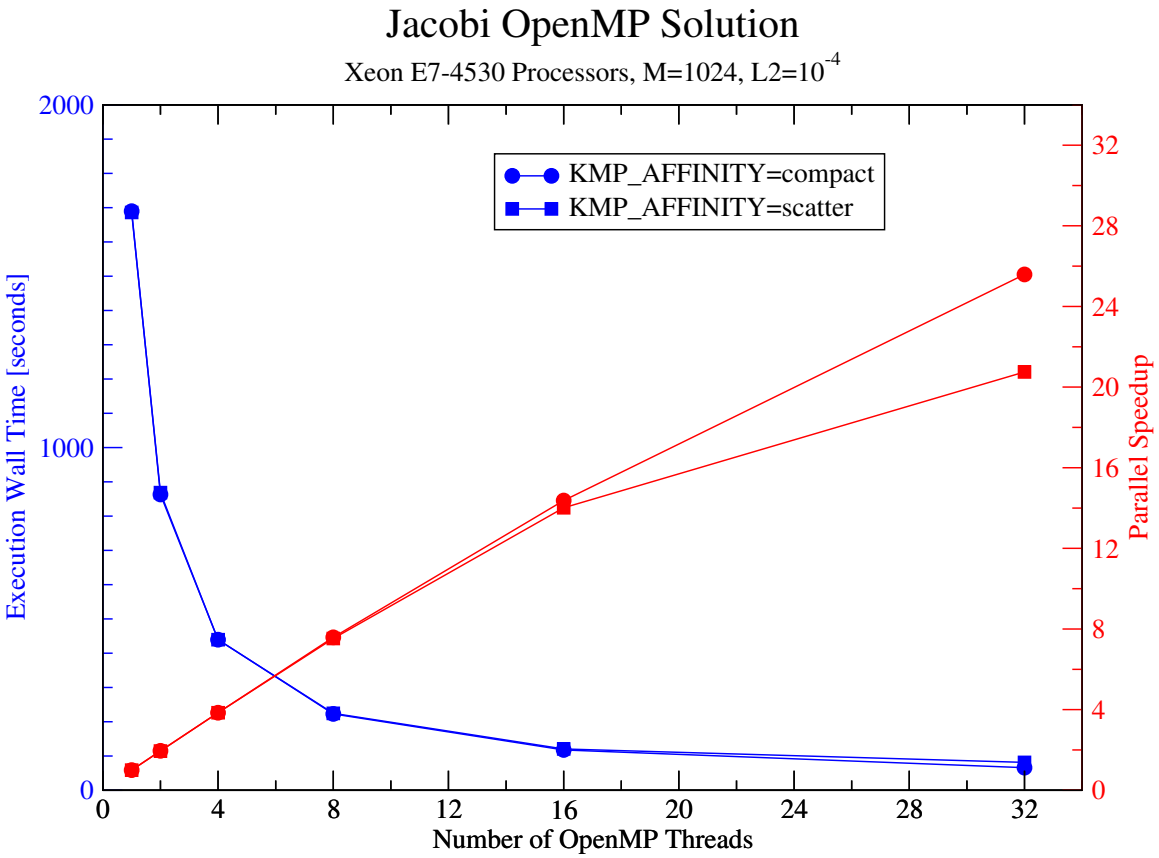

Figure 4: OpenMP parallel performance of the simple Jacobi solver on the **UB/CCR** nodes (32-core systems).

still further, however, the same behavior that we saw in Fig. 3 will manifest itself as the matrices grow too large to completely fit into the cache, and the cores contend for access to main memory.

```
1  MODULE serialMOD
2    implicit none
3    integer,parameter :: dp=selected_real_kind(2*PRECISION(1.0))
4    real(kind=dp),parameter :: PI=3.1415926535897932
5
6  CONTAINS
7    subroutine BC1(sol,m)
8      !
9      ! sol = current solution
10     ! m = 2d grid extent (square assumed 0:m+1)
11     !
12     ! BC: 0<= x,y <=1
13     !  y=0: sol = sin(\pi*x)
14     !  y=1: sol = exp(-\pi)*sin(\pi*x)
15     !
