```
#include "fabm_driver.h"

! ============================================================================
! Derived types that describe a single job
! ----------------------------------------------------------------------------
! A "job" typically describes the work (biogeochemical computations) that is
! done in a single call by the host model. For instance, it describes all calls
! to BGC model instances (and their order!) that need to be made when retrieving
! interior sources. A job can consists of multiple ordered "tasks",
! which in turn groups "calls" to procedures of specific model objects.
! All calls within a task operate on the same spatial domain, e.g., on the
! interior, surface, bottom, or on water columns.
! ============================================================================

module fabm_job

   use fabm_types
   use fabm_schedule
   use fabm_driver
   use fabm_graph
   use fabm_task_order

   implicit none

   private

   public type_job_manager, type_job, type_task, type_call
   public type_global_variable_register

   type type_variable_request
      type (type_internal_variable), pointer :: variable => null()
      type (type_output_variable_set)        :: output_variable_set
      logical                                :: store    =  .false.
      type (type_variable_request),  pointer :: next     => null()
   end type

   type type_call_request
      class (type_base_model),  pointer :: model  => null()
      integer                           :: source = source_unknown
      type (type_call_request), pointer :: next   => null()
   end type

   type type_cache_copy_command
      integer :: read_index  = -1
      integer :: write_index = -1
   end type

   ! A single call to a specific API of a specific biogeochemical model.
   ! It is defined by the combination of a model object ("model") and one of its procedures ("source").
   type type_call
      logical                          :: active = .true.
      integer                          :: source = source_unknown
      class (type_base_model), pointer :: model => null()
      integer                          :: ncopy_int = 0 ! interior variables to copy from write to read cache after call completes
      integer                          :: ncopy_hz = 0  ! horizontal variables to copy from write to read cache after call completes
      type (type_node), pointer        :: graph_node => null()
   end type type_call

   ! A task contains one or more model calls that all use the same operation over the domain.
   ! Valid operations: interior in native direction, interior per column, surface only, bottom only, horizontal-only.

   type type_task
      integer                       :: operation = source_unknown
      type (type_call), allocatable :: calls(:)

      integer, allocatable :: prefill(:)
      integer, allocatable :: prefill_hz(:)
      integer, allocatable :: save_sources(:)
      integer, allocatable :: save_sources_hz(:)
      integer, allocatable :: load(:)
      integer, allocatable :: load_hz(:)
      integer, allocatable :: load_scalar(:)
      type (type_cache_copy_command), allocatable :: copy_commands_int(:) ! interior variables to copy from write to read cache after call completes
      type (type_cache_copy_command), allocatable :: copy_commands_hz(:)  ! horizontal variables to copy from write to read cache after call completes

      type (type_task),  pointer :: next => null()
      class (type_job),  pointer :: job =>  null()

      type (type_variable_set), private :: read_cache_preload
      type (type_variable_set), private :: write_cache_preload
   contains
      procedure :: initialize => task_initialize
      procedure :: finalize   => task_finalize
      procedure :: print      => task_print
   end type

   ! Job states (used for debugging call order)
   integer, parameter :: job_state_none = 0
   integer, parameter :: job_state_created = 1
   integer, parameter :: job_state_graph_created = 2
   integer, parameter :: job_state_tasks_created = 3
   integer, parameter :: job_state_finalized_prefill_settings = 4
   integer, parameter :: job_state_initialized = 5

```

```
   type type_job_node
      class (type_job),     pointer :: p    => null()
      type (type_job_node), pointer :: next => null()
   end type

   type type_job_set
      type (type_job_node), pointer :: first => null()
   contains
      procedure :: add        => job_set_add
      procedure :: find_first => job_set_find_first
      procedure :: finalize   => job_set_finalize
   end type

   ! A job contains one or more tasks, each using their own specific operation over the domain.
   type type_job
      private

      type (type_task), pointer, public :: first_task => null()

      logical, allocatable, public :: interior_store_prefill(:)
      logical, allocatable, public :: horizontal_store_prefill(:)

      integer, allocatable, public :: arg1_sources(:)
      integer, allocatable, public :: arg2_sources(:)

      character(len=attribute_length) :: name = ''
      integer                         :: state = job_state_none
      logical                         :: outsource_tasks     = .false.
      integer                         :: operation = source_unknown
      logical                         :: flag = .false.

      type (type_variable_request), pointer :: first_variable_request => null()
      type (type_call_request),     pointer :: first_call_request     => null()

      type (type_variable_set), public  :: read_cache_loads
      type (type_variable_set)  :: store_prefills

      type (type_graph)   :: graph
      type (type_job_set) :: previous
   contains
      procedure :: request_variable => job_request_variable
      procedure :: request_call     => job_request_call
      procedure :: connect          => job_connect
      procedure :: print            => job_print
      procedure :: finalize         => job_finalize
   end type

   type, extends(type_job_set) :: type_job_manager
   contains
      procedure :: create      => job_manager_create
      procedure :: initialize  => job_manager_initialize
      procedure :: print       => job_manager_print
      procedure :: write_graph => job_manager_write_graph
      procedure :: finalize    => job_manager_finalize
   end type

   type type_variable_register
      type (type_variable_list) :: interior
      type (type_variable_list) :: horizontal
      type (type_variable_list) :: scalar
      logical, private :: frozen = .false.
   contains
      procedure :: finalize => variable_register_finalize
   end type

   type type_global_variable_register
      type (type_variable_register) :: catalog
      type (type_variable_register) :: store
      type (type_variable_register) :: read_cache
      type (type_variable_register) :: write_cache
      type (type_variable_set)      :: unfulfilled_dependencies
   contains
      procedure :: add_to_store       => global_variable_register_add_to_store
      procedure :: add_to_catalog     => global_variable_register_add_to_catalog
      procedure :: add_to_read_cache  => global_variable_register_add_to_read_cache
      procedure :: add_to_write_cache => global_variable_register_add_to_write_cache
      procedure :: print              => global_variable_register_print
      procedure :: finalize           => global_variable_register_finalize
   end type

contains

   subroutine task_initialize(self, variable_register, schedules)
      class (type_task),                    intent(inout) :: self
      type (type_global_variable_register), intent(inout) :: variable_register
      type (type_schedules),                intent(inout) :: schedules

      integer                                        :: icall
      type (type_input_variable_set_node),  pointer :: input_variable_node
      class (type_base_model),              pointer :: parent
      class (type_model_list_node),         pointer :: model_list_node
      type (type_output_variable_set_node), pointer :: variable_node

      ! Initialize individual call objects, then collect all input variables in a task-encompassing set.
      do icall = 1, size(self%calls)
         input_variable_node => self%calls(icall)%graph_node%inputs%first
         do while (associated(input_variable_node))
            _ASSERT_(.not. input_variable_node%p%target%read_indices%is_empty(), 'task_initialize', 'variable without
```

```
 read indices among inputs')
            _ASSERT_(.not. associated(input_variable_node%p%target%write_owner), 'task_initialize', 'write contributi
on among inputs')

            ! Make sure the variable has an entry in the read register.
            call variable_register%add_to_read_cache(input_variable_node%p%target)

            ! Make sure this input is loaded into the read cache before the task is started.
