https://www.dftbplus.org/

MPI/OMP patterns in DFTB+

Ben Hourahine*
University of Strathclyde

* and the DFTB+
developers group

MPI/OMP in DFTB+

- DFTB+ can run in either parallelism mode, but has a few places that benefit from both together
- MPI uses the leader/follower model and splits the MPI_COMM_WORLD in various ways
- OMP usually used for do loops, typically parallelized “one deep”
- Common code base, compile time decision for enabling MPI build – cmake and preprocessing of source https://github.com/dftbplus/mpifx

DFTB+ data structures with MPI

- Overlap and hamiltonian relatively sparse, so store in block compressed matrices with a copy on each communicator member (with indexing neighbour arrays)
- Dense matrices as BLACS format, typically with nearly square grid
- Grids for atoms and split COMM_WORLD and grids for separate k-points/spin groups (similar split for NEGF transport)

DFTB+ MPI choice – funneled

MPI_THREAD_FUNNELED represents a thread support level.
It is used as part of the MPI_Init_thread initialisation.
MPI_THREAD_FUNNELED is the second level; it informs MPI that the application is multithreaded, however all MPI calls will be issued from the master thread only. Other thread support levels are, in order, MPI_THREAD_SINGLE, MPI_THREAD_SERIALIZED and MPI_THREAD_MULTIPLE.

https://rookiehpc.com/mpi/docs/mpi_thread_funneled.php

Wrapped functionalities

Libraries wrapping functionality for

- MPI
  - https://github.com/dftbplus/mpifx
- ScaLAPACK
  - https://github.com/dftbplus/mpifx
- + others

Use Fypp for the preprocessing (https://github.com/aradi/fypp)

```
!> Initializes a threaded MPI environment.
!!
!! \param requiredThreading  Threading support required (MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED,
!!     MPI_THREAD_SERIALIZED, MPI_THREAD_MULTIPLE)
!! \param proviedeThreading  Threading level provided by the MPI-framework. If not present and
!!     the framework offers a lower support than required, the routine stops program execution.
!! \param error Error code on return. If not present and error code would have been non-zero,
!!     routine aborts program execution.
!!
!! \see MPI documentation (\c MPI_INIT)
!!
!! Example:
!!
!!     program test_mpifx
!!       use libmpifx_module
!!       implicit none
!!
!!       type(mpifx_comm) :: mycomm
!!
!!       call mpifx_init_thread(MPI_THREAD_FUNNELED)
!!       call mycomm%init()
!!       :
!!       call mpifx_finalize()
!!
!!     end program test_mpifx
!!

subroutine mpifx_init_thread(requiredThreading, providedThreading, error)
```

MpiFx initialising MPI example

MpiFx comm structure

```
!> MPI communicator with some additional information.
type mpifx_comm
  integer :: id         !< Communicator id.
  integer :: size       !< Nr. of processes (size).
  integer :: rank       !< Rank of the current process.
  integer :: leadrank   !< Index of the lead node.
  logical :: lead       !< True if current process is the lead (rank == 0).
contains
  !> Initializes the MPI environment.
  procedure :: init => mpifx_comm_init

  !> Creates a new communicator by splitting the old one.
  procedure :: split => mpifx_comm_split

  !> Creates a new communicator by splitting the old one given a split type.
  procedure :: split_type => mpifx_comm_split_type

  !> Frees the communicator. The communicator should not be used after this.
  procedure :: free => mpifx_comm_free

end type mpifx_comm
```

```
subroutine mpifx_init_thread(requiredThreading, providedThreading, error)
    integer, intent(in) :: requiredThreading
    integer, intent(out), optional :: providedThreading
    integer, intent(out), optional :: error
    :

    call mpi_init_thread(requiredThreading, providedThreading, error)

    if (present(providedThreading)) then
      providedThreading = providedThreading0
    elseif (providedThreading < requiredThreading) then
      write(*, "(A,I0,A,I0,A)") "Error: Provided threading model (",&
          & providedThreading,") is less than required threading model (",&
          & requiredThreading, ")"
      call mpi_abort(MPI_COMM_WORLD, MPIFX_UNHANDLED_ERROR, error0)
    end if

    call handle_errorflag(error, "Error: mpi_init_thread in mpifx_init_thread()",&
      & error)

  end subroutine mpifx_init_thread
```

