Modeling of a Graphene Membrane Rupture with DFTB and Improving its Computational Efficiency

By: Krystle Reiss, Jacob Blazejewski,
Jacek Jakowski, Kwai Wong

CSURE Program 2015

Abstract

Density Functional Tight Binding (DFTB) is being used to find the cause of the catastrophic rupture of a graphene membrane under the effect of an electric field. Efforts are also being made to increase the computational efficiency of the program by replacing LAPACK calls with ScaLAPACK calls.

Introduction

DFTB+ is being used to determine the cause of a graphene membrane rupture under the influence of an electric field.[1] When an electric field of 3 V/nm is applied to a graphene membrane suspended in a 1 M KCl solution, the membrane ruptures catastrophically, sometimes ripping completely in half. Several different variations of graphene membranes are being tested under varying conditions using molecular dynamics (MD) simulations.

Carbons	Hydrogens	Corners	Flat or Warped
218	40	Free	Flat
			Warped
		Frozen	Flat
			Warped
	58	Free	Flat
			Warped
		Frozen	Flat
			Warped
508	62	Free	Flat
			Warped
		Frozen	Flat
			Warped
	90	Free	Flat
			Warped
		Frozen	Flat
			Warped

Table 1 gives all types of membranes used in MD simulations

Unfortunately running these DFTB calculations is extremely computationally expensive, with the most demanding calculations being linear algebra operations. The time spent on these operations is divided amongst evaluating forces, determining electronic structure and moving and handling the matrices to be used in the operations.

The DFTB code utilizes Linear Algebra Package (LAPACK) functions to perform these calculations. Under these routines DFTB calculations of certain systems can still take far to long to be practical. In an attempt to speed up the software's calculations the LAPACK routines are therefore being replaced with Scalable LAPACK (ScaLAPACK) routines.

Modeling of Graphene

Graphene, the two-dimensional form of graphite, is a fairly new material with many fascinating properties. Stronger than its equivalent weight in steel and very elastic, graphene is composed of a highly conjugated system of carbon atoms giving it 150 times the mobility of silicon. This means that graphene is an extremely good conductor. However, graphene is not yet a viable replacement for silicon switches in electronic devices, as graphene has no band gap. Silicon is a semiconductor, meaning that its band gap is just small enough for electrons to cross it if an electric field of suitable magnitude is applied. When no electric field applied, silicon's electrons are unable to cross the gap. Graphene's lack of a band gap makes it metallic and electrons can move between HOMO and LUMO energy levels without the application of an electric field. Since graphene cannot be activated and deactivated like silicon can, it cannot generate binary code, which inhibits its ability to replace silicon in electronics.

Dr. Ivan Vlassiouk has been experimenting with applying electric fields to circular graphene membranes suspended in a 1 M potassium chloride aqueous solution. When an electric field with a strength of 3 V/nm is applied to these membranes, they rupture. There is no correlation between membrane size and rupture. The tear is so catastrophic, sometimes ripping the membrane entirely in half, that its cause cannot be determined. It is possible that there are defects in the membrane, such as a Stone-Wales defect or a vacancy site,[2] or it may be that an ion is forced through the membrane, causing the rupture.

Computational methods, specifically DFTB+, are being used to determine why the membrane is rupturing. Because of its efficiency, DFTB is ideal for this type of simulation. Molecular dynamics (MD) simulations were set to run for femtosecond 5000 timesteps but were limited to 24-hour runtimes due to scheduling protocols. MD simulations were run using the VelocityVerlet driver with the NoseHoover thermostat set to 300 K and the coupling strength was 600 cm^{-1}. The Hamiltonian was DFTB with an SCC tolerance of $1.0 \bullet 10^{-6}$. The Fermi filling temperature was originally set to 0 K, but the SCC failed to converge at this temperature. When increased to 300 K, convergence was achieved, so this temperature was used throughout the rest of the simulations. Figure 1A and 1B show the results of these basic MD simulations.

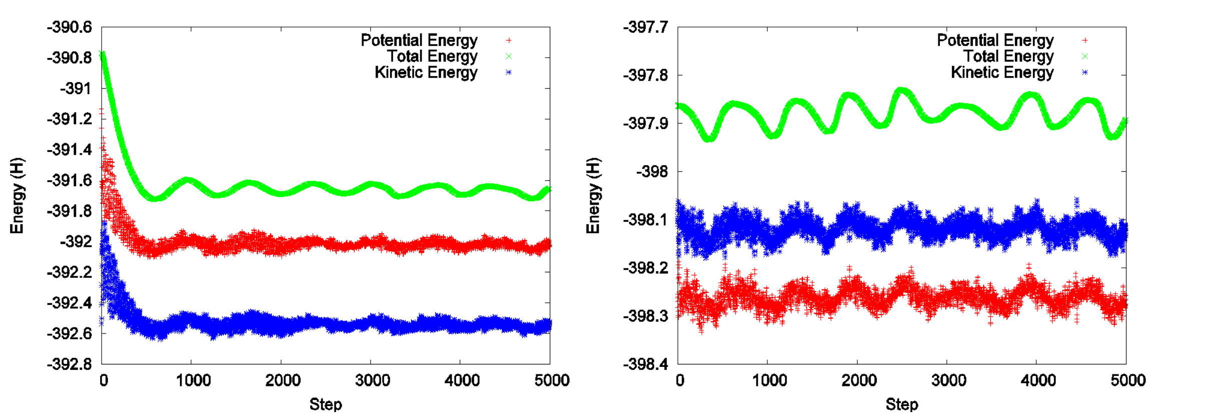

Figure 1A (left) show the results of a 5000 step MD simulation on a 218 carbon sheet where each edge carbons has been saturated with a single hydrogen. **Figure 1B** (right) shows the same simulation conducted on a similar sheet where the carbons along the armchair edge were saturated with two hydrogens each. In both, kinetic energy values have been shifted down ~400 H for scaling purposes.

Unfortunately, larger sheet (508 and 1018 carbons) take much more time and do not finish the 5000 step MD simulation within the 24-hour wall clock limit. The 508-carbon membrane is able to complete 1500 to 2500 steps, depending on edge saturation, while the

1006-carbon membrane does not even finish 300 steps. Even utilizing the GPU, the 1006-carbon membrane only increased it's completed steps by ~50%. This led to the largest membrane being dropped from simulations, even though it would have provided the most realistic results. The 508-carbon membranes completed enough steps for the results to be considered meaningful. The smallest membrane, 218 carbons, finished the MD simulation within twelve hours.

In addition to differently sized membranes, the effects of constraints and waves were also evaluated. Two types of constraints were tested: freezing all membrane edges and freezing only the corners. Freezing entire edges was deemed to be too limiting as it prevented the natural dynamics of the system and so was abandoned. Frozen corners allowed for adequate membrane movement and membranes constrained in this way were tested alongside unconstrained membranes. To create waves in the membranes, an MD simulation was run applying a temperature of 2000 K to the system. This caused the membrane to spasm and warp, creating the desired waves. This allowed for points of polarity to form when an electric field was applied. It should be noted that unless constrained, the membranes reassumed their planar forms upon the removal of the extreme temperature.

To simulate the application of a 3 V/nm electric field, two point charges (±15 eV) were placed on either side of the membrane 10.00 nm away along the y axis (normal to the membrane). No significant effects were noted except a minor flattened region in the unconstrained planar membrane (Figure 2A). Initial drops in energy show the membranes moving into their ideal geometries. The warped membrane (2C & 2D) have larger initial energy drops as they attempt reassume their planar shapes.

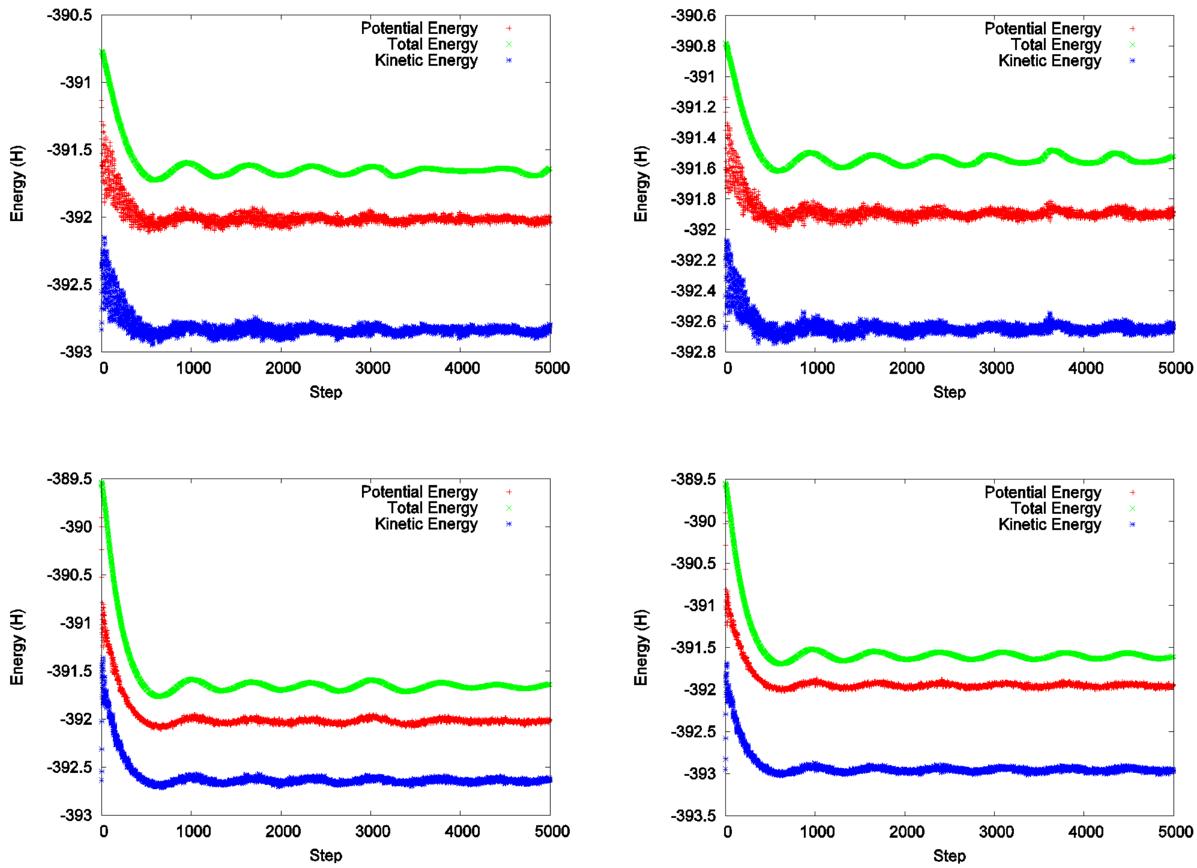

Figure 2A (top left) shows 218 C:40 H planar membrane under a 3 V/nm electric field. **Figure 2B** (top right) shows the same membrane with corners frozen. **Figure 2C** (bottom left) shows a warped membrane under the same conditions as 2A. **Figure 2D** (bottom right) shows a warped membrane under the same conditions as 2B. As in Figure 1, all kinetic energies were shifted down for scaling.