            ! Skip constants and unavailable (= optional) inputs
            if (input_variable_node%p%target%source /= source_constant .and. input_variable_node%p%target%source /= s
ource_unknown) &
               call self%read_cache_preload%add(input_variable_node%p%target)

            input_variable_node => input_variable_node%next
         end do

         ! Make sure the pointer to the model has the highest class (and not a base class)
         ! This is needed because model classes that use inheritance and call base class methods
         ! may end up with model pointers that are base class specific (and do not reference
         ! procedures overwritten at a higher level)
         _ASSERT_(associated(self%calls(icall)%model), 'task_initialize', 'Call without associated model pointer.')
         parent => self%calls(icall)%model%parent
         if (associated(parent)) then
            model_list_node => parent%children%first
            do while (associated(model_list_node))
               if (associated(self%calls(icall)%model, model_list_node%model)) then
                  ! Found ourselves in our parent - use the parent pointer to replace ours.
                  self%calls(icall)%model => model_list_node%model
                  exit
               end if
               model_list_node => model_list_node%next
            end do
         end if

         ! For all output variables that other models are interested in, decide whether to copy their value
         ! from the write to read cache [if the other model will be called as part of the same task],
         ! of to save it to the persistent data store.
         variable_node => self%calls(icall)%graph_node%outputs%first
         do while (associated(variable_node))
            call variable_register%add_to_write_cache(variable_node%p%target)
            if (variable_node%p%copy_to_cache) call variable_register%add_to_read_cache(variable_node%p%target)
            if (variable_node%p%copy_to_store) call variable_register%add_to_store(variable_node%p%target)
            variable_node => variable_node%next
         end do

         call schedules%attach(self%calls(icall)%model, self%calls(icall)%source, self%calls(icall)%active)
      end do
   end subroutine task_initialize

   subroutine task_process_indices(self)
      class (type_task), intent(inout) :: self

      integer                                          :: icall, n_int, n_hz
      type (type_output_variable_set_node), pointer :: output_variable

      do icall = 1, size(self%calls)
         self%calls(icall)%ncopy_int = get_copy_command_count(self%calls(icall), domain_interior)
         self%calls(icall)%ncopy_hz = get_copy_command_count(self%calls(icall), domain_horizontal)
      end do
      allocate(self%copy_commands_int(sum(self%calls%ncopy_int)))
      allocate(self%copy_commands_hz(sum(self%calls%ncopy_hz)))
      n_int = 0
      n_hz = 0
      do icall = 1, size(self%calls)
         call create_cache_copy_commands(self%calls(icall), self%copy_commands_int, domain_interior, n_int)
         call create_cache_copy_commands(self%calls(icall), self%copy_commands_hz, domain_horizontal, n_hz)
      end do
      _ASSERT_(n_int == sum(self%calls%ncopy_int), 'task_process_indices', 'mismatch in count of interior copy comman
ds')
      _ASSERT_(n_hz == sum(self%calls%ncopy_hz), 'task_process_indices', 'mismatch in count of horizontal copy comman
ds')
      _ASSERT_(all(self%copy_commands_int%read_index > 0), 'task_process_indices', 'one or more read_index values for
 interior copy command <= 0')
      _ASSERT_(all(self%copy_commands_int%write_index > 0), 'task_process_indices', 'one or more write_index values f
or interior copy command <= 0')
      _ASSERT_(all(self%copy_commands_hz%read_index > 0), 'task_process_indices', 'one or more read_index values for
horizontal copy command <= 0')
      _ASSERT_(all(self%copy_commands_hz%write_index > 0), 'task_process_indices', 'one or more write_index values fo
r horizontal copy command <= 0')

      ! For all variables that this task computes itself, there is no need to preload a value in cache.
      do icall = 1, size(self%calls)
         output_variable => self%calls(icall)%graph_node%outputs%first
         do while (associated(output_variable))
            if (associated(output_variable%p%target%write_owner)) then
               call self%read_cache_preload%remove(output_variable%p%target%write_owner, discard=.true.)
            else
               call self%read_cache_preload%remove(output_variable%p%target, discard=.true.)
            end if
            output_variable => output_variable%next
         end do
      end do

      ! Find all variables that must be written to persistent storage after this job completes.
      call create_persistent_store_commands(self%save_sources,    domain_interior)
      call create_persistent_store_commands(self%save_sources_hz, domain_horizontal)

      ! Create prefill instructions for all variables that will be written to.
```

```
281       call create_prefill_commands(self%prefill,    domain_interior)
282       call create_prefill_commands(self%prefill_hz, domain_horizontal)
283
284       ! Create read cache load instructions for all input variables.
285       call create_load_commands(self%load,        domain_interior)
286       call create_load_commands(self%load_hz,     domain_horizontal)
287       call create_load_commands(self%load_scalar, domain_scalar)
288
289    contains
290
291       integer function get_copy_command_count(call_node, domain)
292          type (type_call), intent(in) :: call_node
293          integer,          intent(in) :: domain
294
295          type (type_output_variable_set_node), pointer :: variable
296          integer                                       :: read_index
297
298          get_copy_command_count = 0
299          variable => call_node%graph_node%outputs%first
300          do while (associated(variable))
301             if (variable%p%copy_to_cache .and. iand(variable%p%target%domain, domain) /= 0) then
302                _ASSERT_(variable%p%target%write_indices%value > 0, 'get_copy_command_count', 'BUG: ' // trim(variable
    %p%target%name) // ' cannot be copied from write to read cache because it lacks a write cache index.')
303                read_index = variable%p%target%read_indices%value
304                if (associated(variable%p%target%write_owner)) read_index = variable%p%target%write_owner%read_indices
    %value
305                _ASSERT_(read_index > 0, 'get_copy_command_count', 'BUG: ' // trim(variable%p%target%name) // ' cannot
     be copied from write to read cache because it lacks a read cache index.')
306                get_copy_command_count = get_copy_command_count + 1
307             end if
308             variable => variable%next
309          end do
310       end function
311
312       subroutine create_cache_copy_commands(call_node, commands, domain, n)
313          type (type_call),               intent(in)    :: call_node
314          type (type_cache_copy_command), intent(inout) :: commands(:)
315          integer,                        intent(in)    :: domain
316          integer,                        intent(inout) :: n
317
318          type (type_output_variable_set_node), pointer :: variable
319          integer                                       :: max_write_index, read_index, write_index
320
321          ! We will order by source index (i.e., the index in the write cache) to hopefully increase cache hits.
322          ! To allow this ordering, first determine the maximum source index.
323          max_write_index = -1
324          variable => call_node%graph_node%outputs%first
325          do while (associated(variable))
326             if (variable%p%copy_to_cache .and. iand(variable%p%target%domain, domain) /= 0) then
327                max_write_index = max(max_write_index, variable%p%target%write_indices%value)
328             end if
329             variable => variable%next
330          end do
331
332          ! Now process all possible source indices in order, and create cache copy commands where required.
333          do write_index = 1, max_write_index
334             variable => call_node%graph_node%outputs%first
335             do while (associated(variable))
336                if (variable%p%copy_to_cache .and. iand(variable%p%target%domain, domain) /= 0 .and. variable%p%target
    %write_indices%value == write_index) then
337                   n = n + 1
338                   read_index = variable%p%target%read_indices%value
339                   if (associated(variable%p%target%write_owner)) read_index = variable%p%target%write_owner%read_indi
    ces%value
340                   commands(n)%read_index = read_index
341                   commands(n)%write_index = write_index
342                   exit
343                end if
344                variable => variable%next
345             end do
346          end do
347       end subroutine create_cache_copy_commands
348
349       subroutine create_prefill_commands(prefill, domain)
350          integer, intent(out), allocatable :: prefill(:)
351          integer, intent(in)               :: domain
352
353          integer                                       :: ilast
354          integer                                       :: icall
355          type (type_output_variable_set_node), pointer :: output_variable
356          type (type_variable_node),            pointer :: variable_node
357
358          ! Find the last write cache index
359          ilast = 0
360          do icall = 1, size(self%calls)
361             output_variable => self%calls(icall)%graph_node%outputs%first
362             do while (associated(output_variable))
363                if (output_variable%p%target%prefill /= prefill_none .and. iand(output_variable%p%target%domain, domai
    n) /= 0) then
364                   _ASSERT_(output_variable%p%target%write_indices%value > 0, 'create_prefill_commands', 'Variable ' /
    / trim(output_variable%p%target%name) // ' was registered for prefilling, but it does not have a write cache index.')
365                   _ASSERT_(output_variable%p%target%prefill /= prefill_previous_value .or. (output_variable%p%target%
    source == source_external .or. output_variable%p%target%source == source_state .or. output_variable%p%target%store_in
    dex /= store_index_none), 'create_prefill_commands','Variable ' // trim(output_variable%p%target%name) // ' has prefi
    ll==previous value, but it does not have data.')