MpiFx internals for this – slightly pseudo-code

MPI constants are in numerical order for thread support level

DFTB+ MPI structure

```
!> Contains MPI related environment settings
type :: TMpiEnv
  !> Global MPI communicator
  type(mpifx_comm) :: globalComm
  !> Communicator to access processes within current group
  type(mpifx_comm) :: groupComm
  !> Communicator to access equivalent processes in other groups
  type(mpifx_comm) :: interGroupComm
  !> Communicator within the current node
  type(mpifx_comm) :: nodeComm
  !> Size of the process groups
  integer :: groupSize
  !> Number of processor groups
  integer :: nGroup
  !> Group index of the current process (starts with 0)
  integer :: myGroup
  !> Rank of the processes in the given group (with respect of globalComm)
  integer, allocatable :: groupMembersGlobal(:)
  !> Rank of the processes in the given group (with respect of MPI_COMM_WORLD)
  integer, allocatable :: groupMembersWorld(:)
  !> Whether current process is the global lead
  logical :: tGlobalLead
  !> Whether current process is the group lead
  logical :: tGroupLead
```

Split groups (spins, K-points,…)

Internal to shared-memory node (more later)

Patterns for today

1) OMP loops broken over MPI COMM
2) Shared memory windows with MPI
3) Re-distribute BLACS for simple OMP operations (WIP)
4) Loops as hidden MPI COMM operations (WIP)

Pattern 1) Hybrid MPI-OMP loop

```
call distributeRangeInChunks(env, 1, nAtom, iAtFirst, iAtLast)

! Put 1.0 for the diagonal elements of the overlap.
!$OMP PARALLEL DO PRIVATE(iAt1, iSp1, ind, iOrb1) DEFAULT(SHARED) SCHEDULE(RUNTIME)
do iAt1 = iAtFirst, iAtLast
  iSp1 = species(iAt1)
  ind = iPair(0,iAt1) + 1
  do iOrb1 = 1, orb%nOrbAtom(iAt1)
    over(ind) = 1.0_dp
    ind = ind + orb%nOrbAtom(iAt1) + 1
  end do
end do
!$OMP  END PARALLEL DO

call buildDiatomicBlocks(iAtFirst, iAtLast, skOverCont, coords, nNeighbourSK, iNeighbours,&
    & species, iPair, orb, over)

call assembleChunks(env, over)
```

Overlap matrix set-up example, looping over atoms.
For overlaps we are;

- block-diagonal on-site
- use a compressed block sparse array structure for this type of matrix

diatomic (off diagonal) elements handled in similar loop

Loop partitioning internals

```
!> Distributes a range in chunks over processes within a process group.
subroutine distributeRangeInChunks(env, globalFirst, globalLast, localFirst, localLast)

  !> Computational environment settings
  type(TEnvironment), intent(in) :: env

  !> First element of the range
  integer, intent(in) :: globalFirst

  !> Last element of the range
  integer, intent(in) :: globalLast

  !> First element to process locally
  integer, intent(out) :: localFirst

  !> Last element to process locally
  integer, intent(out) :: localLast

#:if WITH_MPI
  call getChunkRanges(env%mpi%groupComm%size, env%mpi%groupComm%rank, globalFirst,&
      & globalLast, localFirst, localLast)
#:else
  localFirst = globalFirst
  localLast = globalLast
#:endif

end subroutine distributeRangeInChunks
```

- groupComm divided off from COMM_WORLD
- Data structure in **env** includes
  - size of group
  - rank inside group
- globalFirst and globalLast loop ranges
- Analogues for nested loops (not shown)
- And some **preprocessing**

