Trial tiling implementation using tra_ldf_iso

Last edited: 20/04/20

Overview

A trial approach to 2D tiling (over *i* and *j*) is implemented for the *tra_ldf_iso* subroutine.

This document describes an implementation using public variables and an alternative approach using derived types. A number of potential issues are then described.

Test configuration

A 1° configuration of GYRE is used with 1 CPU. XIOS is not used.

Branches

- Public variables
- Derived types

Variables

- Global variables
  - *ntsi*, *ntsj*- start index of tile
  - *ntei*, *ntej*- end index of tile
  - *ntsim1*, *ntsjm1*- start index of tile, minus 1
  - *nteip1*, *ntejp1*- end index of tile, plus 1
  - *ntile*- tile number
- Parameters
  - *jpnitile*, *jpnjtile*, *jpnijtile*- number of tiles
- Loop indices
  - *jtile*- loop over tiles
- Namelist
  - *ln_tile*- Logical control on use of tiling
  - *nn_tile_i*, *nn_tile_j*- tile length
- Pre-processor macros
  - *IND_2D*- substitution for ALLOCATE or DIMENSION arguments
- Working variables
  - *iitile*, *ijtile*- tile number
- Dummy arguments
  - *kntile* (*ntile*)

Namelist and tiling decomposition

The tiling decomposition is defined by parameters *jpnitile*, *jpnjtile* and *jpnijtile*, which are analogous to the MPP decomposition parameters *jpni*, *jpnj* and *jpnij*.

These parameters are declared in *par_oce* and set using a new namelist section, *namtile*.

```
!-----------------------------------------------------------------------
&namtile        !   parameters of the tiling
!-----------------------------------------------------------------------
```

```
      ln_tile = .false.      !  Use tiling (T) or not (F)
      nn_tile_i = 10         !  Length of tiles in i
      nn_tile_j = 10         !  Length of tiles in j
   /
```

The decomposition parameters are then calculated in *dom_nam*.

```
   IF( ln_tile ) THEN
      jpnitile = (jpi - 2) / nn_tile_i
      jpnjtile = (jpj - 2) / nn_tile_j
      IF( MOD( jpi - 2, nn_tile_i ) /= 0 ) jpnitile = jpnitile + 1
      IF( MOD( jpj - 2, nn_tile_j ) /= 0 ) jpnjtile = jpnjtile + 1
   ELSE
      jpnitile = 1
      jpnjtile = 1
   ENDIF
   jpnijtile = jpnitile * jpnjtile
```

Implementation of tile subdomains

The DO loop macros in *do_loop_substitute.h90* are modified to refer to a set of domain indices, instead of those for the full domain. A macro is added for *ALLOCATE* and *DIMENSION* arguments.

```
   #define __kIs_      2
   #define __kJs_      2
   #define __kIsm1_    1
   #define __kJsm1_    1
   #define __kIs_      ntsi
   #define __kJs_      ntsj
   #define __kIsm1_    ntsim1
   #define __kJsm1_    ntsjm1
   #define __kIe_      jpim1
   #define __kJe_      jpjm1
   #define __kIep1_    jpi
   #define __kJep1_    jpj
   #define __kIe_      ntei
   #define __kJe_      ntej
   #define __kIep1_    nteip1
   #define __kJep1_    ntejp1

   #define IND_2D      __kIsm1_:__kIep1_,__kJsm1_:__kJep1_
```

These domain indices are declared as public variables in *par_oce*. Their values are set by a new subroutine, *dom_tile*, which takes a tile number as its argument. A negative or zero tile number is used to set the domain indices to the full domain. This is used to initialise the indices in *dom_init*.

```
   SUBROUTINE dom_tile(kntile)
      INTEGER   , INTENT(in ) :: kntile
      INTEGER                 :: iitile, ijtile

      IF( ln_tile .AND. kntile > 0 ) THEN
         iitile = 1 + MOD( kntile - 1, jpnitile )
         ijtile = 1 + (kntile - 1) / jpnitile

         ntile = kntile
         ntsi = 2 + (iitile - 1) * nn_tile_i
         ntsj = 2 + (ijtile - 1) * nn_tile_j
         ntei = MIN(ntsi + nn_tile_i - 1, jpim1)
         ntej = MIN(ntsj + nn_tile_j - 1, jpjm1)
         ntsim1 = ntsi - 1
         ntsjm1 = ntsj - 1
         nteip1 = ntei + 1
         ntejp1 = ntej + 1
      ELSE
         ntile = 1
         ntsi = 2
         ntsj = 2
         ntei = jpim1
         ntej = jpjm1
         ntsim1 = 1
         ntsjm1 = 1
         nteip1 = jpi
         ntejp1 = jpj
      ENDIF
   END SUBROUTINE dom_tile
```

The tiling is implemented by looping over each tile at the time-stepping level (in *stp*) and calling *dom_tile* to set the domain indices for the present tile subdomain. The indices are restored to the full domain after exiting the tiling loop.

```
DO jtile = 1, jpnijtile
   IF( ln_tile ) CALL dom_tile( jtile )
   CALL tra_ldf( kstp, Nbb, Nnn, ts, Nrhs )
END DO
IF( ln_tile ) CALL dom_tile( 0 )
```

tra_ldf_iso is then modified where necessary to work on the tile subdomain, e.g. local working arrays

```
REAL(wp), DIMENSION(jpi,jpj)     ::   zdkt, zdk1t, z2d
REAL(wp), DIMENSION(jpi,jpj,jpk) ::   zdit, zdjt, zftu, zftv, ztfw
REAL(wp), DIMENSION(IND_2D)     ::   zdkt, zdk1t, z2d
REAL(wp), DIMENSION(IND_2D,jpk) ::   zdit, zdjt, zftu, zftv, ztfw
```

and operations on global arrays.

```
akz(:,:,:) = 0._wp
ah_wslp2(:,:,:) = 0._wp
DO_3D_11_11( 1, jpk )
   akz     (ji,jj,jk) = 0._wp
   ah_wslp2(ji,jj,jk) = 0._wp
END_3D
```