366                   ilast = max(ilast, output_variable%p%target%write_indices%value)
367                end if
```

```
368               output_variable => output_variable%next
369            end do
370         end do
371         variable_node => self%write_cache_preload%first
372         do while (associated(variable_node))
373            if (iand(variable_node%target%domain, domain) /= 0) then
374               _ASSERT_(variable_node%target%write_indices%value > 0, 'create_prefill_commands', 'Variable ' // trim(
    variable_node%target%name) // ' is set to be preloaded to write cache, but it does not have a write cache index.')
375               _ASSERT_(variable_node%target%source == source_external .or. variable_node%target%source == source_sta
    te .or. variable_node%target%store_index /= store_index_none, 'create_prefill_commands','Variable ' // trim(variable_
    node%target%name) // ' requires preloading to write cache, but it does not have data.')
376               ilast = max(ilast, variable_node%target%write_indices%value)
377            end if
378            variable_node => variable_node%next
379         end do
380
381         allocate(prefill(ilast))
382         prefill(:) = prefill_none
383
384         if (ilast == 0) return
385
386         do icall = 1, size(self%calls)
387            output_variable => self%calls(icall)%graph_node%outputs%first
388            do while (associated(output_variable))
389               if (output_variable%p%target%prefill /= prefill_none .and. iand(output_variable%p%target%domain, domai
    n) /= 0) then
390                  ilast = output_variable%p%target%write_indices%value
391                  if (output_variable%p%target%prefill == prefill_previous_value) then
392                     if (associated(output_variable%p%target%write_owner)) then
393                        prefill(ilast) = output_variable%p%target%write_owner%catalog_index
394                     else
395                        prefill(ilast) = output_variable%p%target%catalog_index
396                     end if
397                  else
398                     prefill(ilast) = output_variable%p%target%prefill
399                  end if
400               end if
401               output_variable => output_variable%next
402            end do
403         end do
404         variable_node => self%write_cache_preload%first
405         do while (associated(variable_node))
406            if (iand(variable_node%target%domain, domain) /= 0) then
407               ilast = variable_node%target%write_indices%value
408               if (associated(variable_node%target%write_owner)) then
409                  prefill(ilast) = variable_node%target%write_owner%catalog_index
410               else
411                  prefill(ilast) = variable_node%target%catalog_index
412               end if
413            end if
414            variable_node => variable_node%next
415         end do
416      end subroutine create_prefill_commands
417
418      subroutine create_persistent_store_commands(commands, domain)
419         integer, allocatable, intent(out) :: commands(:)
420         integer,              intent(in)  :: domain
421
422         integer                                        :: ilast
423         integer                                        :: icall
424         type (type_output_variable_set_node), pointer :: variable_node
425
426         ! First find the last index in persistent storage that will be written to.
427         ilast = 0
428         do icall = 1, size(self%calls)
429            variable_node => self%calls(icall)%graph_node%outputs%first
430            do while (associated(variable_node))
431               if (variable_node%p%copy_to_store .and. iand(variable_node%p%target%domain, domain) /= 0 .and. self%ca
    lls(icall)%graph_node%source /= source_constant) then
432                  _ASSERT_(variable_node%p%target%write_indices%value > 0, 'create_prefill_commands', 'Variable ' //
    trim(variable_node%p%target%name) // ' has copy_to_store set, but it does not have a write cache index.')
433                  _ASSERT_(variable_node%p%target%store_index /= store_index_none, 'create_persistent_store_commands'
    , 'Variable ' // trim(variable_node%p%target%name) // ' has copy_to_store set, but it does not have a persistent stor
    age index.')
434                  ilast = max(ilast,variable_node%p%target%store_index)
435               end if
436               variable_node => variable_node%next
437            end do
438         end do
439
440         ! Allocate the commands array (go up to the last written-to index in persistent storage only)
441         allocate(commands(ilast))
442         commands(:) = 0
443
444         ! Associate indices in persistent storage with the index in the write cache at which the source variable wil
    l be found.
445         do icall = 1, size(self%calls)
446            variable_node => self%calls(icall)%graph_node%outputs%first
447            do while (associated(variable_node))
448               if (variable_node%p%copy_to_store .and. iand(variable_node%p%target%domain,domain) /= 0 .and. self%cal
    ls(icall)%graph_node%source /= source_constant) &
449                  commands(variable_node%p%target%store_index) = variable_node%p%target%write_indices%value
450               variable_node => variable_node%next
451            end do
452         end do
453      end subroutine create_persistent_store_commands
454
455      subroutine create_load_commands(load, domain)
```

```
456          integer, allocatable, intent(out) :: load(:)
457          integer,              intent(in)  :: domain
458
459          integer                           :: ilast
460          type (type_variable_node), pointer :: input_variable
461
462          ! First determine the maximum index to preload from/to. That determines the length of the array with preload
    ing instructions.
463          ilast = 0
464          input_variable => self%read_cache_preload%first
465          do while (associated(input_variable))
466             _ASSERT_(.not. input_variable%target%read_indices%is_empty(), 'create_load_commands', 'Variable ' // trim
    (input_variable%target%name) // ' is marked for preloading but has no read indices.')
467             _ASSERT_(input_variable%target%read_indices%value /= -1, 'create_load_commands', 'Variable ' // trim(inpu
    t_variable%target%name) // ' is marked for preloading but has no valid read index.')
468             _ASSERT_(input_variable%target%source == source_external .or. input_variable%target%source == source_stat
    e .or. input_variable%target%store_index /= -1, 'create_load_commands', 'Variable ' // trim(input_variable%target%nam
    e) // ' is marked for preloading but has no valid data.')
469             if (iand(input_variable%target%domain, domain) /= 0) ilast = max(ilast, input_variable%target%read_indice
    s%value)
470             input_variable => input_variable%next
471          end do
472
473          ! Allocate array with preloading instructions, and initialize these to "do not preload"
474          allocate(load(ilast))
475          load(:) = 0
476
477          ! Flag variables that require preloading
478          input_variable => self%read_cache_preload%first
479          do while (associated(input_variable))
480             if (iand(input_variable%target%domain, domain) /= 0) load(input_variable%target%read_indices%value) = inp
    ut_variable%target%catalog_index
481             input_variable => input_variable%next
482          end do
483       end subroutine create_load_commands
484
485    end subroutine task_process_indices
486
487    subroutine job_print(self, unit, specific_variable)
488       class (type_job),                               intent(in) :: self
489       integer,                                        intent(in) :: unit
490       type (type_internal_variable), target, optional, intent(in) :: specific_variable
491
492       type (type_task), pointer :: task
493
494       write (unit,'(a,a)') 'Job: ',trim(self%name)
495       task => self%first_task
496       do while (associated(task))
497          write (unit,'(a,a)') '- TASK WITH OPERATION = ',trim(source2string(task%operation))
498          call task%print(unit, indent=2, specific_variable=specific_variable)
499          task => task%next
500       end do
501    end subroutine job_print
502
503    subroutine job_finalize(self)
504       class (type_job), intent(inout) :: self
505
506       type (type_task),             pointer :: task, next_task
507       type (type_variable_request), pointer :: variable_request, next_variable_request
508       type (type_call_request),     pointer :: call_request, next_call_request
509
510       task => self%first_task
511       do while (associated(task))
512          next_task => task%next
513          call task%finalize()
514          deallocate(task)
515          task => next_task
516       end do
517       self%first_task => null()
518
519       variable_request => self%first_variable_request
520       do while (associated(variable_request))
521          next_variable_request => variable_request%next
522          call variable_request%output_variable_set%finalize(owner=.false.)