Non-MPI does whole range

MPI case gets to here

```
subroutine getChunkRanges()
    :
    rangeLength = globalLast - globalFirst + 1
    nLocal = rangeLength / groupSize
    remainder = mod(rangeLength, groupSize)

    if (myRank < remainder) then
      nLocal = nLocal + 1
      localFirst = globalFirst + myRank * nLocal
    else
      localFirst = globalFirst + remainder * (nLocal + 1) + (myRank - remainder) * nLocal
    end if

    localLast = min(localFirst + nLocal - 1, globalLast)

end subroutine getChunkRanges
```

GetChunkRanges internal
- groupSize COMM size
- myRank inside this group

Re-assemble at end of loop
– want all procs to get a copy of resulting array

```
!> Assembles the chunks by summing up contributions within a process group.
subroutine assemble${NAME}$Chunks(env,chunks)

  !> Environment settings
  type(TEnvironment), intent(in) :: env

  !> array to assemble
  ${DTYPE}$, intent(inout) :: chunks${FORTRAN_ARG_DIM_SUFFIX(RANK)}$

#:if WITH_MPI
  call mpifx_allreduceip(env%mpi%groupComm, chunks, MPI_SUM)
#:endif

end subroutine assemble${NAME}$Chunks
```

Some **Fypp** directives here – looped over variable types and conditional compilation for MPI

Re-assembly is just a reduce over relevant COMM group

Pattern 2) Shared-memory MPI windows

– recent contribution from Tobias Melson (MPCDF)

Analogous to OpenMP with locking and barriers

Splits made from global comm – splitting sub-communicators requires affinity or RMA

```
type mpifx_win
  private
  integer, public :: id   !< Window id.
  integer :: comm_id  !< Communicator id.
contains
  !> Initializes an MPI shared memory window.
#:for TYPE in TYPES
  generic :: allocate_shared => mpifx_win_allocate_shared_${TYPE_ABBREVS[TYPE]}$
#:endfor
#:for TYPE in TYPES
  procedure, private :: mpifx_win_allocate_shared_${TYPE_ABBREVS[TYPE]}$
#:endfor

  !> Locks a shared memory segment.
  procedure :: lock => mpifx_win_lock
  !> Unlocks a shared memory segment.
  procedure :: unlock => mpifx_win_unlock
  !> Synchronizes shared memory across MPI ranks.
  procedure :: sync => mpifx_win_sync
  !> Deallocates memory associated with a shared memory segment.
  procedure :: free => mpifx_win_free
end type mpifx_win
```

Incl. barrier

Currently used for neighbour map generation algorithm in DFTB+

Use in DFTB+

Node internal communicator (if possible)

```
call this%globalComm%split_type(MPI_COMM_TYPE_SHARED, this%globalComm%rank,&
  & this%nodeComm)
:
#:if WITH_MPI
      if (associated(neigh%iNeighbourMemory)) then
        call neigh%iNeighbourWin%free()
        nullify(neigh%iNeighbourMemory)
      end if
      dataLength = (maxNeighbour + 1) * nAtom
      call neigh%iNeighbourWin%allocate_shared(env%mpi%nodeComm, dataLength,&
          & neigh%iNeighbourMemory)
      neigh%iNeighbour(0:maxNeighbour,1:nAtom) => neigh%iNeighbourMemory(1:dataLength)
      maxNeighbourLocal = min(ubound(iNeighbour, dim=1), maxNeighbour)
      call neigh%iNeighbourWin%lock()
      neigh%iNeighbour(1:maxNeighbourLocal,startAtom:endAtom) =&
          & iNeighbour(1:maxNeighbourLocal,startAtom:endAtom)
      if (maxNeighbourLocal < maxNeighbour) then
        neigh%iNeighbour(maxNeighbourLocal+1:maxNeighbour,startAtom:endAtom) = 0
      end if
      call neigh%iNeighbourWin%sync()
      call neigh%iNeighbourWin%unlock()
#:endif
```

Part of the neighbour algorithm
- Storage in shared memory
- Neighbour generation similar to Hybrid MPI-OMP loop, but purely MPI (not shown)

