Issues with the NEMO tiling implementation

Resolved issues

- Overlapping assignments

  Issue

  Each tile has an internal area and overlapping halo, as in the MPP and global domains. The internal part of a tile may therefore be partly overwritten by the halo of an adjacent tile, which will change results.

  Solution

  This must be handled on a case-by-case basis. For example, in `tra_ldf` the global arrays `akz` and `ah_wslp2` are zeroed at the start of several subroutines. To preserve the result, this must be done for the internal part of the tile only

  ```
  ! Changes result
  DO_3D_11_11(1, jpk)
     akz(ji,jj,jk) = 0._wp
  END_3D

  ! Does not change result
  DO_3D_00_00(1, jpk)
     akz(ji,jj,jk) = 0._wp
  END_3D
  ```

- Aggregation operations

  Issue

  The aggregation must also be applied over tiles.

  Workaround

  Use a `SAVE, ALLOCATABLE` array to store the aggregation result for the processor domain.

  E.g. for zonal integrals in `diaptr`

  1. Integrate over the tile domain
  2. Add this result to the `SAVE, ALLOCATABLE` array for each tile
  3. Call `mpp_sum` on this array after all tiles are finished

  Solution

  Use XIOS to perform the aggregation, once support for tile subdomains is available.

- `iom_put` calls

  Issue

  All tiles must be complete at the time of the call.

  Workaround

  Use a `SAVE, ALLOCATABLE` array to store each tile’s data, then call using this array after all tiles are finished.

  Solution

  Use XIOS once support for tile subdomains is available.

- `lbc_lnk` calls

  Issue

  All tiles must have been processed at the time of the call.

  Workaround

  Disable tiling for the entire subroutine.

  This is achieved by setting `ntile` to 0, indicating the full domain is to be used

  ```
  IF(ntile == 1) CALL dom_tile(ntsi, ntsj, ntei, ntej, ktile=0)
     CALL xxx
  IF(ln_tile .AND. ntile == 0) CALL dom_tile(ntsi, ntsj, ntei, ntej,
  ktile=1)
  ```

  Solution

  Remove `lbc_lnk` calls from routines called within the tiling loop.

  Calls used only for `iom_put` can be removed now. Others will remain in place while one-point haloes are still supported, but the associated workarounds can be removed by requiring that tiling only be used with multi-point haloes (`nn_hls` > 1).

- `fld_read` calls

  Issue

  Must be called only once per timestep.

  Solution

  Disable tiling for the call (as above, for `lbc_lnk` calls) and call only for the first tile.

- `trd_tra` calls

  Issue

  Is not currently tiled and contains `iom_put` calls.

  Workaround

  Use a `SAVE, ALLOCATABLE` array to store each tile's data, then call using this array after all tiles are finished.

  Solution

  Implement tiling in other `trd_tra` routines. XIOS support for tile subdomains is required.

- Declaration of dummy array argument dimensions

  Issue

  Consider the following subroutine

  ```
  SUBROUTINE eos_insitu( pts, prd, pdep )
     REAL(wp), DIMENSION(jpi,jpj,jpk,jpts), INTENT(in   ) ::   pts
     REAL(wp), DIMENSION(jpi,jpj,jpk     ), INTENT(  out) ::   prd
     REAL(wp), DIMENSION(jpi,jpj,jpk     ), INTENT(in   ) ::   pdep
  ```

  and the following two calls (via an `INTERFACE`) in `step.F90` and `zpshde.F90` respectively

  ```
  CALL eos( ts(:,:,:,:,Nbb), rhd, gdept_0(:,:,:) )
  CALL eos( zti, zhi, zri )
  ```

In the first call, the actual arguments are global arrays allocated in nemogcm.F90 with the dimensions of the full domain (jpi, jpj). In the second call, they are local working arrays in zps_hde declared with the dimensions of the tile (ntsi:ntsj, ntei:ntej) to minimise memory consumption.

The explicit shape declarations for the dummy arguments in eos_insitu will not conform with the actual arguments in both eos calls.

Solution

Use a wrapper routine to pass in the shape of the actual array arguments.