523          deallocate(variable_request)
524          variable_request => next_variable_request
525       end do
526       self%first_variable_request => null()
527
528       call_request => self%first_call_request
529       do while (associated(call_request))
530          next_call_request => call_request%next
531          deallocate(call_request)
532          call_request => next_call_request
533       end do
534       self%first_call_request => null()
535
536       call self%read_cache_loads%finalize()
537       call self%store_prefills%finalize()
538       call self%previous%finalize()
539       call self%graph%finalize()
540    end subroutine job_finalize
541
542    subroutine print_output_variable(variable, unit, indent)
543       type (type_output_variable), intent(in) :: variable
544       integer,                     intent(in) :: unit
545       integer, optional,           intent(in) :: indent
546
```

```
547       integer                              :: read_index
548       type (type_node_set_member),pointer :: pnode
549
550       read_index = variable%target%read_indices%value
551       if (associated(variable%target%write_owner)) read_index = variable%target%write_owner%read_indices%value
552
553       write (unit,'(a,"   - ",a,", write@",i0)',advance='no') repeat(' ',indent), trim(variable%target%name), variabl
    e%target%write_indices%value
554       if (variable%copy_to_cache) write (unit,'(", cache@",i0)',advance='no') read_index
555       if (variable%copy_to_store) write (unit,'(", store@",i0)',advance='no') variable%target%store_index
556       write (unit,*)
557       pnode => variable%dependent_nodes%first
558       do while (associated(pnode))
559          if (associated(pnode%p%model)) then
560             write (unit,'(a,"     <- ",a,": ",a)') repeat(' ',indent), trim(pnode%p%model%get_path()), trim(source2st
    ring(pnode%p%source))
561          else
562             write (unit,'(a,"     <- host")') repeat(' ',indent)
563          end if
564          pnode => pnode%next
565       end do
566    end subroutine print_output_variable
567
568    subroutine print_input_variable(variable, unit, indent)
569       type (type_internal_variable), intent(in) :: variable
570       integer,                       intent(in) :: unit
571       integer,optional,              intent(in) :: indent
572
573       write (unit,'(a,"   - ",a,", read@",i0)') repeat(' ',indent), trim(variable%name), variable%read_indices%value
574    end subroutine print_input_variable
575
576    subroutine task_print(self, unit, indent, specific_variable)
577       class (type_task),                                intent(in) :: self
578       integer,                                          intent(in) :: unit
579       integer,optional,                                 intent(in) :: indent
580       type (type_internal_variable), target, optional, intent(in) :: specific_variable
581
582       integer                                        :: indent_
583       integer                                        :: i
584       integer                                        :: icall
585       type (type_output_variable_set_node), pointer :: output_variable
586       logical                                        :: show
587       type (type_input_variable_set_node),  pointer :: input_variable
588       logical                                        :: header_written
589       logical                                        :: subheader_written
590
591       indent_ = 0
592       if (present(indent)) indent_ = indent
593
594       if (size(self%load) > 0 .or. size(self%load_hz) > 0 .or. size(self%load_scalar) > 0) write (unit,'(a,a)') repea
    t(' ', indent_), 'Read cache prefilling:'
595       do i = 1, size(self%load)
596          if (self%load(i) /= 0) write (unit,'(a,"  - interior[",i0,"]")') repeat(' ', indent_), i
597       end do
598       do i = 1, size(self%load_hz)
599          if (self%load_hz(i) /= 0) write (unit,'(a,"  - horizontal[",i0,"]")') repeat(' ', indent_), i
600       end do
601       do i = 1, size(self%load_scalar)
602          if (self%load_scalar(i) /= 0) write (unit,'(a,"  - scalar[",i0,"]")') repeat(' ', indent_), i
603       end do
604
605       if (size(self%prefill) > 0 .or. size(self%prefill_hz) > 0) write (unit,'(a,a)') repeat(' ', indent_), 'Write ca
    che prefilling:'
606       do i = 1, size(self%prefill)
607          select case (self%prefill(i))
608          case (prefill_none)
609          case (prefill_constant)
610             write (unit,'(a,"  - interior[",i0,"] = constant value")') repeat(' ', indent_), i
611          case default
612             write (unit,'(a,"  - interior[",i0,"] = previous value")') repeat(' ', indent_), i
613          end select
614       end do
615       do i = 1, size(self%prefill_hz)
616          select case (self%prefill_hz(i))
617          case (prefill_none)
618          case (prefill_constant)
619             write (unit,'(a,"  - horizontal[",i0,"] = constant value")') repeat(' ', indent_), i
620          case default
621             write (unit,'(a,"  - horizontal[",i0,"] = previous value")') repeat(' ', indent_), i
622          end select
623       end do
624
625       do icall = 1, size(self%calls)
626          header_written = .false.
627          if (.not. present(specific_variable)) call write_header()
628
629          subheader_written = .false.
630          input_variable => self%calls(icall)%graph_node%inputs%first
631          do while (associated(input_variable))
632             show = .true.
633             if (present(specific_variable)) show = associated(input_variable%p%target, specific_variable)
634             if (show) then
635                call write_header()
636                if (.not. subheader_written) then
637                   write (unit,'(a,"   ",a)') repeat(' ', indent_), 'inputs:'
638                   subheader_written = .true.
639                end if
640                call print_input_variable(input_variable%p%target, unit, indent_)
```

```
641             end if
642             input_variable => input_variable%next
643          end do
644
645          subheader_written = .false.
646          output_variable => self%calls(icall)%graph_node%outputs%first
647          do while (associated(output_variable))
648             show = .true.
649             if (present(specific_variable)) show = associated(output_variable%p%target, specific_variable)
650             if (show) then
651                call write_header()
652                if (.not. subheader_written) then
653                   write (unit,'(a,"   ",a)') repeat(' ', indent_), 'outputs:'
654                   subheader_written = .true.
655                end if
656                call print_output_variable(output_variable%p, unit, indent_)
657             end if
658             output_variable => output_variable%next
659          end do
660       end do
661
662    contains
663
664       subroutine write_header()
665          if (.not. header_written) then
666             if (associated(self%calls(icall)%model)) then
667                write (unit,'(a,a,": ",a)') repeat(' ', indent_), trim(self%calls(icall)%model%get_path()), trim(sourc
   e2string(self%calls(icall)%source))
668             else
669                write (unit,'(a,"host")') repeat(' ', indent_)
670             end if
671             header_written = .true.
672          end if
673       end subroutine
674
675    end subroutine task_print
676
677    subroutine task_finalize(self)
678       class (type_task), intent(inout) :: self
679
680       call self%read_cache_preload%finalize()
681       call self%write_cache_preload%finalize()
682    end subroutine task_finalize
683
684    subroutine job_request_variable(self, variable, store)
685       class (type_job),target,       intent(inout)         :: self
686       type (type_internal_variable), intent(inout), target :: variable
687       logical, optional,             intent(in)            :: store
688
689       type (type_variable_request), pointer :: variable_request
690
691       _ASSERT_(self%state >= job_state_created, 'job_request_variable', 'Job has not been created yet.')
692       _ASSERT_(self%state <= job_state_created, 'job_request_variable', 'Job "' // trim(self%name) // '" has already
   begun initialization; variables can no longer be requested.')
693
694       ! Make sure this variable will not be merged (thus variable request must be filed before starting to merge vari
   ables!)
695       variable%write_operator = ior(variable%write_operator, operator_merge_forbidden)
696
697       _ASSERT_(.not. associated(variable%write_owner), 'job_request_variable','BUG: requested variable is co-written.
   ')
698
699       allocate(variable_request)
700       variable_request%variable => variable
701       if (present(store)) variable_request%store = store
702       variable_request%next => self%first_variable_request
703       self%first_variable_request => variable_request
704    end subroutine job_request_variable
705
706    subroutine job_request_call(self, model, source)
707       class (type_job),target,intent(inout) :: self
708       class (type_base_model),intent(in),target :: model
709       integer,                intent(in)    :: source
710
711       type (type_call_request), pointer :: call_request
712
713       _ASSERT_(self%state >= job_state_created, 'job_request_call', 'Job has not been created yet.')
714       _ASSERT_(self%state <= job_state_created, 'job_request_call', 'Job "' // trim(self%name) // '" has already begu
   n initialization; calls can no longer be requested.')