Clean out old data

Allocate new

locking

Store local part in shared window

release

Pattern 3) Redistribute for easy OMP

WIP

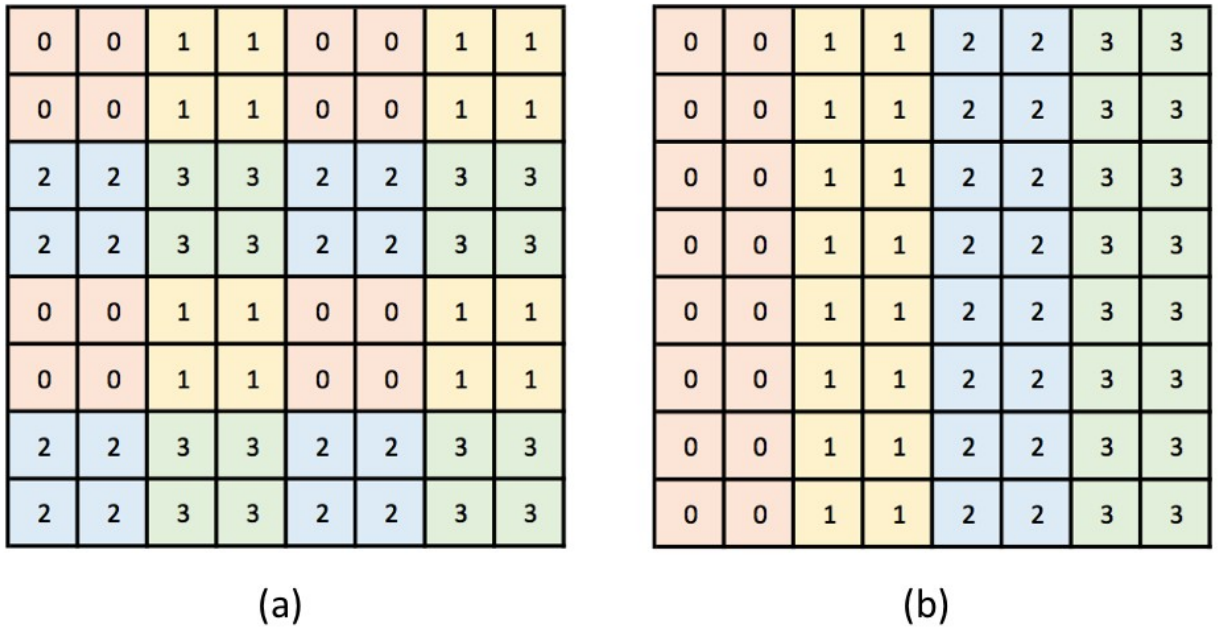
Need to solve some problems with conjugate gradient (CG), for multiple right sides (i.e. A X = B), and want to product DFTB+ sparse format matrices directly with distributed 2D BLACS matrices:

- Inversion of overlap S S^{-1} = 1 as S^{-1} is reasonably dense in mid-size systems.
- Sternheimer response properties, solving H C' = H' C for response of wavefunction due to H' (optionally with a projection onto virtual states).

Already have serial/OMP code for CG with multiple RHS (for response calculations, drops scaling from N^3 to ~$N^{2.2}$ with moderate sparsity at N_{basis}~4000 functions). CG requires multiple SYMM operations.

Data distribution

p?SYMM operation for BLACS would require communication for each CG iteration (with >10 steps depending on convergence and pre-conditioning).
Usual block cyclic BLACS (a) optimized for ‘single shot’ matrix operations.



V. W.-z. Yu et al. Comp. Phys. Comm. **222**, 267-285 (2018)

Cyclic/stripped (b) is amenable for multiple local SYMV operations with no communication (also allows for a few easy CG optimisations for multiple RHS converging at different rates).

BLACS re-distribution

p*GEMR2D from the BLACS redistribution routines would seem to do this perfectly for dense matrices. Set up an (n_{basis}, n_{basis}) matrix with row-like block sizes (n_{basis}, ~1), then:

- GEMR2D block cyclic → column distributed
- Do CG sparse stuff with local matrix part on each proc.
- GEMR2D column distributed → block cyclic

In principle a p*GEMR2D equivalent could be done simply with one-sided MPI comms. as the data pattern is pre-known.