With no significant effects caused by the 3 V/nm field, the field strength was increased to 30 V/nm by increasing both of the point charges tenfold. Although it did not break any of the membranes, this did cause significant movement in the unconstrained membranes. These twisted to align themselves with the field, shown by the second major drop in energy in Figure 3A & 3C.

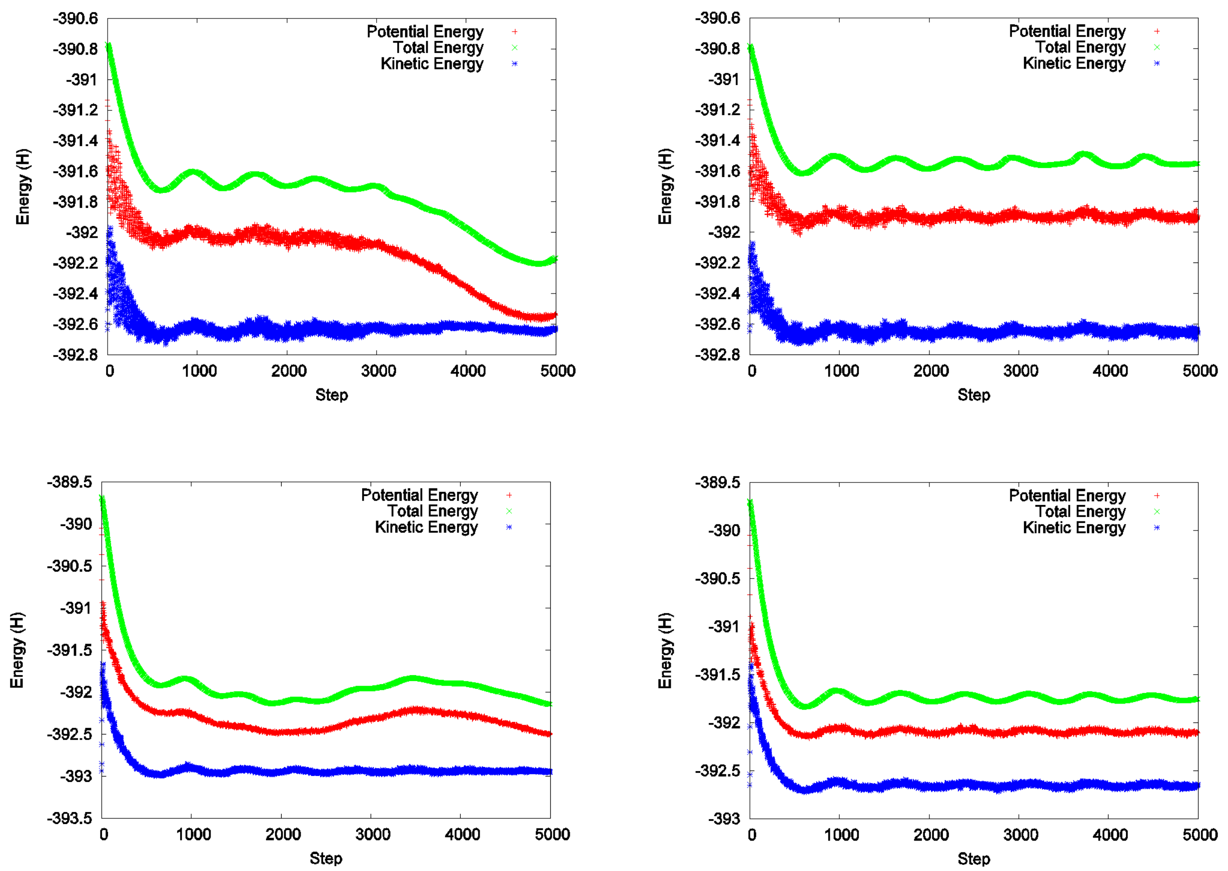

Figure 3A (top left) shows 218 C:40 H planar membrane under a 30 V/nm electric field. **Figure 3B** (top right) shows the same membrane with corners frozen. **Figure 3C** (bottom left) shows a warped membrane under the same conditions as 2A. **Figure 3D** (bottom right) shows a warped membrane under the same conditions as 3B. As in Figure 1, all kinetic energies were shifted down for scaling. The drops in energy in the two unconstrained membranes are due to the membranes twisting the align themselves with the electric field.

To more closely examine what phenomena might be occurring during the simulation, single point calculations were performed for 21 individual steps from the overall MD simulation (every 250th step from 0 to 5000). From these steps, data from a carbon atom on each edge (C10, C55, C109, C164) was taken, including orbital populations and resolved total energy. Samples of these results are given in Figure 4. There were no significant effects caused by the 3 V/nm electric field. Movement caused by the 30 V/nm field is clearly evident in the single point calculations as the left edge moved toward the anode and right edge toward the cathode. This caused a spike in the electron population of the 2p orbitals on the right edge and a drop on the left edge. Conversely the resolved total energy dropped near the cathode and increased near the anode.

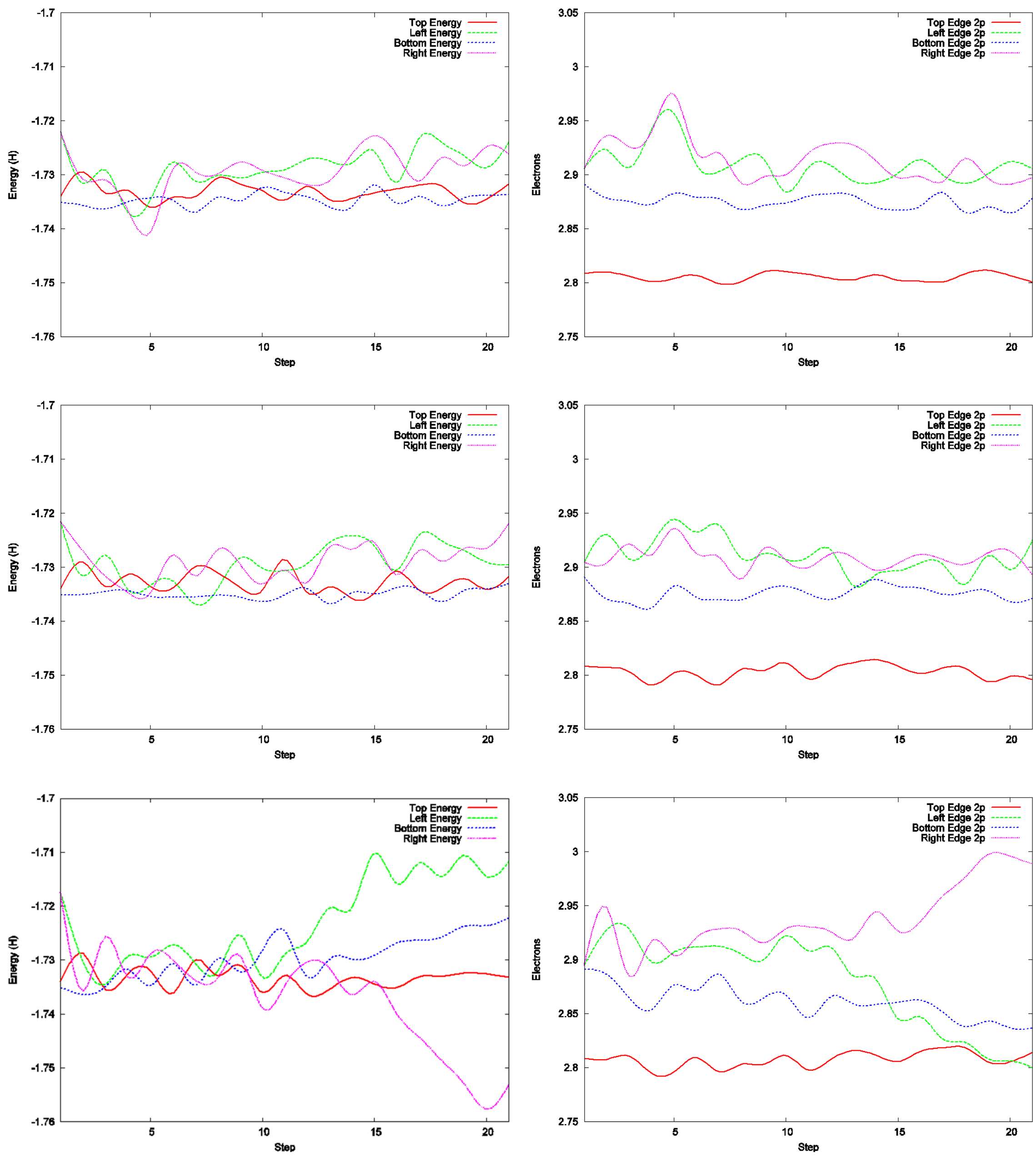

Figure 4A (top left) is the energy of a 218 C:40 H membrane. **Figure 4B** (top right) is the filling of 2p orbitals for the same membrane. The top edge carbon is lower in energy as it is bonded to three other carbons, whereas as the others are bound to two carbons and a single hydrogen. **Figure 4C** (middle left) is the energy of the same membrane under a 3 V/nm electric field. **Figure 4D** (middle right) is the filling of 2p orbitals for the same membrane under a 3 V/nm electric field. **Figure 4E** (bottom left) is the energy of the same membrane under a 30 V/nm electric field. The drastic split between the left and right edges was caused by the membrane aligning itself with the field. **Figure 4F** (bottom right) is the filling of 2p orbitals of the same membrane under a 30 V/nm electric field.

As graphene's mobility is so high, it is unlikely that the membrane rupture is due to an electric field alone. This is supported by the results shown in Figure 4. As even the 30 V/nm field applied in a vacuum did not significantly stress the membrane, it is more likely that the rupture was due to imperfections in the membrane or an ion puncturing it. There are several types of imperfections common to graphene membranes. The first, and most simple, are vacancy-type defects. In these, one or more carbons are absent from the graphene membrane, disturbing the conjugated system and lowering the strength of the membrane. Six variations of a vacancy-type defect were created for these simulations. These can be seen in Figure 5. While initial MD simulations showed no apparent difference between a mono-vacancy and a pristine graphene membrane, the di-vacancy defects allowed for significantly more warping movement in the membrane, comparable to that seen in a pristine membrane subjected to 2000 K temperatures. After initial MD simulations were run to acquire baseline results, a 3 V/nm field was applied to each of the membranes in addition to warped versions of the double vacancy defects output from the original MD simulations. Simulations were run both with and without frozen corners.