715
716       if (.not. model%implements(source)) return
717       allocate(call_request)
718       call_request%model => model
719       call_request%source = source
720       call_request%next => self%first_call_request
721       self%first_call_request => call_request
722    end subroutine job_request_call
723
724    subroutine job_create_graph(self, variable_register)
725       class (type_job), target,             intent(inout) :: self
726       type (type_global_variable_register), intent(inout) :: variable_register
727
728       type (type_job_node),        pointer :: job_node
729       type (type_variable_request),pointer :: variable_request
730       type (type_call_request),    pointer :: call_request, next_call_request
731       type (type_node),            pointer :: graph_node
732
733       _ASSERT_(self%state >= job_state_created, 'job_create_graph', 'This job has not been created yet.')
```

```
734       _ASSERT_(self%state < job_state_graph_created, 'job_create_graps', trim(self%name) // ': graph for this job has
     already been created.')
735
736       ! If we are linked to an earlier called job, make sure its graph has been created already.
737       ! This is essential because we can skip calls if they appear already in the previous job - we determine this by
     exploring its graph.
738       job_node => self%previous%first
739       do while (associated(job_node))
740          _ASSERT_(job_node%p%state >= job_state_graph_created, 'job_create_graph', trim(self%name) // ': graph for pr
evious job (' // trim(job_node%p%name) // ') has not been created yet.')
741          job_node => job_node%next
742       end do
743
744       ! Construct the dependency graph by adding explicitly requested variables and calls.
745       ! We clean up [deallocate] the variable and call requests at the same time.
746       variable_request => self%first_variable_request
747       do while (associated(variable_request))
748          call self%graph%add_variable(variable_request%variable, variable_request%output_variable_set, copy_to_store=
variable_request%store)
749          if (variable_request%store) then
750             ! FABM must be able to provide data for this variable across the entire spatial domain.
751             ! If this variable is a constant, explicitly request a data field for it in the persistent store.
752             ! If it is not a constant, the above call to add_variable will ensure that if the variable needs explicit
     computation,
753             ! its value will also be copied to the persistent store.
754             if (variable_request%variable%source == source_constant) call variable_register%add_to_store(variable_req
uest%variable)
755          else
756             if (.not. associated(variable_request%output_variable_set%first)) call variable_register%add_to_write_cac
he(variable_request%variable)
757          end if
758          variable_request => variable_request%next
759       end do
760
761       call_request => self%first_call_request
762       do while (associated(call_request))
763          next_call_request => call_request%next
764          graph_node => self%graph%add_call(call_request%model, call_request%source)
765          deallocate(call_request)
766          call_request => next_call_request
767       end do
768       self%first_call_request => null()
769
770       !self%graph%frozen = .true.
771
772       self%state = job_state_graph_created
773    end subroutine job_create_graph
774
775    subroutine job_create_tasks(self, log_unit)
776       class (type_job), target, intent(inout) :: self
777       integer,                  intent(in)    :: log_unit
778
779       type (type_job_node),                  pointer :: job_node
780       type (type_step), allocatable                  :: steps(:)
781       integer                                        :: itask
782       type (type_task),                      pointer :: task
783       integer                                        :: ncall
784       integer                                        :: icall
785       type (type_graph_subset_node_pointer), pointer :: pnode
786
787       _ASSERT_(self%state < job_state_tasks_created, 'job_create_tasks', trim(self%name) // ': tasks for this job hav
e already been created.')
788       _ASSERT_(self%state >= job_state_graph_created, 'job_create_tasks', trim(self%name) // ': the graph for this jo
b have not been created yet.')
789
790       ! If we are linked to an earlier called job, make sure its task list has already been created.
791       ! This is essential if we will try to outsource our own calls to previous tasks/jobs.
792       job_node => self%previous%first
793       do while (associated(job_node))
794          _ASSERT_(job_node%p%state >= job_state_tasks_created, 'job_create_tasks', trim(self%name) // ': tasks for pr
evious job (' // trim(job_node%p%name) // ') have not been created yet.')
795          job_node => job_node%next
796       end do
797
798       if (log_unit /= -1) write (log_unit,'(a)') trim(self%name)
799       call find_best_order(self%graph, self%operation, log_unit, steps)
800
801       ! Build task list by prepending
802       do itask = size(steps), 1, -1
803          ! Create the task and preprend it to the list.
804          allocate(task)
805          task%job => self
806          task%next => self%first_task
807          self%first_task => task
808          task%operation = steps(itask)%operation
809
810          ncall = 0
811          pnode => steps(itask)%first
812          do while (associated(pnode))
813             ncall = ncall + 1
814             pnode => pnode%next
815          end do
816          allocate(task%calls(ncall))
817
818          ! Collect all calls for this task.
819          ! Preserve the order in which calls appear in the "step",
820          ! as this also represents the desired call order.
821          pnode => steps(itask)%first
```

```
822          do icall = 1, ncall
823             task%calls(icall)%graph_node => pnode%p%graph_node
824             task%calls(icall)%model      => pnode%p%graph_node%model
825             task%calls(icall)%source     =  pnode%p%graph_node%source
826             _ASSERT_(associated(task%calls(icall)%model), 'create_tasks', 'Call node does not have a model pointer.')
827             _ASSERT_(task%calls(icall)%source /= source_constant .and. task%calls(icall)%source /= source_state .and. task%calls(icall)%source /= source_external .and. task%calls(icall)%source /= source_unknown, 'create_tasks', 'Call node has invalid source.')
828             pnode => pnode%next
829          end do
830
831          ! Clean-up array with processed calls.
832          call steps(itask)%finalize()
833       end do
834
835       if (self%outsource_tasks) then
836          task => self%first_task
837          do while (associated(task))
838             do icall = 1, size(task%calls)
839                !call move_call_backwards(self, task, task%calls(icall))
840             end do
841             task => task%next
842          end do
843       end if
844
845       if (associated(self%first_task)) then
846          _ASSERT_(self%operation == source_unknown .or. .not. associated(self%first_task%next), 'job_select_order', 'Multiple tasks created while only one source was acceptable.')
847       end if
848
849       self%state = job_state_tasks_created
850
851    !contains
852
853       !subroutine move_call_backwards(job,task,call_node)
854       !   class (type_job),target :: job
855       !   type (type_task),target :: task
856       !   type (type_call),target :: call_node
857       !
858       !   class (type_job),pointer :: current_job
859       !   type (type_task),pointer :: current_task, target_task
860       !   type (type_call),pointer :: current_call
861       !   integer :: operation_after_merge
862       !   logical :: compatible
863       !
864       !   write (*,*) 'moving '//trim(call_node%graph_node%as_string())
865       !   current_job => job
866       !   current_task => task
867       !   target_task => null()
868       !   compatible = .true.
869       !   do while (move_one_step_backwards(current_job,current_task,call_node))
870       !      operation_after_merge = current_task%operation
871       !      compatible = is_source_compatible(operation_after_merge,call_node%source)
872       !      compatible = compatible .and. operation_after_merge==current_task%operation
873       !      if (compatible) then
874       !         write (*,*) '  new task is compatible'
875       !         target_task => current_task
876       !      end if
877       !   end do
878       !   if (associated(target_task)) then
879       !      ! Remove node from original task
880       !      if (associated(task%first_call,call_node)) then
881       !         task%first_call => call_node%next
882       !      else
883       !         current_call => task%first_call
884       !         do while (.not.associated(current_call%next,call_node))
885       !            current_call => current_call%next
886       !         end do
887       !         current_call%next => call_node%next
888       !      end if
889       !
890       !      ! Append to target task
891       !      current_call => target_task%first_call
892       !      do while (associated(current_call%next))
893       !         current_call => current_call%next
894       !      end do
895       !      current_call%next => call_node
896       !      call_node%next => null()
897       !
898       !      !call target_task%initialize()
899       !   end if
900       !end subroutine move_call_backwards
901       !
902       !function move_one_step_backwards(job, task, travelling_call) result(moved)
903       !   class (type_job),pointer :: job
904       !   type (type_task),pointer :: task
905       !   type (type_call),intent(in) :: travelling_call
906       !   logical :: moved
907       !
908       !   type (type_call),pointer             :: call_node
909       !   type (type_node_set_member), pointer :: dependency
910       !
911       !   moved = .false.
912       !