In the above example, eos_insitu is replaced by:

```
SUBROUTINE eos_insitu( pts, prd, pdep )
   REAL(wp), DIMENSION(:,:,:,:), INTENT(in   ) ::   pts
   REAL(wp), DIMENSION(:,:,:)  , INTENT(  out) ::   prd
   REAL(wp), DIMENSION(:,:,:)  , INTENT(in   ) ::   pdep

CALL eos_insitu_t( pts, is_tile(pts), prd, is_tile(prd), pdep,
is_tile(pdep) )
END SUBROUTINE eos_insitu

SUBROUTINE eos_insitu_t( pts, ktts, prd, ktrd, pdep, ktdep )
   INTEGER, INTENT(in   ) ::   ktts, ktrd, ktdep
   REAL(wp), DIMENSION(ST_2DT(ktts) ,jpk,jpts), INTENT(in   ) ::   pts
   REAL(wp), DIMENSION(ST_2DT(ktrd) ,jpk     ), INTENT(  out) ::   prd
   REAL(wp), DIMENSION(ST_2DT(ktdep),jpk     ), INTENT(in   ) ::   pdep
```

where is_tile is an INTERFACE of functions returning 0 or 1, e.g.:

```
PURE FUNCTION is_tile_2d( pt )
   REAL(wp), DIMENSION(:,:), INTENT(in) ::   pt
   INTEGER :: is_tile_2d
   IF( ln_tile .AND. SIZE(pt, 1) < jpi ) THEN
      is_tile_2d = 1
   ELSE
      is_tile_2d = 0
   ENDIF
   END FUNCTION is_tile_2d
```

and ST_2DT is a CPP macro returning “1:jpi,1:jpj“ or “ntsi:ntei,ntsj:ntej”:

```
#define ST_1DTi(T) (ntsi-nn_hls-1)*T+1:(ntei+nn_hls-jpi)*T+jpi
#define ST_1DTj(T) (ntsj-nn_hls-1)*T+1:(ntej+nn_hls-jpj)*T+jpj
#define ST_2DT(T) ST_1DTi(T),ST_1DTj(T)
```

In this example, the wrapper routine eos_insitu determines the shape of the input arguments and passes this information to eos_insitu_t, the original eos_insitu routine. The dummy array arguments are then dynamically declared with the correct explicit shape.

Other solutions

An ideal solution would be to use assumed-shape declarations, i.e. DIMENSION(:,:), but these do not work in this case as they do not preserve the bounds of the actual array argument. For example:

```
SUBROUTINE test( arg )
   REAL(wp), DIMENSION(:,:), INTENT(in) :: arg
   PRINT *,LBOUND(arg),UBOUND(arg)
```

```
REAL(wp), DIMENSION(3:7,5:9) :: arg

PRINT *,LBOUND(arg),UBOUND(arg)  ! Prints “(3,5) (7,9)”
CALL test(arg)                   ! Prints “(1,1) (5,5)”
```

Declaring tile arrays with a lower bound starting from > 1 allows DO loops to access both the tile and the equivalent part of the full domain. This is not possible if the bounds information is lost.

Unresolved issues

- `prt_ctl` calls

  Issue

  The result is not preserved when tiling is used.

  Solutions

  `prt_ctl` essentially takes the global sum of an array and prints the result. This is used to assess bit comparability and is similar to `stpctl` (which writes to the run.stat file), but is more granular and called from various places within the timestep loop.

  The same workaround as for other aggregation operations can be used, where the result is stored and aggregated over tiles for the processor domain. However, this introduces an extra step in the calculation and bit-level differences are introduced if more than one tile is used.

  While the use of tiling preserves results for `stpctl` and other diagnostics, it does not do so for `prtctl` diagnostics. Therefore, `prtctl` cannot be used to compare results with and without tiling.

  A further issue is that the sum result must be stored for each unique `prt_ctl` call. For example, the following call is from `eos_insitu`:

  ```
  IF(sn_cfctl%l_prtctl)   CALL prt_ctl( tab3d_1=prd, clinfo1=' eos-insitu
  : ', kdim=jpk )
  ```

  The result could be stored in a variable declared locally in `eos_insitu` and passed to `prt_ctl`. However, `eos_insitu` itself is called by more than one routine per timestep, so the variable would contain the sum over several calls.

  I have explored a solution that makes use of the descriptive information passed as part of the call, i.e. `clinfo1` in the above example, in order to identify unique `prt_ctl` calls. This is a work in progress.