So far these simulations with defective sheets have not yielded a rupture. Future simulations will attempt to force an ion, such as fluoride, through the membrane. As the membrane was suspended in an ionic solution when it ruptured, it is possible that the rupture was caused by an ion being shot through the membrane by the electric field. Fluoride will be an ideal candidate for DFTB MD simulations as it is extremely electronegative (meaning it will not lose its electron easily and become neutral) and it is small with fewer non-valance electrons for DFBT to estimate for. This anion will be placed between the membrane and the anode in the hopes that it may puncture the membrane.

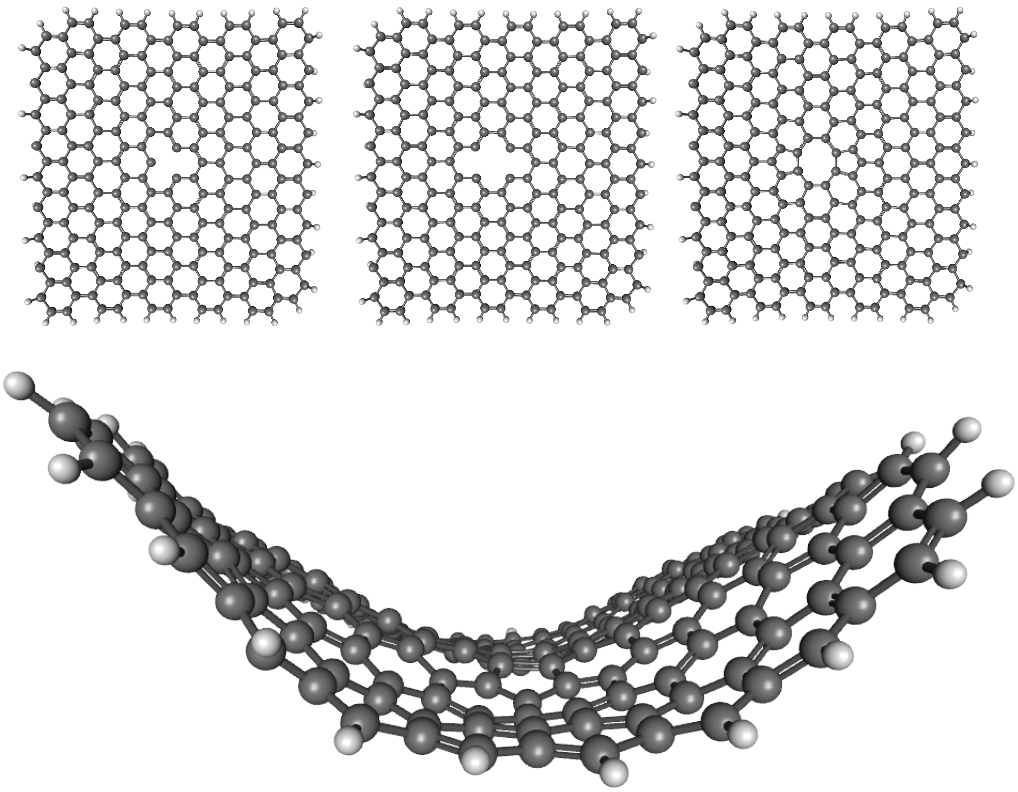

Figure 5A (top left): Mono-vacancy defect
Figure 5B (top middle): Di-vacancy defect
Figure 5C (top right): Reconstructed di-vacancy defect
Figure 5D (bottom): Severe bending during MD simulation due to di-vacancy defect. Shown is step 2300 from the membrane shown in 6B

Improving Computational Speed of DFTB

Even though DFTB is a semi-empirical method, which allows it to be faster than more traditional methods such as Density Functional Theory (DFT), the code can be computationally expensive when evaluating large systems. The largest cost comes from the linear algebra operations, such as matrix-matrix multiplication, Choleskey factorization, and diagonalization. Current DFTB code utilizes LAPACK (Linear Algebra Package) functions to perform these basic operations. LAPACK is inherently a serial code. There are some libraries that allow LAPACK to use multiple threads to perform calculations faster. However, when executing calculations on a supercomputer it is best to use functions that can operate over multiple nodes that are working in parallel. ScaLAPACK was developed for just such a purpose. ScaLAPACK calls are designed utilize a distributed memory system that is then run in parallel. The distributed memory allows a global input matrix to be split into smaller pieces. These smaller pieces are then each sent to their own processor, where the desired linear algebra function is performed. Each processor receiving a portion of the matrix operates in parallel, allowing for faster calculations.

ScaLAPACK functions are able to communicate between various compute nodes by utilizing Basic Linear Algebra Communication Subprograms (BLACS). BLACS is easily initiated with four function calls. A call to `blacs_pinfo`[3] sets up the virtual machine that will be using the process grid to operate in parallel. It determines the number of processes available for use in the process grid as well as labels each process for the user to have a better way of distinguishing the processes. The `blacs_get`[3] call establishes a context label for the process grid that is then used to identify this function throughout the rest of the code. This label is especially important when more than one process grid is being operated within one code. Next, the `blacs_gridinit`[3] function takes every available CPU process and assigns it coordinates in the machines process grid. The user is able to selectively shape the desired process grid size by inputting the desired dimensions of the. In all work for this project only square process grids were used for ease of visualization and computation. Lastly, a call to `blacs_gridinfo`[3] simply returns information about the process grid with the input context label argument. In other words it serves as a double check that all process grid information was correctly established. Once a process grid is finished being used it should be released using `blacs_gridexit`[3] to allow the context label to be recycled if necessary. The `blacs_exit`[3] call releases all memory allocated for the process grid as well as any remaining process grid labels.

```
    ...
!===============INITIALIZE PROCESS GRID====================

    prow =  2 ! number of process rows
    pcol =  2 ! number of process columns
    mb   =  6 ! number of columns in block
    nb   =  6 ! number of rows in block

    call blacs_pinfo   (me,procs)
    call blacs_get     (0, 0, icontxt)
    call blacs_gridinit(icontxt, 'R', prow, pcol)
    call blacs_gridinfo(icontxt, prow, pcol, myrow, mycol)
    ...
```

```
      ...
!=============END BLACS========================
      call blacs_gridexit(icontxt)
      call blacs_exit(0)
      ...
```

Figure 6A (top) shows ample calls for initializing BLACS process grid, while **Figure 6B** (bottom) shows sample calls for terminating a BLACS process grid (bottom)

After the process grid has been created the global matrix must be divided over the process grid. Each CPU process on the grid receives a local array, which is a portion of the global matrix. The data is distributed in a block-cyclic fashion[4] (see Figure 7). The local arrays utilize dynamic memory allocation. This means each process must allocate memory space for the local array, and deallocate the memory after the local arrays are no longer needed. To perform the block cyclic distribution, two subroutines were found to accomplish this task[5] (see Appendix I). One takes the coordinates of an entry in the global matrix and then returns the entry's CPU process grid's coordinate as well as its local array coordinate. The other works in the opposite direction by using the coordinates of a local array entry along with its process grid location to obtain its global array coordinates (see Appendix II).

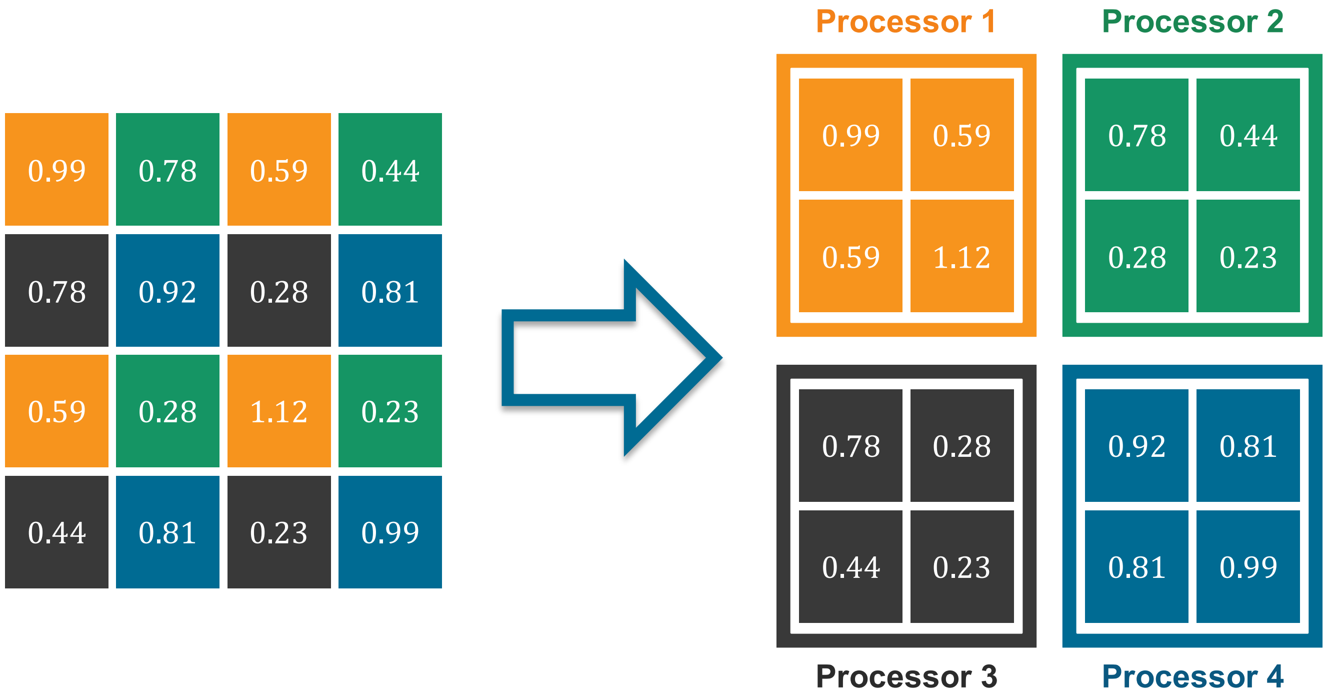

Figure 7 Demonstrating clock cyclic distribution of a 4 x 4 matrix onto a 2 x 2 process grid using a 1 x 1 block size.