913       !   ! First determine if we can leave the current task
914       !   ! (we cannot if it also handles one or more of our dependencies)
915       !   call_node => task%first_call
```

```
916       !   do while (associated(call_node))
917       !      dependency => travelling_call%graph_node%dependencies%first
918       !      do while (associated(dependency))
919       !         if (associated(dependency%p,call_node%graph_node)) then
920       !            write (*,*) '  cannot move past '//trim(call_node%graph_node%as_string())
921       !            return
922       !         end if
923       !         dependency => dependency%next
924       !      end do
925       !      call_node => call_node%next
926       !   end do
927       !
928       !   ! Move to previous task (if any)
929       !   call get_previous_task(job,task)
930       !   if (.not.associated(task)) return
931       !
932       !   moved = .true.
933         !end function move_one_step_backwards
934
935      end subroutine job_create_tasks
936
937      function get_last_task(job) result(task)
938         class (type_job),intent(in) :: job
939         type (type_task),pointer :: task
940
941         task => job%first_task
942         if (.not. associated(task)) return
943         do while (associated(task%next))
944            task => task%next
945         end do
946      end function get_last_task
947
948      recursive function find_responsible_task(job, output_variable) result(task)
949         class (type_job), intent(in)        :: job
950         type (type_output_variable), target :: output_variable
951         type (type_task), pointer :: task
952         type (type_job_node), pointer :: job_node
953
954         task => job%first_task
955         do while (associated(task))
956            if (task_is_responsible(task, output_variable)) return
957            task => task%next
958         end do
959
960         job_node => job%previous%first
961         do while (associated(job_node) .and. .not. associated(task))
962            task => find_responsible_task(job_node%p, output_variable)
963            job_node => job_node%next
964         end do
965      end function
966
967      logical function task_is_responsible(task, output_variable)
968         class (type_task), intent(in)       :: task
969         type (type_output_variable), target :: output_variable
970
971         integer                                        :: icall
972         type (type_output_variable_set_node), pointer :: output_variable_node
973
974         task_is_responsible = .true.
975         do icall = 1, size(task%calls)
976            ! Loop over all outputs of this call
977            output_variable_node => task%calls(icall)%graph_node%outputs%first
978            do while (associated(output_variable_node))
979               if (associated(output_variable_node%p, output_variable)) return
980               output_variable_node => output_variable_node%next
981            end do
982         end do
983         task_is_responsible = .false.
984      end function
985
986      logical function output_is_produced_before(task, reference_output_variable, output_variable)
987         class (type_task), intent(in)       :: task
988         type (type_output_variable), target :: reference_output_variable, output_variable
989
990         integer                                        :: icall
991         type (type_output_variable_set_node), pointer :: output_variable_node
992
993         output_is_produced_before = .false.
994         do icall = 1, size(task%calls)
995            ! Loop over all outputs of this call
996            output_variable_node => task%calls(icall)%graph_node%outputs%first
997            do while (associated(output_variable_node))
998               if (associated(output_variable_node%p, output_variable)) output_is_produced_before = .true.
999               if (associated(output_variable_node%p, reference_output_variable)) return
1000              output_variable_node => output_variable_node%next
1001           end do
1002        end do
1003        _ASSERT_(.false., 'output_is_produced_before', 'reference output not found in task')
1004     end function output_is_produced_before
1005
1006     subroutine job_finalize_prefill_settings(self)
1007        class (type_job), target, intent(inout) :: self
1008
1009        type (type_task),                    pointer :: task, last_task
1010        class (type_job),                    pointer :: first_job
1011        type (type_variable_request),        pointer :: variable_request
1012        type (type_output_variable_set_node), pointer :: output_variable
1013        logical                                        :: responsible
```

```
1014
1015       _ASSERT_(self%state < job_state_finalized_prefill_settings, 'job_finalize_prefill_settings', 'This job has alre
     ady been initialized.')
1016       _ASSERT_(self%state >= job_state_tasks_created, 'job_finalize_prefill_settings', 'Tasks for this job have not b
     een created yet.')
1017
1018       first_job => self
1019       do while (associated(first_job%previous%first))
1020          first_job => first_job%previous%first%p
1021       end do
1022
1023       ! Set copy-to-cache and copy-to-store based on dependencies between different calls/tasks.
1024       task => self%first_task
1025       do while (associated(task))
1026          call prepare_task(task)
1027          task => task%next
1028       end do
1029
1030       last_task => get_last_task(self)
1031
1032       variable_request => self%first_variable_request
1033       do while (associated(variable_request))
1034          if (.not. variable_request%store) then
1035             ! This variable needs to end up in the write cache
1036             ! Any contributions from tasks other than the last need to be saved in the store and loaded into the writ
     e cache by the last task.
1037             ! If the variable is not written by anyone, it needs to be preloaded into the write cache by the last tas
     k
1038             output_variable => variable_request%output_variable_set%first
1039             if (.not. associated(output_variable) .and. variable_request%variable%source /= source_constant) &
1040                call last_task%write_cache_preload%add(variable_request%variable)
1041             do while (associated(output_variable))
1042                responsible = associated(last_task)
1043                if (responsible) responsible = task_is_responsible(last_task, output_variable%p)
1044                if (.not. responsible) then
1045                   output_variable%p%copy_to_store = .true.
1046                   call last_task%write_cache_preload%add(output_variable%p%target)
1047                end if
1048                output_variable => output_variable%next
1049             end do
1050          end if
1051          call link_cowritten_outputs(variable_request%output_variable_set, last_task)
1052          variable_request => variable_request%next
1053       end do
1054
1055       self%state = job_state_finalized_prefill_settings
1056
1057    contains
1058
1059       subroutine link_cowritten_outputs(output_variable_set, requesting_task)
1060          type (type_output_variable_set), intent(in) :: output_variable_set
1061          type (type_task), pointer                   :: requesting_task
1062
1063          type type_variable_and_task
1064             type (type_output_variable), pointer :: output_variable
1065             type (type_task),            pointer :: task
1066          end type
1067
1068          type (type_output_variable_set_node), pointer :: output_variable
1069          logical                                       :: multiple_tasks
1070          type (type_task),                     pointer :: task
1071          integer                                       :: n
1072          type (type_variable_and_task), allocatable    :: variable_and_tasks(:)
1073          type (type_job_set)                           :: job_set
1074          class (type_job),                     pointer :: first_job
1075          type (type_task),                     pointer :: first_task
1076
1077          if (.not. associated(output_variable_set%first)) return
1078
1079          output_variable => output_variable_set%first
1080          task => find_responsible_task(self, output_variable%p)
1081          _ASSERT_(associated(task), 'job_finalize_prefill_settings', 'Task responsible for ' // trim(output_variable%
     p%target%name) // ' not found.')
1082          multiple_tasks = .false.
1083          output_variable => output_variable%next
1084          do while (associated(output_variable) .and. .not. multiple_tasks)
1085             if (.not. task_is_responsible(task, output_variable%p)) multiple_tasks = .true.
1086             output_variable => output_variable%next
1087          end do
1088
1089          ! If all outputs are written by the same task, it can handle its own initialization
1090          ! and there is no need to temporarily store results and reload them into the write cache.
1091          ! Thus we are done.
1092          if (.not. multiple_tasks) return
1093
1094          ! Build a list of output variables and responsible tasks
1095          n = 0
1096          output_variable => output_variable_set%first
1097          do while (associated(output_variable))
1098             n = n + 1
1099             output_variable => output_variable%next
1100          end do
1101          allocate(variable_and_tasks(n))
1102          n = 0
1103          output_variable => output_variable_set%first
1104          do while (associated(output_variable))
1105             task => find_responsible_task(self, output_variable%p)
1106             _ASSERT_(associated(task), 'job_finalize_prefill_settings', 'Task responsible for ' // trim(output_variab
```

```
le%p%target%name) // ' not found.')
            if (.not. associated(task, requesting_task)) output_variable%p%copy_to_store = .true.
            n = n + 1
            variable_and_tasks(n)%task => task
            variable_and_tasks(n)%output_variable => output_variable%p
            call job_set%add(task%job)
            output_variable => output_variable%next
         end do

         ! Find the first job (returns null if multiple jobs run in parallel)
         first_job => job_set%find_first()
         call job_set%finalize()

         ! Find the first task in the first job (if any)
         first_task => null()
         if (associated(first_job)) then
            first_task => first_job%first_task
            do while (associated(first_task))
               do n = 1, size(variable_and_tasks)
                  if (associated(first_task, variable_and_tasks(n)%task)) exit
               end do
               if (n <= size(variable_and_tasks)) exit
               first_task => first_task%next
            end do
            _ASSERT_(associated(first_task), 'link_cowritten_outputs', 'No contributing task found within first job.'
