```
! Solve_master

module solver

  use problem_class, only : problem_type

  implicit none
  private

  public  :: solve

contains

!=======================================================================
! Master Solver
!
subroutine solve(pb)

  use output,       only : screen_init, screen_write, ox_write, ot_write
  use constants, only: MY_RANK, MPI_parallel

  type(problem_type), intent(inout)  :: pb

  if (MY_RANK==0) call screen_init(pb)

  ! Time loop
  do while (pb%it /= pb%itstop)
    pb%it = pb%it + 1
    call do_bsstep(pb)
! if stress exceeds yield call Coulomb_solver ! JPA Coulomb quick and dirty
!                         or (cleaner version) do linear adjustment of
!                         timestep then redo bsstep
!                         or (cleanest version) iterate tiemstep adjustment and
!                         bsstep until stress is exactly equal to yield
    call update_field(pb)
    call ot_write(pb)
    call check_stop(pb)   ! here itstop will change
    !--------Output onestep to screen and ox file(snap_shot)
    if(mod(pb%it-1,pb%ot%ntout) == 0 .or. pb%it == pb%itstop) then
        if (MY_RANK==0) call screen_write(pb)
    endif
    if (MY_RANK==0) call ox_write(pb)
  enddo

end subroutine solve

!=======================================================================
! check stop:
!
subroutine check_stop(pb)

  use output, only : time_write

  type(problem_type), intent(inout) :: pb

  double precision :: vmax_old = 0d0, vmax_older = 0d0
  save vmax_old, vmax_older

  if (pb%itstop == -1) then
      !         STOP soon after end of slip localization
    if (pb%NSTOP == 1) then
      if (pb%ot%llocnew > pb%ot%llocold) pb%itstop=pb%it+2*pb%ot%ntout

      ! STOP soon after maximum slip rate
    elseif (pb%NSTOP == 2) then

      if (pb%it > 2 .and. vmax_old > vmax_older .and. pb%v(pb%ot%ivmax) < vmax_old)
 &
          pb%itstop = pb%it+10*pb%ot%ntout
      vmax_older = vmax_old
      vmax_old = pb%v(pb%ot%ivmax)

        !         STOP at a slip rate threshold
```

```
    elseif (pb%NSTOP == 3) then
      if (pb%v(pb%ot%ivmax) > pb%tmax) pb%itstop = pb%it    !here tmax is threshhold
 velocity

        !         STOP if time > tmax
    else
      call time_write(pb)
      if (pb%tmax > 0.d0 .and. pb%time > pb%tmax) pb%itstop = pb%it
    endif
  endif

end subroutine check_stop


!=====================================================================
! pack, do bs_step and unpack
!
! IMPORTANT NOTE : between pack/unpack pb%v & pb%theta are not up-to-date
!
subroutine do_bsstep(pb)

  use derivs_all
  use ode_bs

  type(problem_type), intent(inout) :: pb

  double precision, dimension(pb%neqs*pb%mesh%nn) :: yt, dydt, yt_scale

  ! Pack v, theta into yt
! yt(2::pb%neqs) = pb%v(pb%rs_nodes) ! JPA Coulomb
  yt(2::pb%neqs) = pb%v
  yt(1::pb%neqs) = pb%theta
  dydt(2::pb%neqs) = pb%dv_dt
  dydt(1::pb%neqs) = pb%dtheta_dt
  if ( pb%neqs == 3) then           ! Temp solution for normal stress coupling
    yt(3::pb%neqs) = pb%sigma
    dydt(3::pb%neqs) = pb%dsigma_dt
  endif

  ! this update of derivatives is only needed to set up the scaling (yt_scale)
  call derivs(pb%time,yt,dydt,pb)
  yt_scale=dabs(yt)+dabs(pb%dt_try*dydt)
  ! One step
  call bsstep(yt,dydt,pb%neqs*pb%mesh%nn,pb%time,pb%dt_try,pb%acc,yt_scale,pb%dt_did
,pb%dt_next,pb)
!PG: Here is necessary a global min, or dt_next and dt_max is the same in all proces
sors?.
  if (pb%dt_max >  0.d0) then
    pb%dt_try = min(pb%dt_next,pb%dt_max)
  else
    pb%dt_try = pb%dt_next
  endif

! Unpack yt into v, theta
!  pb%v(pb%rs_nodes) = yt(2::pb%neqs) ! JPA Coulomb
  pb%v = yt(2::pb%neqs)
  pb%theta = yt(1::pb%neqs)
  pb%dv_dt = dydt(2::pb%neqs)
  pb%dtheta_dt = dydt(1::pb%neqs)
  if ( pb%neqs == 3) then           ! Temp solution for normal stress coupling
    pb%sigma = yt(3::pb%neqs)
    pb%dsigma_dt = dydt(3::pb%neqs)
  endif

end subroutine do_bsstep


!=====================================================================
! Update field: slip, tau, potency potency rate, crack,
!