Each ScaLAPACK call requires an array descriptor to trace every global memory entry to its process and process array location.

```
   ...
!****PREPARE ARRAY DESCRIPTORS FOR SCALAPACK****
      ides_a(1) = 1          ! descriptor type
      ides_a(2) = icontxt    ! blacs context
      ides_a(3) = n          ! global number of rows
      ides_a(4) = n          ! global number of columns
      ides_a(5) = nb         ! row block size
      ides_a(6) = nb         ! column block size
      ides_a(7) = 0          ! initial process row
      ides_a(8) = 0          ! initial process column
      ides_a(9) = myArows    ! leading dimension of local array
   ...
```

Figure 8 Sample array descriptor

Once the matrix has been distributed and an array descriptor successfully assigned the user is ready to call a ScaLAPACK function. Functions that were of specific interest to this investigation are seen in Table 2.[3]

LAPACK Function	ScaLAPACK Function	Function
DGEMM	PDGEMM	Performs αAB=βC, where α and β are scalars and A,B,C are all N x N matrices
DPOTRF	PDPOTRF	Performs a Cholesky factorization of a real symmetric positive definite N x N matrix utilizing solely its upper or lower triangular matrix
DPOTRI	PDPOTRI	Inverts a real symmetric positive definite N x N matrix by utilizing the output from DPOTRF/PDPOTRF
DSYEV	PDSYEV	Determines the eigenvalues, and if desired, eigenvectors of an N x N real symmetric matrix.
DSYEVD	PDSYEVD	Determines the eigenvalues, and if desired, eigenvectors of an N x N real symmetric matrix utilizing a divide and conquer algorithm
DSYGVD	PDSYEVD	Determines the eigenvalues, and if desired, eigenvectors of the following eigenproblem A*x = λB*x, where A and B are symmetric positive definite N x N matrices.

Table 2: LAPACK functions and their ScaLAPACK equivalents that were used in the benchmarking. See appendices for examples of code.

Common parameters required of a ScaLAPACK function include the name of the local array along with its array descriptor, and the coordinates of its leading entry. Some functions allow for extra calculation options such as utilizing the transpose of an input matrix or solving different arrangements of eigenvector problems. Eigen solver functions also require work matrices to allow for adequate memory space to perform calculations.

Rather than simply replacing LAPACK calls in the DFTB code with ScaLAPACK calls, a little bit of benchmarking was done. All of the ScaLAPACK codes from Table 2 were combined into one code and timed (see the Appendix II for benchmarking code). The following graph demonstrates preliminary speed up seen when using ScaLAPACK functions. As the process grid size of ScaLAPACK is increased, so did the speed of each calculation.

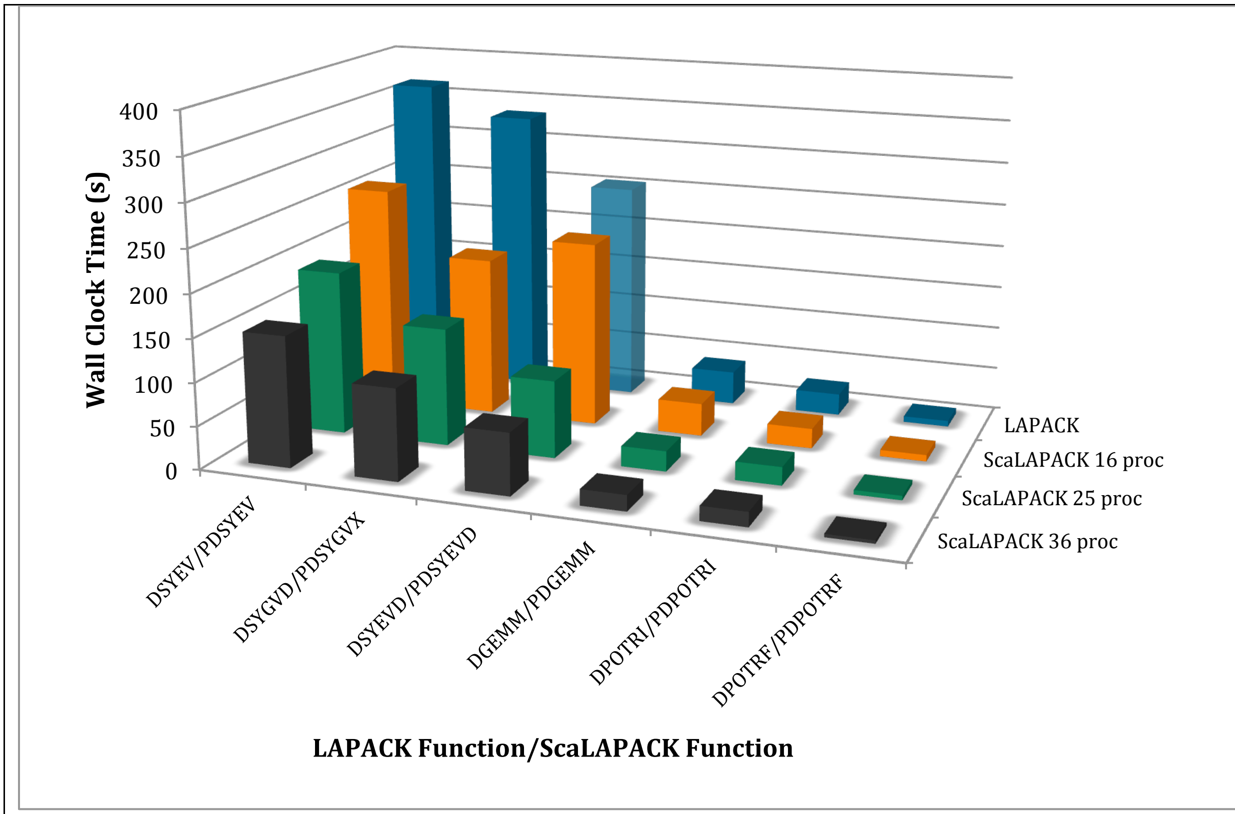

Figure 9 Preliminary benchmarking results comparing LAPACK and ScaLAPACK functions. The transparent bar is the result of calculations run on Darter. All other calculations were run on Beacon using Intel's Math Kernel Library (MKL) along with the Intel compiler. The shown times are the longest processor run time.

All modifications to the DFTB code have taken place mainly in the scf_diis_atrs subroutine in the `scf.90` file. The BLACS process grid initiation and termination have been added to the `prog.f90` file. LAPACK functions are being replaced with ScaLAPACK functions by inserting subroutines that contain the ScaLAPACK function call in place of LAPACK functions (See Appendix III for subroutines). The modified DFTB code is then tested by running a single point MD simulation of an ethene molecule and comparing its results with those of LAPACK DFTB. No benchmarking has been performed on the modified DFTB code as of yet. Currently, only the LAPACK matrix-matrix multiplication function (DGEMM) has been successfully replaced with the ScaLAPACK matrix-matrix multiplication (PDGEMM). Work is also being done to replace the LAPACK eigensolver (DSYEV) with the ScaLAPACK eigensolver (PDSYEV). Unfortunately there is an error in ScaLAPACK's eigenvectors, which is skewing the DFTB calculations. More work is still needed to determine the exact cause of the problem.

The next steps of this project will include continuing to push the limits of ScaLAPACK routines to determine the ideal parameters for process grids. Items to be explored are adjusting the block size to be larger, distributing contiguous blocks of memory, and continuing to increase process grid size. The ultimate goal is to be able to perform 1 MD timestep in under a minute for large systems. As for the DFTB code, once the eigen solver is fixed, the rest of the investigated ScaLAPACK routines will also be added to the code as subroutines. There will also be some investigation on improving the memory efficiency of the DFTB code by having it generate the global matrix data within the local process arrays.

Acknowledgements

We would like to thank the National Science Foundation for funding our research this summer along with University of Tennessee, Knoxville and Oak Ridge National Lab for hosting the CSURE program.

References

[1] DFTB+[Computer software].(2013).Retrieved from http://www.dftb-plus.info

[2] Daniels, C.; Horning, A.; Phillips, A.; Massote, D.; Liang, L.; Bullard, Z.; Sumpter, B.; Meunier, V. Mechanisms Of Stress Release in Graphene Materials.

[3] *NETLIB Repository*. University of Tennessee-Knoxville & Oak Ridge National Lab. Web. 7

[4] LibSci Example https://www.nersc.gov/users/software/programming-libraries/math-libraries/libsci/libsci-example/ (accessed Jun 2015).

[5] Susan , B. Details of Example Program #1 http://netlib.org/scalapack/slug/node28.html (accessed Jul 2015).

Appendix I : Block-Cyclic Distribution Code

```
! convert global index to local index in block-cyclic distribution

      subroutine g2l(i,n,np,nb,p,il)

      implicit none
      integer :: i     ! global array index, input
      integer :: n     ! global array dimension, input
      integer :: np    ! processor array dimension, input
      integer :: nb    ! block size, input
      integer :: p     ! processor array index, output
      integer :: il    ! local array index, output
      integer :: im1

      im1 = i-1
      p   = mod((im1/nb),np)
      il  = (im1/(np*nb))*nb + mod(im1,nb) + 1

      return
      end
! convert local index to global index in block-cyclic distribution

      subroutine l2g(il,p,n,np,nb,i)

      implicit none
      integer :: il    ! local array index, input
      integer :: p     ! processor array index, input
      integer :: n     ! global array dimension, input
      integer :: np    ! processor array dimension, input
      integer :: nb    ! block size, input
      integer :: i     ! global array index, output
      integer :: ilm1

      ilm1 = il-1
      i    = (((ilm1/nb) * np) + p)*nb + mod(ilm1,nb) + 1

     return
      end
```