)
         end if

         do n = 1, size(variable_and_tasks)
            if (.not. associated(first_task, variable_and_tasks(n)%task)) &
               call variable_and_tasks(n)%task%write_cache_preload%add(variable_and_tasks(n)%output_variable%target)
            if (.not. associated(first_task))  call first_job%store_prefills%add(variable_and_tasks(n)%output_variabl
e%target)
         end do
      end subroutine

      subroutine prepare_task(task)
         type (type_task), pointer :: task

         integer                                      :: icall
         type (type_input_variable_set_node),  pointer :: input_variable
         type (type_output_variable_set_node), pointer :: output_variable
         type (type_output_variable),          pointer :: final_output_variable

         do icall = 1, size(task%calls)
            ! For all inputs that this call requires, determine whether they are produced by the same task
            ! (solved by copying between write and read cache) or by an earlier task (solved by temporary storing)
            input_variable => task%calls(icall)%graph_node%inputs%first
            do while (associated(input_variable))
               final_output_variable => null()
               output_variable => input_variable%p%sources%first
               do while (associated(output_variable))
                  if (task_is_responsible(task, output_variable%p) .and. input_variable%p%update) then
                     ! The call that is responsible for computing this input is part of the same task.
                     ! Therefore the output needs to be copied to the read cache.
                     ! But only if it is the last variable in this task contributing to this input.
                     if (.not. associated(final_output_variable)) then
                        final_output_variable => output_variable%p
                     elseif (output_is_produced_before(task, output_variable%p, final_output_variable)) then
                        final_output_variable => output_variable%p
                     end if
                  else
                     ! The call that is responsible for computing this input is part of another task.
                     ! Therefore the output needs to be copied to the persistent store.
                     output_variable%p%copy_to_store = .true.
                  end if
                  output_variable => output_variable%next
               end do
               if (associated(final_output_variable)) final_output_variable%copy_to_cache = .true.
               if (input_variable%p%update) call link_cowritten_outputs(input_variable%p%sources, task)
               input_variable => input_variable%next
            end do
         end do
      end subroutine prepare_task

   end subroutine job_finalize_prefill_settings

   subroutine job_set_add(self, job)
      class (type_job_set), intent(inout) :: self
      class (type_job), target            :: job

      type (type_job_node), pointer :: job_node
      integer,              pointer :: pmember

      job_node => self%first
      pmember => job%state
      do while (associated(job_node))
         ! Note: for Cray 10.0.4, the comparison below fails for class pointers! Therefore we compare type member ref
erences.
         if (associated(pmember, job_node%p%state)) return
         job_node => job_node%next
      end do
      allocate(job_node)
      job_node%p => job
      job_node%next => self%first
      self%first => job_node
   end subroutine

```

```
1201    function job_set_find_first(job_set) result(first)
1202       class (type_job_set), intent(in) :: job_set
1203       class (type_job), pointer :: first
1204
1205       type (type_job_node), pointer :: job_node
1206
1207       first => null()
1208
1209       job_node => job_set%first
1210       if (.not. associated(job_node)) return
1211       if (.not. associated(job_node%next)) then
1212          first => job_node%p
1213          return
1214       end if
1215       do while (associated(job_node))
1216          job_node%p%flag = .true.
1217          job_node => job_node%next
1218       end do
1219
1220       job_node => job_set%first
1221       do while (associated(job_node))
1222          if (.not. has_flagged_ancestor(job_node%p)) then
1223             if (associated(first)) then
1224                first => null()
1225                exit
1226             end if
1227             first => job_node%p
1228          end if
1229          job_node => job_node%next
1230       end do
1231
1232       job_node => job_set%first
1233       do while (associated(job_node))
1234          job_node%p%flag = .false.
1235          job_node => job_node%next
1236       end do
1237
1238    contains
1239
1240       recursive function has_flagged_ancestor(job) result(found)
1241          class (type_job), intent(in) :: job
1242          logical :: found
1243
1244          type (type_job_node), pointer :: job_node
1245
1246          found = .true.
1247          job_node => job%previous%first
1248          do while (associated(job_node))
1249             if (job_node%p%flag .or. has_flagged_ancestor(job_node%p)) return
1250             job_node => job_node%next
1251          end do
1252          found = .false.
1253       end function
1254
1255    end function
1256
1257    subroutine job_set_finalize(self)
1258       class (type_job_set), intent(inout) :: self
1259
1260       type (type_job_node), pointer :: job_node, next
1261
1262       job_node => self%first
1263       do while (associated(job_node))
1264          next => job_node%next
1265          deallocate(job_node)
1266          job_node => next
1267       end do
1268       self%first => null()
1269    end subroutine
1270
1271    subroutine job_initialize(self, variable_register, schedules)
1272       class (type_job), target,             intent(inout) :: self
1273       type (type_global_variable_register), intent(inout) :: variable_register
1274       type (type_schedules),                intent(inout) :: schedules
1275
1276       type (type_task), pointer :: task
1277
1278       _ASSERT_(self%state < job_state_initialized, 'job_initialize', trim(self%name) // ': this job has already been
    initialized.')
1279       _ASSERT_(self%state >= job_state_finalized_prefill_settings, 'job_initialize', 'Prefill settings for this job h
    ave not been finalized yet.')
1280
1281       if (associated(self%first_task)) call self%first_task%read_cache_preload%update(self%read_cache_loads)
1282
1283       ! Initialize tasks
1284       task => self%first_task
1285       do while (associated(task))
1286          call task%initialize(variable_register, schedules)
1287          task => task%next
1288       end do
1289
1290       self%state = job_state_initialized
1291
1292    end subroutine job_initialize
1293
1294    subroutine job_process_indices(self)
1295       class (type_job), target, intent(inout) :: self
1296
```

```
1297       type (type_task), pointer :: task
1298
1299       if (allocated(self%arg1_sources)) then
1300          _ASSERT_(all(self%arg1_sources > 0), 'job_process_indices', 'BUG: one or  more source indices for argument 1
     of job ' // trim(self%name) // ' are invalid.')
1301       end if
1302       if (allocated(self%arg2_sources)) then
1303          _ASSERT_(all(self%arg2_sources > 0), 'job_process_indices', 'BUG: one or  more source indices for argument 2
     of job ' // trim(self%name) // ' are invalid.')
1304       end if
1305
1306       task => self%first_task
1307       do while (associated(task))
1308          call task_process_indices(task)
1309          task => task%next
1310       end do
1311
1312       call gather_prefill(domain_interior, self%interior_store_prefill)
1313       call gather_prefill(domain_horizontal, self%horizontal_store_prefill)
1314
1315    contains
1316
1317       subroutine gather_prefill(domain, flags)
1318          integer, intent(in)  :: domain
1319          logical, allocatable :: flags(:)
1320
1321          type (type_variable_node), pointer :: variable_node
1322          integer :: prefill_max
1323
1324          prefill_max = 0
1325          variable_node => self%store_prefills%first
1326          do while (associated(variable_node))
1327             if (iand(variable_node%target%domain, domain) /= 0) prefill_max = max(prefill_max, variable_node%target%s
     tore_index)
1328             variable_node => variable_node%next
1329          end do
1330          allocate(flags(prefill_max))
1331          flags(:) = .false.
1332          variable_node => self%store_prefills%first
1333          do while (associated(variable_node))
1334             if (iand(variable_node%target%domain, domain) /= 0) flags(variable_node%target%store_index) = .true.