subroutine update_field(pb)

  use output, only : crack_size
```

```
  use friction, only : friction_mu

  type(problem_type), intent(inout) :: pb

  integer :: i,ix,iw
  double precision :: vtemp

  nLocal=nnLocal_perproc(MY_RANK)
  nnGlobal=sum(nnLocal_perproc)

  ! Update slip, stress.
  pb%slip = pb%slip + pb%v*pb%dt_did
  pb%tau = pb%sigma * friction_mu(pb%v,pb%theta,pb) + pb%coh
  ! update potency and potency rate
  pb%pot=0d0;
  pb%pot_rate=0d0;
  if (pb%mesh%dim == 0 .or. pb%mesh%dim == 1) then
    pb%pot = sum(pb%slip) * pb%mesh%dx
    pb%pot_rate = sum(pb%v) * pb%mesh%dx
  else
    do iw=1,pb%mesh%nw
      do ix=1,pb%mesh%nx
        i=(iw-1)*pb%mesh%nx+ix
        pb%pot = pb%pot + pb%slip(i) * pb%mesh%dx * pb%mesh%dw(iw)
        pb%pot_rate = pb%pot_rate + pb%v(i) * pb%mesh%dx * pb%mesh%dw(iw)
      end do
    end do
  endif
!PG: is this (crack_size) only in local processor ?.
  ! update crack size
  pb%ot%lcold = pb%ot%lcnew
  pb%ot%lcnew = crack_size(pb%slip,pb%mesh%nn)
  pb%ot%llocold = pb%ot%llocnew
  pb%ot%llocnew = crack_size(pb%dtau_dt,pb%mesh%nn)
  ! Output time series at max(v) location
 !PG, only local or global mx ?.
  vtemp=0d0
  do i=1,pb%mesh%nn
     if ( pb%v(i) > vtemp) then
       vtemp = pb%v(i)
       pb%ot%ivmax = i
     end if
  end do
! Finding global vmax
 if (MPI_parallel) then

   call max_allproc(pb%ot,pb%ot%ivmaxglob)

 endif

end subroutine update_field


!=====================================================================
! Collect global fault nodes to MY_RANK=0 for outputs
  subroutine pb_global(pb)

  use fault_stress, only: nnLocal_perproc,nnoffset_glob_perproc

  use constants, only: NPROCS,MY_RANK

  type(problem_type), intent(inout) :: pb
  integer :: nLocal,nnGlobal

  nLocal=nnLocal_perproc(MY_RANK)
  nnGlobal=sum(nnLocal_perproc)

  allocate(pb%v_glob(nnGlobal),pb%theta_glob(nnGlobal),pb%tau_glob(nnGlobal),&
           pb%slip_glob(nnGlobal),pb%sigma_glob(nnGlobal),pb%dv_dt_glob(nnGlobal),&
           pb%dtheta_dt_glob(nnGlobal),pb%dtau_dt_glob(nnGlobal))

  pb%v_glob=0
  pb%theta_glob=0
```

```
  pb%tau_glob=0
  pb%slip_glob=0
  pb%sigma_glob=0
  pb%dv_dt_glob=0
  pb%dtheta_dt_glob=0
  pb%dtau_dt_glob=0

  call gather_allvdouble_root(pb%v,nLocal,pb%v_glob,nnLocal_perproc, &
                              nnoffset_glob_perproc,nnGlobal,NPROCS)
  call gather_allvdouble_root(pb%dv_dt,nLocal,pb%dv_dt_glob,nnLocal_perproc, &
                              nnoffset_glob_perproc,nnGlobal,NPROCS)
  call gather_allvdouble_root(pb%theta,nLocal,pb%theta_glob,nnLocal_perproc, &
                              nnoffset_glob_perproc,nnGlobal,NPROCS)
  call gather_allvdouble_root(pb%dtheta_dt,nLocal,pb%dtheta_dt_glob,nnLocal_perproc,
 &
                              nnoffset_glob_perproc,nnGlobal,NPROCS)
  call gather_allvdouble_root(pb%tau,nLocal,pb%tau_glob,nnLocal_perproc, &
                              nnoffset_glob_perproc,nnGlobal,NPROCS)
  call gather_allvdouble_root(pb%dtau_dt,nLocal,pb%dtau_dt_glob,nnLocal_perproc, &
                              nnoffset_glob_perproc,nnGlobal,NPROCS)
  call gather_allvdouble_root(pb%slip,nLocal,pb%slip_glob,nnLocal_perproc, &
                              nnoffset_glob_perproc,nnGlobal,NPROCS)
  call gather_allvdouble_root(pb%sigma,nLocal,pb%sigma_glob,nnLocal_perproc, &
                              nnoffset_glob_perproc,nnGlobal,NPROCS)

  end subroutine pb_global
!=====================================================================
end module solver
```