Balancing problem with NUMROC

In DFTB+ we use an (approximately) square BLACS processor grid, and the default NUMROC leads to major imbalance in memory (and load) for column distributed:

- (m, m) = n_{procs} grid
  - (n_{basis},n_{basis}) block cyclic pattern, stores ~ n_{basis} / m elements on each proc
  - (n_{basis},n_{basis}) column pattern matrix, stores n^2_{basis} / m element on the first m procs, and 0 on the other m(m-1) processors.

Balancing with NUMROC

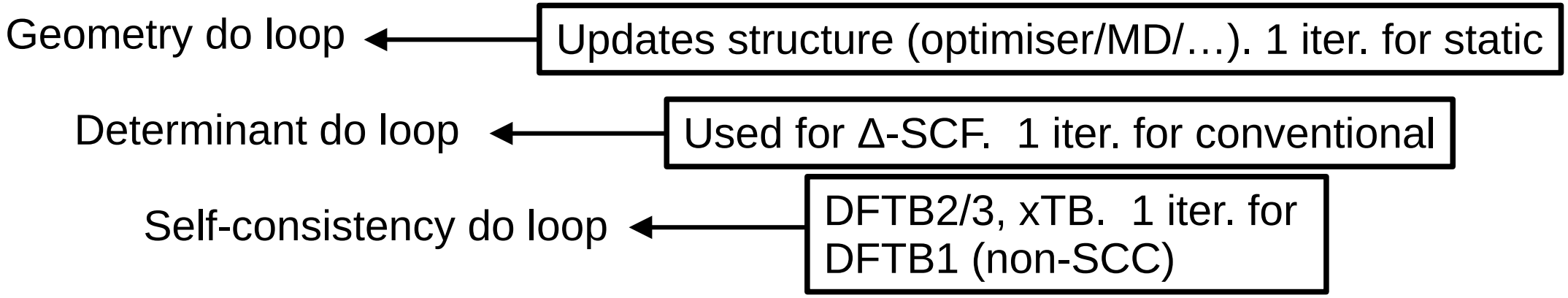
Instead use an (m^2, 1) BLACS grid on the same processors as the usual (m,m) block cyclic distribution, and NUMROC gives a balanced distribution:

- (m^2, 1) processor grid
  - (n_{basis},n_{basis}) cyclic, stores ~ n_{basis} / m elements on each proc.
  - Depending on ScaLAPACK block shape, cyclic is either interleaved (n_{basis},~1) or with multiple column together (n_{basis}, ~n_{basis}/m).

Pattern 4) Hiding MPI operations inside loops

WIP

Current main DFTB+ flow:



Determinant loop could also be used for
1) REKS/CI-/… multiple determinant methods
2) Finite difference derivatives wrt external fields
3) Constrained electronic states (undetermined multiplier and “external field”)

1&2 naturally parallel, 3 is sequential (optimization problem). Probably interesting to eventually mix 1, 2 & 3 together in same calculation.

Analogous cases for geometry loop (replica geometries) or self-consistency loop (determinants/constraints).

Data structure needs to have

Do loop treated more as a do while

Lends itself to OOP Fortran

Counts for independent cases to process

- Determines group splitting on comm world
- Determines storage copies on local processors

- Iterator which can handle collective operation over groups in split and global world
- Termination criteria to break loop (constraint case)
- Pre- and post-processing over stored copies
- Case dependant calculation modifications
- Initializer that accepts types for the different sorts of calculations with their own methods init[Δ-SCF_calc], init[finite_diff_calc], ...
  - and combinations init[finite_diff_calc[Δ-SCF_calc]]
- Some load balancing (if pool of case >> groups, shouldn't be to complicated for static).

Use similar pattern already elsewhere

Status and summary

1) OMP loops broken over MPI COMM

   Common use in various places

2) Shared memory windows with MPI

   Neighbour maps (so far)

3) Re-distribute BLACS for simple OMP operations

   GEMR2D in place, not used seriously yet

4) Loops as hidden MPI COMM operations

   WIP collection of PRs to refactor and extend