APPENDIX II : ScaLAPACK Benchmarking Code

```
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! Timing of the ScaLAPACK: PDGEMM,PDPOTRI,PDPOTRF,PDSYEV,PDSYEVD,PDSYGVX
! filename:  time_scalapack.f90
! compile:   mpiifort -o timing.f90 test_scalapack.f90
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! input:     input.txt
!              prow    number of rows in proc grid
!              pcol    number of columns in proc grid
!              n       number of rows/columns in matrix A
!              nb      matrix distribution block size

! oputput:   fort.u, where u=10+processor number, and stdout
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

      use   timing
      implicit none

      integer :: MC, MM, TRF, TRI, EV, EVD, GVX  !if loop variables 1 = run
      integer :: prin     ! matrix print variable
      integer :: n, nb    ! problem size and block size
      integer :: m, nz    ! number of eigen values and vectors
      integer :: myunit   ! local output unit number
      integer :: myArows, myAcols   ! size of local subset of global array
      integer :: i,j, igrid,jgrid, iproc,jproc, myi,myj, pi ! navigating variables
      integer open_status, close_status ! variables for read in files
      integer :: numroc   ! blacs routine
      integer :: me, procs, icontxt, prow, pcol, myrow, mycol  ! blacs data
      integer :: lwork, liwork    !eigen variables
      integer :: info    ! scalapack return value
      integer, dimension(:), allocatable :: ifail,iclustr !PDSYGVX outputs
      integer, dimension(:), allocatable :: iwork ! work array
      integer, dimension(9)   :: ides_a, ides_b, ides_c, ides_z ! scalapack array desc
      real*8, dimension(:), allocatable :: W,WW, work  ! eigen values and work arrays
      real*8, dimension(:), allocatable :: gap ! PDSYGVX output
      real*8, dimension(:,:), allocatable :: A,B,C,D,E,F,Z,ID ! global arrays
      real*8, dimension(:,:), allocatable :: myA,myB,myC,myZ ! local arrays
      real*8 :: vl, vu, il, iu,x,y  !unreferenced range values
      real*8  :: abstol,orfac,PDLAMCH ! PDSYGVX variables

!  Read problem description

!      open(unit=15,file="./ABCp.txt",status="old",iostat=open_status)
!      read(15,*)prow
!      read(15,*)pcol
!      read(15,*)n
!      read(15,*)nb

!====================VARIABLE READ IN=========================
      open(unit=15,file='./input.txt',status='old',iostat=open_status)
      read(15,*),prow  ! number of process rows
      read(15,*),pcol  ! number of process columns
      read(15,*),n     ! leading dimension of global matrix
      read(15,*),nb    ! leading dimension of block size

      read(15,*),prin  ! if prin=1 print all calculations
      read(15,*),MC    ! if 1 print global matrices
      read(15,*),MM    ! if 1 perform PDGEMM on A*B = C
      read(15,*),TRF   ! if 1 perform PDPOTRF Cholesky factorization of A
      read(15,*),TRI   ! if 1 perform PDPOTRI of A (MUST HAVE TRF.eq.1)
```

```
      read(15,*),EV     ! if 1 perform PDSYEV to compute eigenvalues and optionally
eigenvectors
      read(15,*),EVD    ! if 1 perform PDSYEVD to compute eigenvalues and optionally
eigenvectors
      read(15,*),GVX    ! if 1 perform PDSYGVX to compute eigenvalues and optionally
eigenvectors
      lwork = -1        ! must be -1 to give proper dimension for work
      liwork = (7 * N) + (8 * pcol) + 2 !must be -1 to give proper dimension for
liwork
      if (((n/nb) < prow) .or. ((n/nb) < pcol)) then
         print *,"Problem size too small for processor set!"
         stop 100
      endif
!===================GLOBAL MATRIX SET UP==========================
      call time_start(1)

!****ALLOCATING GLOBAL MATRICES****
!MATRIX GUIDE:
!  A  ::  PDGEMM, PDPOTRF, PDPOTRI
!  B  ::  PDGEMM, PDSYEV
!  C  ::  PDSYEVD
!  D  ::  PDSYGVX
!  E  ::  PDSYGVX
      allocate (A(N,N))
      allocate (B(N,N))
      allocate (ID(N,N)) ! will be the identity

!   fill A and B with random numbers
      call random_number(A)
      call random_number(B)

      DO i = 1,N
            DO j = 1,N
            A(j,i)=A(i,j)  ! assure A is symmetric
            B(j,i)=B(i,j)  ! assure B is symmetric
            ID(i,j)=0.0d0
            ID(i,i)=1.0d0
            END DO
      END DO
!   make A & B diagonal dominate to ensure positive definite
      A = A + (ID * N)
      B = B + (ID * N)
!   the order of allocation is an attempt to maximize memory usage
      deallocate(ID)
      allocate (C(N,N))
      allocate (D(N,N))
      allocate (E(N,N))
      C = B
      D = A
      E = B
      call time_stop(1)
!====================INITIALIZE PROCESS GRID==========================
      call blacs_pinfo   (me,procs)
      call blacs_get     (0, 0, icontxt)
      call blacs_gridinit(icontxt, 'R', prow, pcol)
      call blacs_gridinfo(icontxt, prow, pcol, myrow, mycol)
      myunit = 10+me !processor output file label "fort.myunit"
!   process grid info check
      write(myunit,*)"--------"
      write(myunit,*)"Output for processor ",me," to unit ",myunit
      write(myunit,*)"Proc ",me,": myrow, mycol in p-array is ", &
         myrow, mycol
      flush(myunit)
!   determining dimension of local array
```

```
      myArows = numroc(n, nb, myrow, 0, prow)
      myAcols = numroc(n, nb, mycol, 0, pcol)
!   process grid info check
      write(myunit,*)"Size of global array is ",n," x ",n
      write(myunit,*)"Size of block is        ",nb," x ",nb
      write(myunit,*)"Size of local array is  ",myArows," x ",myAcols
      flush(myunit)
!   this prints the info check in the master output file
      if (me.eq.0) then
      write(*,*)"Size of global array is ",n," x ",n
      write(*,*)"Size of block is        ",nb," x ",nb
      write(*,*)"Size of local array is  ",myArows," x ",myAcols
      end if
!=====================GLOBAL MATRIX PRINT CHECK=========================
        if (MC.eq.1) then
         write(myunit,*)"--- matrix check -----"
         write(myunit,*) 'Matrix A'
          do i = 1,N
                write (myunit,9998) (A(i,j), j=1,N)
          end do
          write(myunit,*)
         write(myunit,*)
         write(myunit,*) 'Matrix B'
          do i = 1,N
                write(myunit,9998) (B(i,j), j=1,N)
          end do
         write(myunit,*)
         write(myunit,*) 'Matrix C'
          do i = 1,N
                write(myunit,9998) (C(i,j), j=1,N)
          end do
         write(myunit,*)
         write(myunit,*) 'Matrix D'
          do i = 1,N
                write(myunit,9998) (D(i,j), j=1,N)
          end do
         write(myunit,*)
         write(myunit,*) 'Matrix E'
          do i = 1,N
                write(myunit,9998) (E(i,j), j=1,N)
          end do
         write(myunit,*)
         write(myunit,*) 'Matrix Z'
          do i = 1,N
                write(myunit,9998) (Z(i,j), j=1,N)
          end do
         write(myunit,*)
!         write(myunit,*) 'Matrix ID'
!          do i = 1,N
!                write(myunit,9998) (ID(i,j), j=1,N)
!          end do
!          write(myunit,*)
         write(myunit,*) 'Matrix W'
          do i = 1,N
                write(myunit,9998) W(i)
          end do
        end if

!====================TIMING PRINTING=========================
      write(*,*) 'Time for Matrix Generation (sec)', timetab(1)
!====================START PDGEMM==========================
      if (MM.eq.1) then
      write(myunit,*)"*******PDGEMM*********"
!****INTIALIZE LOCAL ARRAYS****
```

```
       allocate(myA(myArows,myAcols))
       allocate(myB(myArows,myAcols))
       allocate(myC(myArows,myAcols))

       write(myunit,*)"--- before MM -----"
       do i=1,n
          call g2l(i,n,prow,nb,iproc,myi) ! see subroutines
          if (myrow==iproc) then
             do j=1,n
                call g2l(j,n,pcol,nb,jproc,myj)
                if (mycol==jproc) then
                   myA(myi,myj) = A(i,j)
                   myB(myi,myj) = B(i,j)
                   myC(myi,myj) = 0.0d0
!   check matrix filling
                   if (prin.eq.1) then
                   write(myunit,*)"A(",i,",",j,")", &
                                  " --> myA(",myi,",",myj,")=",myA(myi,myj), &
                                  "on proc(",iproc,",",jproc,")"
                   write(myunit,*)"B(",i,",",j,")", &
                                  " --> myB(",myi,",",myj,")=",myB(myi,myj), &
                                  "on proc(",iproc,",",jproc,")"
                   write(myunit,*)"C(",i,",",j,")", &
                                  " --> myC(",myi,",",myj,")=",myC(myi,myj), &
                                  "on proc(",iproc,",",jproc,")"
                   end if
                end if
             end do
          end if
       end do
       flush(myunit)

!****PREPARE ARRAY DESCRIPTORS FOR SCALAPACK****
       ides_a(1) = 1          ! descriptor type
       ides_a(2) = icontxt    ! blacs context
       ides_a(3) = n          ! global number of rows
       ides_a(4) = n          ! global number of columns
       ides_a(5) = nb         ! row block size
       ides_a(6) = nb         ! column block size
       ides_a(7) = 0          ! initial process row
       ides_a(8) = 0          ! initial process column
       ides_a(9) = myArows    ! leading dimension of local array
!  assiging descriptors to all local matrices
       do i=1,9
          ides_b(i) = ides_a(i)
          ides_c(i) = ides_a(i)
       enddo

!****CALL PDGEMM****
       call time_start(2)
       call pdgemm('T','T',n,n,n,1.0d0, myA,1,1,ides_a,  &
                     myB,1,1,ides_b,0.d0, &
                     myC,1,1,ides_c )
       call time_stop(2)

!   Print results
       write(myunit,*)"--- after MM -----"

       do i=1,n
          call g2l(i,n,prow,nb,iproc,myi)
          if (myrow==iproc) then
             do j=1,n
                call g2l(j,n,pcol,nb,jproc,myj)
                if (mycol==jproc) then
```

```
                  if (prin.eq.1) then
                     write(myunit,*)"A(",i,",",j,")", &
                               " --> myA(",myi,",",myj,")=",myA(myi,myj), &
                               "on proc(",iproc,",",jproc,")"
                     write(myunit,*)"B(",i,",",j,")", &
                               " --> myB(",myi,",",myj,")=",myB(myi,myj), &
                               "on proc(",iproc,",",jproc,")"
                     write(myunit,*)"C(",i,",",j,")", &