16     implicit none
17     integer :: m
18     real(kind=dp) :: sol(0:m+1,0:m+1)
19
20     integer :: i
21     real(kind=dp) :: x
22
23     ! BCs initialized to Dirichlet values, rest of points
24     !  set to zero
25     sol=0.0_dp
26     do i=0,m+1
27        x=dble(i)/dble(m+1)
28        sol(i,0)=sin(PI*x)
29        sol(i,m+1)=sin(PI*x)*exp(-PI)
30     end do
31     return
32   end subroutine BC1
33 end MODULE serialMOD
34
35 program laplace_omp
36   USE serialMOD
37   USE omp_lib
38   implicit none
39
40   integer,parameter :: MAXITER=15000000,outiter=10000
41   !real(kind=dp),parameter :: tolerance=1.0d-6
42   real(kind=dp) :: tolerance
43   integer :: i,j,m,iter,nthreads,ctick1,ctick2,ctickrate,ctickmax
44   real(kind=dp) :: t0,t1,t2,t3,fops,delta,tdelta,residual,exacterr,exactsol
45   real(kind=dp),allocatable :: sol(:,:),oldsol(:,:)
46   real(kind=dp) :: dsecnd
47   external dsecnd
48
49   print*,'Enter grid extent, m: (0:m+1) and tolerance'
50   read(*,*) m,tolerance
51   print*,'Using grid of size: ',m,', 0,m+1 used for BCs.'
52   print*,'Using Tolerance = ',tolerance
53
54   ALLOCATE(sol(0:m+1,0:m+1),oldsol(0:m+1,0:m+1))
55   call BC1(sol,m) ! Apply Dirichlet boundary conditions
56   oldsol=sol
57
58 #ifdef _OPENMP
59   t0=OMP_GET_WTIME()
60 #else
61   t0=dsecnd()
62 #endif
63   iter=0
64   residual=1.d6
65   do while ( (residual > tolerance).and.(iter <= MAXITER) )
66      iter = iter+1
67      ! update interior points based on "+" points
68      residual = 0.0_dp
69 !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(i,j,delta)
70 !$OMP DO REDUCTION(+:residual)
71      do j=1,m
72         do i=1,m
73            sol(i,j)=0.25_dp*(oldsol(i+1,j)+oldsol(i-1,j)+oldsol(i,j+1)+oldsol(i,j-1))
74            delta = sol(i,j)-oldsol(i,j)
75            residual = residual+delta*delta
```

```
76          end do
77       end do
78  !$OMP END DO
79  !$OMP DO
80       do j=1,m
81          do i=1,m
82             oldsol(i,j)=sol(i,j)
83          end do
84       end do
85  !$OMP END DO
86  !$OMP END PARALLEL
87       residual = sqrt(residual)
88       if(iter.eq.1.or.MOD(iter,outiter).eq.0) &
89           write(*,'(" iter = ",i6," residual = ",e12.4)') iter,residual
90    end do
91    fops = 8.0*m*m
92    fops = fops*iter
93  #ifdef _OPENMP
94    t1=OMP_GET_WTIME()
95  #else
96    t1=dsecnd()
97  #endif
98    write(*,'("Total Runtime[s], (under)estimated Mflop/s: ",f14.6,f12.2)') t1-t0, &
99          fops/(1.e6*(t1-t0))
100   !
101   ! Output results for plotting
102   !
103   exacterr = -1.0
104   do i=0,m+1
105      do j=0,m+1
106         exactsol = SIN( PI*DBLE(i)/DBLE(m+1) )*EXP( -PI*DBLE(j)/DBLE(m+1) )
107         exacterr = MAX( exacterr, ABS(exactsol-sol(i,j)) )
108  !        print*, 'i,j,err = ',i,j,exactsol-sol(i,j)
109      end do
110   end do
111   open(unit=10,file="laplace_omp.dat",status='unknown',form='unformatted')
112   print*,' Max value in sol: ',MAXVAL(sol)
113   print*,' Min value in sol: ',MINVAL(sol)
114   print*,' Max error from analytic solution: ',exacterr
115   write(10) ((sngl(sol(i,j)),i=0,m+1),j=0,m+1)
116   close(10)
117   DEALLOCATE(sol,oldsol)
118   print*,'Solution reached in ',iter,' iterations.'
119 end program laplace_omp
```