Some code in *tra_ldf* and *tra_ldf_iso*, mainly calls to other subroutines, has not been tiled and the best approach needs to be decided. This issue is summarised at the end of this document. At the moment this code is limited to running on one tile only; either the first (e.g. *WRITE* statements) or last (e.g. the *dia_ptr_hst* call) as appropriate.

Alternative implementation using derived types

In this implementation, the domain indices are contained within a derived type *TILE*, declared in *dom_oce*.

```
TYPE, PUBLIC ::   TILE
```

```
      INTEGER   ::   ntsi, ntsj, ntei, ntej, ntsim1, ntsjm1, nteip1, ntejp1, ntile
   END TYPE TILE
```

dom_tile instead returns an instance of *TILE*

```
   SUBROUTINE dom_tile(kntile, ktile)
      INTEGER    , INTENT(in ) :: kntile
      TYPE(TILE), INTENT(out) :: ktile
      INTEGER                  :: iitile, ijtile

      IF( ln_tile .AND. kntile > 0 ) THEN
         iitile = 1 + MOD( kntile - 1, jpnitile )
         ijtile = 1 + (kntile - 1) / jpnitile

         ktile % ntile = kntile
      ...
```

which is passed to *tra_ldf_iso* via an additional positional argument in *tra_ldf* and *tra_ldf_iso* (shown below for the call to *tra_ldf* in *stp*).

```
   DO jtile = 1, jpnijtile
      CALL dom_tile( jtile, stile )
      CALL tra_ldf( stile, kstp, Nbb, Nnn, ts, Nrhs )
   END DO
```

In *do_loop_substitute.h90*, duplicate DO loop macros have been added for tiled subroutines, e.g.

```
   #define __kIs_T     ktile % ntsi
   #define __kIe_T     ktile % ntei
   #define __kJs_T     ktile % ntsj
   #define __kJe_T     ktile % ntej
   #define DO_2D_00_00_T   DO jj = __kJs_T, __kJe_T   ;   DO ji = __kIs_T, __kIe_T
```

There is no need to call *dom_tile(0)* to revert to the global domain or to initialise in *dom_init*; untiled subroutines simply do not have the additional positional argument. However, each call to *tra_ldf_iso* and its calling subroutines must have this additional argument, even if that subroutine has not yet been tiled. This results in a more widespread set of code changes.

Because *dom_tile* returns an instance of *TILE*, rather than modifying public variables, the domain indices should be thread-private. However, this can also be achieved for the other approach by using an *OMP PRIVATE* directive.

Duplicate DO loop macros are needed in *do_loop_substitute.h90*, as not all subroutines will be tiled and *ktile* (the instance of *TILE* passed in) will be undefined in these cases.

In summary, using a derived type to pass around the domain indices is less tidy, involves more code changes and doesn't appear to have much functional benefit when compared to the approach using public variables.

Things to consider

Untiled code within the tiling loop

A number of operations within the tiling loop will not function correctly, as they will now be called multiple times.

1. Operations that should only be performed once
   - *WRITE* statements, e.g. *lwp* controlled writes to *numout*
   - NEMO timer (*timing_start*)

2. Operations that should not proceed until all tiles are at the same point
    - MPI operations (*mpp_sum*, etc)
    - Halo exchanges (*lbc_lnk*)
    - NEMO timer (*timing_stop*)
    - XIOS diagnostics (*iom_put*)

In the trial implementation, operations under 1. and 2. are only performed for the first and last tiles respectively. This clearly will not work if the tiles are to be run concurrently using e.g. OpenMP.

Some formal solutions are being developed (XIOS diagnostics) and some may be quite straightforward (*lwp* can depend on the tile number, *ntile*, in addition to the processor number, *narea*). Other issues require more thought, particularly with regards to MPP.

Extended haloes

The tiling implementation will be developed in tandem with the extended haloes development. There are a few potential issues here.

1. Refactoring of loops
2. Movement of *lbc_lnk* calls
3. Use of extended haloes, *nn_hls > 1*

The trial implementation of tiling mainly involves higher level code changes via the DO loop macros and only changes lower level code in a few cases, so 1. will hopefully not be too disruptive.

The tiling development should be able to proceed on 1 CPU, so 2. should not be an issue for the time being. Do we expect any *lbc_lnk* calls to remain in the tiling loop after the extended halo development? If so, how should we deal with this when using MPP?

Potentially, resolving 3. could be as simple as modifying the calculations where *nn_hls = 1* is assumed, e.g. the domain indices in *dom_tile*

```
ntsi = 1 + nn_hls + (iitile - 1) * nn_tile_i
ntei = MIN(ntsi + nn_tile_i - 1, jpi – nn_hls)
ntsi_1 = ntsi – nn_hls
ntei_1 = ntei + nn_hls
```

where *ntsi_1* and *ntei_1* have replaced *ntsim1* and *nteip1* as the indices for the full domain in this example.