                               " --> myC(",myi,",",myj,")=",myC(myi,myj), &
                               "on proc(",iproc,",",jproc,")"
                  end if
               end if
            end do
         end if
      end do
      flush(myunit)

!****DEALLOCATING LOCAL MATRICES****
      deallocate(myA, myB, myC)
      end if
!====================TIMING PRINTING==========================
      if (MM.eq.1) then
      write(*,*) 'Time for PDGEMM (sec)', timetab(2)
      end if
!====================END PDGEMM==========================

!====================START PDPOTRF==========================
      if (TRF.eq.1) then
      write(myunit,*)"*******PDPOTRF*********"
!****INITIALIZING LOCAL ARRAYS****
      allocate(myA(myArows,myAcols))

      write(myunit,*)"--- before Cholesky -----"
      do i=1,n
         call g2l(i,n,prow,nb,iproc,myi)    ! see subroutines
         if (myrow==iproc) then
            do j=1,n
               call g2l(j,n,pcol,nb,jproc,myj)
               if (mycol==jproc) then
                  myA(myi,myj) = A(i,j)
!   check matrix filling
                  if (prin.eq.1)then
                  write(myunit,*)"A(",i,",",j,")", &
                            " --> myA(",myi,",",myj,")=",myA(myi,myj), &
                            "on proc(",iproc,",",jproc,")"
                  end if
               end if
            end do
         end if
      end do
      flush(myunit)


!****PREPARE ARRAY DESCRIPTORS FOR SCALAPACK
      ides_a(1) = 1          ! descriptor type
      ides_a(2) = icontxt    ! blacs context
      ides_a(3) = n          ! global number of rows
      ides_a(4) = n          ! global number of columns
      ides_a(5) = nb         ! row block size
      ides_a(6) = nb         ! column block size
      ides_a(7) = 0          ! initial process row
      ides_a(8) = 0          ! initial process column
      ides_a(9) = myArows    ! leading dimension of local array
!   assigning descriptors to all local matrices
```

```
       do i=1,9
          ides_b(i) = ides_a(i)
          ides_c(i) = ides_a(i)
       end do

 !****CALL PDPOTRF****
       call time_start(3)
       call pdpotrf('U',n, myA,1,1,ides_a,info)
       call time_stop(3)

 ! Print results
       if (prin.eq.1) then
       write(myunit,*)"--- after Cholesky -----"
          do i=1,n
             call g2l(i,n,prow,nb,iproc,myi)
             if (myrow==iproc) then
                do j=1,n
                   call g2l(j,n,pcol,nb,jproc,myj)
                   if (mycol==jproc) then
                      write(myunit,*)"A(",i,",",j,")", &
                                     " --> myA(",myi,",",myj,")=",myA(myi,myj), &
                                     "on proc(",iproc,",",jproc,")"
                   end if
                end do
             end if
          end do
       flush(myunit)
       end if

 !****DEALLOCATE LOCAL MATRICES****
       deallocate(myA)

       end if
 !====================TIMING PRINTING=========================
       if (TRF.eq.1) then
       write(*,*) 'Time for PDPOTRF (sec)', timetab(3)
       end if
 !====================END PDPOTRF=========================

 !====================START PDPOTRI=========================
       if (TRI.eq.1) then
       write(myunit,*)"*******PDPOTRI*********"
 !****INITIALIZE LOCAL ARRAYS****
       allocate(myA(myArows,myAcols))
       write(myunit,*)"--- before inversion -----"
       do i=1,n
          call g2l(i,n,prow,nb,iproc,myi)   ! see subroutine
          if (myrow==iproc) then
             do j=1,n
                call g2l(j,n,pcol,nb,jproc,myj)
                if (mycol==jproc) then
                   myA(myi,myj) = A(i,j)
 !   check matrix filling
                   if (prin.eq.1) then
                      write(myunit,*)"A(",i,",",j,")", &
                                     " --> myA(",myi,",",myj,")=",myA(myi,myj), &
                                     "on proc(",iproc,",",jproc,")"
                   end if
                end if
             end do
          end if
       end do
       flush(myunit)
```

```
!****PREPARE ARRAY DESCRIPTORS FOR SCALAPACK****
      ides_a(1) = 1          ! descriptor type
      ides_a(2) = icontxt    ! blacs context
      ides_a(3) = n          ! global number of rows
      ides_a(4) = n          ! global number of columns
      ides_a(5) = nb         ! row block size
      ides_a(6) = nb         ! column block size
      ides_a(7) = 0          ! initial process row
      ides_a(8) = 0          ! initial process column
      ides_a(9) = myArows    ! leading dimension of local array
!   assigning descriptors to all local matrices
      do i=1,9
         ides_b(i) = ides_a(i)
         ides_c(i) = ides_a(i)
      end do

!****CALL PDPOTRI****
     call time_start(4)
     call pdpotri('U',n, myA,1,1,ides_a,info)
     call time_stop(4)

! Print results
      if (prin.eq.1)then
         write(myunit,*)"--- after inversion -----"
         do i=1,n
            call g2l(i,n,prow,nb,iproc,myi)
            if (myrow==iproc) then
               do j=1,n
                  call g2l(j,n,pcol,nb,jproc,myj)
                  if (mycol==jproc) then
                     write(myunit,*)"A(",i,",",j,")", &
                                    " --> myA(",myi,",",myj,")=",myA(myi,myj), &
                                    "on proc(",iproc,",",jproc,")"
                  end if
               end do
            end if
      end do
      flush(myunit)
      end if

!****DEALLOCATING LOCAL MATRICES****
      deallocate(myA)
      end if
!====================TIMING PRINTING=========================
      if (TRI.eq.1) then
      write(*,*) 'Time for PDPOTRI (sec)', timetab(4)
      end if
!====================END PDPOTRI=========================
!****DEALLOCATION TO SAVE MEMORY****
      deallocate(A)
!****INITIALIZING MORE GLOBAL ARRAYS****
      allocate (Z(N,N))
      allocate (W(N))
      DO i = 1,N
            W(i) = 0.0d0
            DO j = 1,N
            Z(i,j)=0.0d0
            END DO
      END DO
!====================START PDSYEV=========================
      if (EV.eq.1) then
      write(myunit,*)"*******PDSYEV*********"
!****INITIALIZE LOCAL ARRAYS****
      allocate(myB(myArows,myAcols))
```

```
      allocate(myZ(myArows,myAcols))
      allocate(work(1))
      write(myunit,*)"--- before operation -----"
      do i=1,n
         call g2l(i,n,prow,nb,iproc,myi)
         if (myrow==iproc) then
            do j=1,n
               call g2l(j,n,pcol,nb,jproc,myj)
               if (mycol==jproc) then
                  myB(myi,myj) = B(i,j)
                  myZ(myi,myj) = Z(i,j)
                  if (prin.eq.1) then
                     write(myunit,*)"B(",i,",",j,")", &
                                    " --> myB(",myi,",",myj,")=",myB(myi,myj), &
                                   "on proc(",iproc,",",jproc,")"
                     write(myunit,*)"Z(",i,",",j,")", &
                                    " --> myA(",myi,",",myj,")=",myZ(myi,myj), &
                                    "on proc(",iproc,",",jproc,")"
                  end if
               end if
            end do
         end if
      end do
      flush(myunit)

!****PREPARE ARRAY DESCRIPTORS FOR SCALAPACK****
      ides_a(1) = 1          ! descriptor type
      ides_a(2) = icontxt    ! blacs context
      ides_a(3) = n          ! global number of rows
      ides_a(4) = n          ! global number of columns
      ides_a(5) = nb         ! row block size
      ides_a(6) = nb         ! column block size
      ides_a(7) = 0          ! initial process row
      ides_a(8) = 0          ! initial process column
      ides_a(9) = myArows    ! leading dimension of local array
!   Assigning descriptors to all local matrices
      do i=1,9
         ides_b(i) = ides_a(i)
         ides_z(i) = ides_a(i)
      end do
      write(myunit,*) 'descriptor arrays assigned'
      write(myunit,*)'Made it to PDSYEV'
      flush(myunit)
!****CALL PDSYEV****
      call time_start(5)
!   first call is to obtain dimension for work array
      call pdsyev('V','U',n,myB,1,1,ides_b,w,myZ,1,1,ides_z,work,lwork,info)
      lwork = work(1)    ! assinging lwork to proper value
      deallocate(work)   ! resizing work to perform calculation
      allocate(work(lwork))
      flush(myunit)
!   second call performs actual calculation
      call pdsyev('V','U',n,myB,1,1,ides_b,w,myZ,1,1,ides_z,work,lwork,info)
      call time_stop(5)
      write(myunit,*)'Completed PDSYEV'

!   print resutls
      if (prin.eq.1) then
         write(myunit,*)"--- after operation -----"
         do i=1,n
            call g2l(i,n,prow,nb,iproc,myi)
            if (myrow==iproc) then
               do j=1,n
                  call g2l(j,n,pcol,nb,jproc,myj)
```

```
                  if (mycol==jproc) then
                     write(myunit,*)"B(",i,",",j,")", &
                                    " --> myB(",myi,",",myj,")=",myB(myi,myj), &
                                    "on proc(",iproc,",",jproc,")"
                     write(myunit,*)"Z(",i,",",j,")", &
                                    " --> myA(",myi,",",myj,")=",myZ(myi,myj), &
                                    "on proc(",iproc,",",jproc,")"
                  end if
               end do
            end if
         end do
         write(myunit,*)"--- eigen values -----"
         write(myunit, 9998) w
         flush(myunit)
      end if
!****DEALLOCATING ARRAYS****
      deallocate(myB, myZ)
      deallocate(work)
      end if
!====================TIMING PRINTING=========================
      if (EV.eq.1) then
      write(*,*) 'Time for PDSYEV (sec)', timetab(5)
      end if
!====================END PDSYEV=========================
!****DEALLOCATION TO SAVE MEMORY****
      deallocate(B)
      lwork = -1  ! reassin to perform PDSYEVD
! Reset W and Z
      do i = 1,n
         W(i) = 0.0d0
         do j = 1,n
            Z(i,j) = 0.0d0
         end do
      end do
!====================START PDSYEVD=========================
      if (EVD.eq.1) then
      write(myunit,*)"*******PDSYEVD*********"
!****INITIALIZING LOCAL ARRAYS****
      allocate(myC(myArows,myAcols))
      allocate(myZ(myArows,myAcols))
      allocate(work(1))
      allocate(iwork(1))
      write(myunit,*)"--- before operation -----"
      do i=1,n
         call g2l(i,n,prow,nb,iproc,myi)
         if (myrow==iproc) then
            do j=1,n
               call g2l(j,n,pcol,nb,jproc,myj)
               if (mycol==jproc) then
                  myC(myi,myj) = C(i,j)
                  myZ(myi,myj) = Z(i,j)
                  if (prin.eq.1) then
!   check matrix filling
                     write(myunit,*)"B(",i,",",j,")", &
                                    " --> myB(",myi,",",myj,")=",myC(myi,myj), &
                                    "on proc(",iproc,",",jproc,")"
                     write(myunit,*)"Z(",i,",",j,")", &
                                    " --> myA(",myi,",",myj,")=",myZ(myi,myj), &
                                    "on proc(",iproc,",",jproc,")"
                  end if
               end if
            end do
         end if
      end do
```

```
      flush(myunit)

!****PREPARE ARRAY DESCRIPTORS FOR SCALAPACK****
      ides_a(1) = 1          ! descriptor type
      ides_a(2) = icontxt    ! blacs context
      ides_a(3) = n          ! global number of rows
      ides_a(4) = n          ! global number of columns
      ides_a(5) = nb         ! row block size
      ides_a(6) = nb         ! column block size
      ides_a(7) = 0          ! initial process row
      ides_a(8) = 0          ! initial process column
      ides_a(9) = myArows    ! leading dimension of local array
!   assigning descriptors to all local matrices
      do i=1,9
         ides_c(i) = ides_a(i)
         ides_z(i) = ides_a(i)
      enddo
      write(myunit,*) 'descriptor arrays assigned'

!****CALL PDSYEVD****
      write(myunit,*)'Made it to PDSYEVD'
      flush(myunit)
      call time_start(6)
!   first call is to obtain dimension for work and iwork
      call pdsyevd('V','U',n,myC,1,1,ides_c,w,myZ,1,1,ides_z,&
           work,lwork,iwork,liwork,info)
      lwork = work(1)           ! assinging lwork to proper value
      deallocate(work,iwork)    ! resizing work and iwork to perform calculation
      allocate(work(lwork))
      allocate(iwork(liwork))
      flush(myunit)
!   second call performs actual calculation
      call pdsyevd('V','U',n,myC,1,1,ides_c,w,myZ,1,1,ides_z, &
           work,lwork,iwork,liwork,info)
      call time_stop(6)
      write(myunit,*)'Completed PDSYEVD'
!   print results
      if (prin.eq.1) then
         do i=1,n
            call g2l(i,n,prow,nb,iproc,myi)
            if (myrow==iproc) then
               do j=1,n
                  call g2l(j,n,pcol,nb,jproc,myj)
                  if (mycol==jproc) then
                     write(myunit,*)"B(",i,",",j,")", &
                                    " --> myB(",myi,",",myj,")=",myC(myi,myj), &
                                    "on proc(",iproc,",",jproc,")"
                     write(myunit,*)"Z(",i,",",j,")", &
                                    " --> myA(",myi,",",myj,")=",myZ(myi,myj), &
                                    "on proc(",iproc,",",jproc,")"
                  end if
               end do
            end if
         end do
         flush(myunit)
         write(myunit,*)"--- eigen values -----"
         write(myunit, 9998) w
       end if
!****DEALLOCATING MATRICES****
      deallocate(myC,myZ)
      deallocate(work,iwork)
      end if
!====================TIMING PRINTING==========================
      if (EVD.eq.1) then
```

```
      write(*,*) 'Time for PDSYEVD (sec)', timetab(6)
      end if
!====================END PDSYEVD==========================
!****DEALLOCATION TO SAVE MEMORY****
      deallocate(C)
!****INIRIALIZINF MORE GLOBAL ARRAYS****
      allocate (ifail(N))
      allocate (iclustr(2*(prow*pcol)))
      allocate (gap(prow*pcol))
      lwork = -1  ! reassign to perform PDSYGVX
      liwork = -1 ! reassign to perform PDSYGVX
! Reset W and Z
      do i = 1,n
         W(i) = 0.0d0
         do j = 1,n
            Z(i,j) = 0.0d0
         end do
      end do
!====================START PDSYGVX==========================
      if (GVX.eq.1) then
      write(myunit,*)"*******PDSYGVX*********"
!****ALLOCATING LOCAL ARRAYS****
      allocate(myA(myArows,myAcols))
      allocate(myB(myArows,myAcols))
      allocate(myZ(myArows,myAcols))
      allocate(work(1))
      allocate(iwork(1))

!****DISTRIBUTING GLOBAL MATRIX****=
      write(myunit,*)"--- before operation -----"
         do i=1,n
         call g2l(i,n,prow,nb,iproc,myi)
         if (myrow==iproc) then
            do j=1,n
               call g2l(j,n,pcol,nb,jproc,myj)
               if (mycol==jproc) then
                  myA(myi,myj) = D(i,j)
                  myB(myi,myj) = E(i,j)
                  myZ(myi,myj) = Z(i,j)
!   matrix check
                    if (prin.eq.1) then
                       write(myunit,*)"A(",i,",",j,")", &
                                      " --> myA(",myi,",",myj,")=",myA(myi,myj), &
                                      "on proc(",iproc,",",jproc,")"
                       write(myunit,*)"B(",i,",",j,")", &
                                      " --> myA(",myi,",",myj,")=",myB(myi,myj), &
                                      "on proc(",iproc,",",jproc,")"
                       write(myunit,*)"Z(",i,",",j,")", &
                                      " --> myA(",myi,",",myj,")=",myZ(myi,myj), &
                                      "on proc(",iproc,",",jproc,")"
                    end if
               endif
            enddo
         endif
      enddo

!     Assinging the appropriate value accodring to documentation
      abstol = PDLAMCH(icontxt,'U')
!****PREPARE ARRAY DESCRIPTORS FOR SCLAPACK****
      ides_a(1) = 1          ! descriptor type
      ides_a(2) = icontxt    ! blacs context
      ides_a(3) = n          ! global number of rows
      ides_a(4) = n          ! global number of columns
      ides_a(5) = nb         ! row block size
```

```
       ides_a(6) = nb         ! column block size
       ides_a(7) = 0          ! initial process row
       ides_a(8) = 0          ! initial process column
       ides_a(9) = myArows    ! leading dimension of local array
!    assinging descriptors to all local matrices
       do i=1,9
          ides_b(i) = ides_a(i)
          ides_z(i) = ides_a(i)
       enddo

!       write(myunit,*) 'descriptor arrays assigned'
       write(myunit,*)'Made it to PDSYGVX'
       flush(myunit)

!****CALL PDSYGVX****
       call time_start(7)
!    first call to get proper work array dimensions
       call PDSYGVX(1,'V','A','L',N,myA,1,1,&
            ides_a,myB,1,1,ides_b,vl,vu,il,iu,&
            abstol,m,nz,w,orfac,myZ,1,1,ides_z,&
            work,lwork,iwork,liwork,ifail,iclustr,&
            gap,info)
!    reassign proper dimensions for work arrays
       lwork = work(1)
       liwork = iwork(1)
       deallocate(work,iwork)
       allocate(work(lwork))
       allocate(iwork(liwork))
!    second call performs actual calculation
       call PDSYGVX(1,'V','A','L',n,myA,1,1,&
            ides_a,myB,1,1,ides_b,vl,vu,il,iu,&
            abstol,m,nz,w,orfac,myZ,1,1,ides_z,&
            work,lwork,iwork,liwork,ifail,iclustr,&
            gap,info)
       call time_stop(7)
       write(myunit,*)'Completed PDSYGVX'

!      Print Resutls
       if (prin.eq.1) then
          write(myunit,*)"---- after operation -----"
          do i=1,n
             call g2l(i,n,prow,nb,iproc,myi)
             if (myrow==iproc) then
                do j=1,i
                   call g2l(j,n,pcol,nb,jproc,myj)
                   if (mycol==jproc) then
                      write(myunit,*)"A(",i,",",j,")", &
                                     " --> myA(",myi,",",myj,")=",myA(myi,myj), &
                                     "on proc(",iproc,",",jproc,")"
                      write(myunit,*)"B(",i,",",j,")", &
                                     " --> myA(",myi,",",myj,")=",myB(myi,myj), &
                                     "on proc(",iproc,",",jproc,")"
                      write(myunit,*)"Z(",i,",",j,")", &
                                     " --> myA(",myi,",",myj,")=",myZ(myi,myj), &
                                     "on proc(",iproc,",",jproc,")"
                   end if
                end do
             end if
          end do
          flush(myunit)
          write(myunit,*)"Number of eign values found:", m
          write(myunit,*)"--- eigen values -----"
          write(myunit, 9998) w
          write(myunit,*)"# eigen vectors computed:", nz
```

```
      end if
!****DEALLOCATING ARRAYS****
      deallocate(myA,myB,myZ)
      deallocate(work,iwork)
      deallocate(ifail,iclustr,gap)
      end if
!====================TIMING PRINTING==========================
      if (GVX.eq.1) then
      write(*,*) 'Time for PDSYGVX (sec)', timetab(7)
      end if
!====================END PDSYGVX==========================

!=============DEALLOCATE REMAINING MATRICES==========================
      deallocate(D,E,Z,W)
!==================================END BLACS==================================
      call blacs_gridexit(icontxt)
      call blacs_exit(0)

      close(15,iostat=close_status)   ! end read in
9998    FORMAT( 11(:,1X,F8.5) )
      end

!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
!++++++++++++++++++++++++SUBROUTINES++++++++++++++++++++++++++++
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

! convert global index to local index in block-cyclic distribution

      subroutine g2l(i,n,np,nb,p,il)

      implicit none
      integer :: i    ! global array index, input
      integer :: n    ! global array dimension, input
      integer :: np   ! processor array dimension, input
      integer :: nb   ! block size, input
      integer :: p    ! processor array index, output
      integer :: il   ! local array index, output
      integer :: im1

      im1 = i-1
      p   = mod((im1/nb),np)
      il  = (im1/(np*nb))*nb + mod(im1,nb) + 1

      return
      end
! convert local index to global index in block-cyclic distribution

      subroutine l2g(il,p,n,np,nb,i)

      implicit none
      integer :: il   ! local array index, input
      integer :: p    ! processor array index, input
      integer :: n    ! global array dimension, input
      integer :: np   ! processor array dimension, input
      integer :: nb   ! block size, input
      integer :: i    ! global array index, output
      integer :: ilm1

      ilm1 = il-1
      i    = (((ilm1/nb) * np) + p)*nb + mod(ilm1,nb) + 1

     return
      end
```

Appendix III : MYPDGEMM and MYPDSYEV DFTB Subroutines

```
! PDGEMM Subroutine for global matrices AB=C
      subroutine MYPDGEMM(n,nb,mb,icontxt,prow,pcol,myrow,mycol, A, B, C)

      implicit none
      integer :: n    ! leading dimension of global matrices--INPUT
      real*8, dimension(n,n) :: A,B  ! global matrices to be multiplied--INPUT
      real*8, dimension(n,n) :: C   ! global product matrix--OUTPUT
      integer :: icontxt, prow,pcol,myrow, mycol  ! blacs data--INPUT
      integer :: nb, mb    ! problem size and block size
      integer :: myunit       ! local output unit number
      integer :: myArows, myAcols   ! size of local subset of global array
      integer :: i,j, igrid,jgrid, iproc,jproc, myi,myj, p   !navigating variables
      integer :: numroc   ! blacs routine
      integer :: info    ! scalapack return value
!      integer :: open_status, close_status
      integer, dimension(9)   :: ides_a, ides_b, ides_c ! scalapack array desc
      real*8, dimension(:,:), allocatable :: myA,myB,myC    ! local matrices

!    prow =  2 ! number of process rows
!    pcol =  2 ! number of process columns
!    mb   =  6 ! number of columns in block
!    nb   =  6 ! number of rows in block
!======================INITIALIZING GLOBAL MATRICES=======================
!      allocate (A(N,N))
!     allocate (B(N,N))
!    allocate (C(N,N))

!===================INITIALIZE PROCESS GRID=========================
!     write(*,*)'... entering mypdgemm' ; call flush(6)
!      call blacs_pinfo   (me,procs)
!      write(*,*)'  ok -1, me:',me;  call flush(6)
!      call blacs_get     (0, 0, icontxt)
!      write(*,*)'  ok -2, me:',me;  call flush(6)
!      write(*,*)'icontxt:',icontxt,'me',me; call flush(6)
!      call blacs_gridinit(icontxt, 'R', prow, pcol)
!     write(*,*)'  ok -3, me:',me;  call flush(6)
!      call blacs_gridinfo(icontxt, prow, pcol, myrow, mycol)
!      write(*,*)'  ok -4, me:',me;  call flush(6)

!      myunit = 10+me
!     process grid info check
!      write(myunit,*)"--------"
!      write(myunit,*)"Output for processor ",me," to unit ",myunit
!      write(myunit,*)"Proc ",me,": myrow, mycol in p-array is ", &
!         myrow, mycol
!      flush(myunit)

! global structure:  matrix A of n rows and n columns
!                    matrix B of n rows and n column
!                    matrix C of n rows and n column

!     determining size of local array
      myArows = numroc(n, nb, myrow, 0, prow)
      myAcols = numroc(n, nb, mycol, 0, pcol)
!     process grid info check
!      write(myunit,*)"Size of global array is ",n," x ",n
!      write(myunit,*)"Size of block is        ",nb," x ",nb
!      write(myunit,*)"Size of local array is  ",myArows," x ",myAcols
!      flush(myunit)
```

```
!====================GLOBAL MATRIX CHECK==========================
!      if (myunit.eq.10) then    ! Initialize to have only one process print
!          write(myunit,*)"--- matrix check -----"
!          write(myunit,*) 'Matrix', A
!          do i = 1,n
!             write(myunit,9998) (A(i,j), j=1,n)
!          end do
!          write(myunit,*)
!          write(myunit,*) 'Matrix', B
!          do i = 1,n
!             write(myunit,9998) (B(i,j), j=1,n)
!          end do
!          write(myunit,*)
!      end if

!====================INITIALIZING LOCAL ARRAYS===========================
!       write(myunit,*)"--- matrix check all    all  -----"
      allocate(myA(myArows,myAcols))
!       write(*,*) "mya: ", allocated(myA)
      allocate(myB(myArows,myAcols))
!       write(*,*)
      allocate(myC(myArows,myAcols))

 !      write(myunit,*)"--- before MM -----"
      do i=1,n
         call g2l(i,n,prow,nb,iproc,myi) ! see subroutines
         if (myrow==iproc) then
            do j=1,n
               call g2l(j,n,pcol,nb,jproc,myj)
               if (mycol==jproc) then
                  myA(myi,myj) = A(i,j)
                  myB(myi,myj) = B(i,j)
!                  myC(myi,myj) = C(i,j)
!                 check matrix filling
!                  write(myunit,*)"A(",i,",",j,")", &
!                               " --> myA(",myi,",",myj,")=",myA(myi,myj), &
!                               "on proc(",iproc,",",jproc,")"
!                  write(myunit,*)"B(",i,",",j,")", &
!                               " --> myB(",myi,",",myj,")=",myB(myi,myj), &
!                               "on proc(",iproc,",",jproc,")"
!                  write(myunit,*)"C(",i,",",j,")", &
!                               " --> myC(",myi,",",myj,")=",myC(myi,myj), &
!                               "on proc(",iproc,",",jproc,")"
               end if
            end do
         end if
      end do
!       flush(myunit)


!=============PREPARE ARRAY DESCRIPTORS FOR SCLAPACK==========================
      ides_a(1) = 1          ! descriptor type
      ides_a(2) = icontxt    ! blacs context
      ides_a(3) = n          ! global number of rows
      ides_a(4) = n          ! global number of columns
      ides_a(5) = nb         ! row block size
      ides_a(6) = nb         ! column block size
      ides_a(7) = 0          ! initial process row
      ides_a(8) = 0          ! initial process column
      ides_a(9) = myArows    ! leading dimension of local array
!     assigning descriptors to all local matrices
      do i=1,9
         ides_b(i) = ides_a(i)
         ides_c(i) = ides_a(i)
```

```
      end do

!=============SCALAPACK ROUTINE==========================
      call pdgemm('N','N',n,n,n,1.0d0, myA,1,1,ides_a,  &
                    myB,1,1,ides_b,0.d0, &
                    myC,1,1,ides_c )

! Print results
!       write(myunit,*)"--- after MM -----"
      do i=1,n
         call g2l(i,n,prow,nb,iproc,myi)
         if (myrow==iproc) then
            do j=1,n
               call g2l(j,n,pcol,nb,jproc,myj)
               if (mycol==jproc) then
                  C(i,j) = myC(myi,myj)
!                  write(myunit,*)"A(",i,",",j,")", &
!                                 " --> myA(",myi,",",myj,")=",myA(myi,myj), &
!                                 "on proc(",iproc,",",jproc,")"
!                  write(myunit,*)"B(",i,",",j,")", &
!                                 " --> myB(",myi,",",myj,")=",myB(myi,myj), &
!                                 "on proc(",iproc,",",jproc,")"
!                  write(myunit,*)"C(",i,",",j,")", &
!                                 " --> myC(",myi,",",myj,")=",myC(myi,myj), &
!                                 "on proc(",iproc,",",jproc,")"
               end if
            end do
         end if
      end do
!       flush(myunit)

!       write(myunit,*)
!       write(myunit,*) 'Matrix', C
!       do i = 1,n
!          write(myunit,9998) (C(i,j), j=1,n)
!       end do
!=============DEALLOCATE ALL MATRICES==========================
      deallocate(myA, myB, myC)
!       deallocate(A, B, C)
!      close(15,iostat=close_status) !close read in
!=============END BLACS==========================
!        call blacs_gridexit(icontxt)
!        call blacs_exit(0)
9998    FORMAT( 11(:,1X,F8.5) )
      end subroutine
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

      subroutine MYPDSYEV(n,nb,mb,icontxt,prow,pcol,myrow,mycol,A,W)

      implicit none
      integer :: n        ! leading dimension of global matrix--INPUT
      real*8, dimension(n,n) :: A ! global matrix to be solved--INPUT
      real*8, dimension(n)   :: W ! eigenvalues--OUTPUT
!       real*8, dimension(n,n) :: Z ! eigenvectors--OUTPUT
      integer :: nb,mb       ! problem size and block size
      integer :: myunit   ! local output unit number
      integer :: myArows, myAcols   ! size of local subset of global array
      integer :: i,j, igrid,jgrid, iproc,jproc, myi,myj, p !navigating variables
      integer :: numroc   ! blacs routine
      integer :: me, procs, icontxt, prow, pcol, myrow, mycol  ! blacs data
      integer :: info    ! scalapack return value
      integer :: lwork
!       integer open_status, close_status  ! read in variables
      integer, dimension(9)   :: ides_a, ides_z ! scalapack array desc
```

```
      real*8, dimension(:), allocatable   :: work ! work array
      real*8, dimension(:,:), allocatable :: myA, myZ ! local matrices
!      write(*,*) '***********IN MYPDSYEV**************'
!      prow  = 1   ! number of process rows
!      pcol  = 1   ! number of process columns
!      nb    = 1   ! leading dimension of block size
!      lwork = -1  ! returns idealized workspace
      lwork = -1   ! allows first PDSYEV call to return proper work dimension

!     determining size of local array
      myArows = numroc(n, nb, myrow, 0, prow)
      myAcols = numroc(n, nb, mycol, 0, pcol)
!====================INITIALIZING LOCAL ARRAYS=========================
      allocate(myA(myArows,myAcols))
      allocate(myZ(myArows,myAcols))
      allocate(work(1))
!      write(myunit,*)"--- before operation -----"
      do i=1,n
         call g2l(i,n,prow,nb,iproc,myi) ! see subroutines
         if (myrow==iproc) then
            do j=1,n
               call g2l(j,n,pcol,nb,jproc,myj)
               if (mycol==jproc) then
                  myA(myi,myj) = A(i,j)
                  myZ(myi,myj) = 0.0d0
!                 check matrix filling
!                  write(myunit,*)"A(",i,",",j,")", &
!                                 " --> myA(",myi,",",myj,")=",myA(myi,myj), &
!                                 "on proc(",iproc,",",jproc,")"
!                  write(myunit,*)"Z(",i,",",j,")", &
!                                 " --> myA(",myi,",",myj,")=",myZ(myi,myj), &
!                                 "on proc(",iproc,",",jproc,")"
               endif
            enddo
         endif
      enddo
      flush(myunit)
!      write(*,*) 'MATRICES DISTRIBUTED'
!=============PREPARE ARRAY DESCRIPTORS FOR SCLAPACK=========================
      ides_a(1) = 1         ! descriptor type
      ides_a(2) = icontxt   ! blacs context
      ides_a(3) = n         ! global number of rows
      ides_a(4) = n         ! global number of columns
      ides_a(5) = nb        ! row block size
      ides_a(6) = nb        ! column block size
      ides_a(7) = 0         ! initial process row
      ides_a(8) = 0         ! initial process column
      ides_a(9) = myArows   ! leading dimension of local array
!     assigning descriptors to all local matrices
      do i=1,9
         ides_z(i) = ides_a(i)
      enddo
!      write(myunit,*) 'descriptor arrays assigned'

!=============SCALAPACK ROUTINE=========================
!      write(myunit,*)'Made it to PDSYEV'
!      flush(myunit)
!     First call is to obtain dimension for work
!      write(*,*) 'First Call'
      call pdsyev('V','U',n,mya,1,1,ides_a,w,myZ,1,1,ides_z,work,lwork,info)
!      write(myunit,*) 'work is'
!      write(myunit,9998) work
!      flush(myunit)
      lwork = work(1)
```

```
!      write(*,*) 'lwork: ', lwork
      deallocate(work)
      allocate(work(lwork))
!      flush(myunit)
!     performing actual calculation
!      write(*,*) 'work reallocated'
      call pdsyev('V','U',n,mya,1,1,ides_a,w,myZ,1,1,ides_z,work,lwork,info)
!      write(*,*) 'made it through second call'
!      write(myunit,*)'Completed PDSYEV'
!     print results
!!      do iproc=1,prow
!!      if (myrow==iproc) then
!!         do jproc=1,pcol
!!         if (mycol==jproc) then
!!      do myi=1,myArows
!        call l2g(myi,iproc,n,prow,nb,i)
      do i=1,n
         call g2l(i,n,prow,nb,iproc,myi)
         if (myrow==iproc) then
!!           do myj=1,myAcols
!!              call l2g(myj,jproc,n,pcol,nb,j)
               do j=1,n
               call g2l(j,n,pcol,nb,jproc,myj)
               if (mycol==jproc) then
                  A(i,j) = myZ(myi,myj)
!                  write(myunit,*)"A(",i,",",j,")", &
!                                 " --> myA(",myi,",",myj,")=",myA(myi,myj), &
!                                 "on proc(",iproc,",",jproc,")"
                  write(*,*)"Z(",i,",",j,")", &
                                " --> myA(",myi,",",myj,")=",myZ(myi,myj), &
                              "on proc(",iproc,",",jproc,")"
               endif
            enddo
         endif
      enddo
!!      end if
!!      end do
!!      end if
!!      end do
      flush(myunit)


!      write(myunit,*)"--- eigen values -----"
!      write(myunit, 9998) w

!=============DEALLOCATE ALL MATRICES=========================
      deallocate(myA, myZ)
!      deallocate(A,W,Z)
!     close(15,iostat=close_status) ! close read in
!=============END BLACS=========================
!      call blacs_gridexit(icontxt)
!      call blacs_exit(0)
9998    FORMAT( 11(:,1X,F8.5) )

      end subroutine
```