1335             variable_node => variable_node%next
1336          end do
1337       end subroutine
1338
1339    end subroutine job_process_indices
1340
1341    subroutine job_manager_create(self, job, name, source, outsource_tasks, previous)
1342       class (type_job_manager), intent(inout) :: self
1343       class (type_job), target, intent(inout) :: job
1344       character(len=*),         intent(in)    :: name
1345       integer, optional,        intent(in)    :: source
1346       logical, optional,        intent(in)    :: outsource_tasks
1347       class (type_job), target, optional      :: previous
1348
1349       type (type_job_node), pointer :: node
1350
1351       _ASSERT_(job%state < job_state_created, 'job_manager_create','This job has already been created with name ' //
     trim(job%name) // '.')
1352       job%state = job_state_created
1353
1354       job%name = name
1355       if (present(source)) then
1356          job%operation = source2operation(source)
1357          job%graph%operation = job%operation
1358       end if
1359       if (present(outsource_tasks)) job%outsource_tasks = outsource_tasks
1360
1361       allocate(node)
1362       node%p => job
1363       node%next => self%first
1364       self%first => node
1365
1366       if (present(previous)) call previous%connect(job)
1367    end subroutine job_manager_create
1368
1369    subroutine check_graph_duplicates(self)
1370       class (type_job_manager), intent(in) :: self
1371
1372       type (type_job_node),         pointer :: node
1373       type (type_node_list_member), pointer :: graph_node, graph_node2
1374       type (type_node_list)                 :: global_call_list
1375
1376       node => self%first
1377       do while (associated(node))
1378          graph_node => node%p%graph%first
1379          do while (associated(graph_node))
1380             graph_node2 => global_call_list%find_node(graph_node%p%model, graph_node%p%source)
1381             _ASSERT_(.not. associated(graph_node2), 'job_manager_initialize', 'Call ' // trim(graph_node%p%as_string(
     )) // ' appears multiple times in global graph.')
1382             call global_call_list%append(graph_node%p)
1383             graph_node => graph_node%next
1384          end do
1385          node => node%next
1386       end do
1387       call global_call_list%finalize()
1388    end subroutine check_graph_duplicates
1389
```

```
1390    subroutine job_manager_initialize(self, variable_register, schedules, log_unit, finalize_job)
1391       class (type_job_manager),             intent(inout) :: self
1392       type (type_global_variable_register), intent(inout) :: variable_register
1393       type (type_schedules),                intent(inout) :: schedules
1394       integer,                              intent(in)    :: log_unit
1395       type (type_job),                      intent(inout) :: finalize_job
1396
1397       type (type_job_node),                 pointer :: node, first_ordered
1398       type (type_node_list_member),         pointer :: graph_node
1399       type (type_input_variable_set_node),  pointer :: input_variable
1400
1401       ! Order jobs according to call order.
1402       ! This ensures that jobs that are scheduled to run earlier are also initialized earlier.
1403       ! Ðuring initialization, a job can therefore expect any preceding jobs to have initialized completely.
1404       first_ordered => null()
1405       do while (associated(self%first))
1406          node => self%first
1407          self%first => self%first%next
1408          call add_to_order(node%p)
1409          deallocate(node)
1410       end do
1411       self%first => first_ordered
1412
1413       ! Create all graphs. This must be done across all jobs before other operations that use graphs,
1414       ! since a job can add to graphs owned by other jobs.
1415       node => self%first
1416       do while (associated(node))
1417          call job_create_graph(node%p, variable_register)
1418          node => node%next
1419       end do
1420
1421 #ifndef NÐEBUG
1422       ! Ensure each call appears exactly once in the superset of all graphs
1423       call check_graph_duplicates(self)
1424 #endif
1425
1426       ! Create tasks. This must be done for all jobs before job_finalize_prefill_settings is called, as this API oper
     ates across all jobs.
1427       node => self%first
1428       do while (associated(node))
1429          call job_create_tasks(node%p, log_unit)
1430          node => node%next
1431       end do
1432
1433       ! Make sure all stale inputs are still calculated somewhere
1434       node => self%first
1435       do while (associated(node))
1436          graph_node => node%p%graph%first
1437          do while (associated(graph_node))
1438             input_variable => graph_node%p%inputs%first
1439             do while (associated(input_variable))
1440                if (.not. input_variable%p%update) then
1441                   call finalize_job%graph%add_variable(input_variable%p%target, input_variable%p%sources)
1442                end if
1443                input_variable => input_variable%next
1444             end do
1445             graph_node => graph_node%next
1446          end do
1447          node => node%next
1448       end do
1449
1450       ! Finalize prefill settings (this has cross-job implications, so must be done for all jobs before they initiali
     ze (initialization uses prefill settings)
1451       node => self%first
1452       do while (associated(node))
1453          call job_finalize_prefill_settings(node%p)
1454          node => node%next
1455       end do
1456
1457       ! Initialize all jobs. This add variables to the register (i.e., to the read and write caches and the persisten
     t store)
1458       node => self%first
1459       do while (associated(node))
1460          call job_initialize(node%p, variable_register, schedules)
1461          node => node%next
1462       end do
1463
1464       ! If we have unfulfilled dependneices, stop here and let the host/user deal with them.
1465       if (associated(variable_register%unfulfilled_dependencies%first)) return
1466
1467       variable_register%read_cache%frozen = .true.
1468       variable_register%write_cache%frozen = .true.
1469       variable_register%store%frozen = .true.
1470
1471       ! Create cache preload and copy instructions per task and call, and simultaneously check whether all dependenci
     es are fulfilled.
1472       ! This requires all indices (catalog/store/read cache/write cache) to be set.
1473       node => self%first
1474       do while (associated(node))
1475          call job_process_indices(node%p)
1476          node => node%next
1477       end do
1478
1479    contains
1480
1481       recursive subroutine add_to_order(job)
1482          class (type_job), target :: job
1483
```

```
1484          type (type_job_node), pointer :: node
1485          integer,              pointer :: pmember
1486
1487          ! Make sure job is not yet in list
1488          node => first_ordered
1489          pmember => job%state
1490          do while (associated(node))
1491             ! Note: for Cray 10.0.4, the comparison below fails for class pointers! Therefore we compare type member
     references.
1492             if (associated(pmember, node%p%state)) return
1493             node => node%next
1494          end do
1495
1496          ! Append any jobs that run earlier first
1497          node => job%previous%first
1498          do while (associated(node))
1499             call add_to_order(node%p)
1500             node => node%next
1501          end do
1502
1503          ! Append to list
1504          if (associated(first_ordered)) then
1505             node => first_ordered
1506             do while (associated(node%next))
1507                node => node%next
1508             end do
1509             allocate(node%next)
1510             node => node%next
1511          else
1512             allocate(first_ordered)
1513             node => first_ordered
1514          end if
1515          node%p => job
1516       end subroutine add_to_order
1517
1518    end subroutine job_manager_initialize
1519
1520    subroutine job_manager_print(self, unit, specific_variable)
1521       class (type_job_manager),                        intent(in) :: self
1522       integer,                                         intent(in) :: unit
1523       type (type_internal_variable), target, optional, intent(in) :: specific_variable
1524
1525       type (type_job_node), pointer :: node
1526
1527       node => self%first
1528       do while (associated(node))
1529          call node%p%print(unit, specific_variable)
1530          node => node%next
1531       end do
1532    end subroutine job_manager_print
1533
1534    subroutine job_manager_finalize(self)
1535       class (type_job_manager), intent(inout) :: self
1536
1537       type (type_job_node), pointer :: job_node
1538
1539       job_node => self%first
1540       do while (associated(job_node))
1541          call job_node%p%finalize()
1542          job_node => job_node%next
1543       end do
1544       call self%type_job_set%finalize()
1545    end subroutine
1546
1547    subroutine job_manager_write_graph(self, unit)
1548       class (type_job_manager), intent(in) :: self
1549       integer,                  intent(in) :: unit
1550
1551       type (type_job_node), pointer :: node
1552
1553       write (unit,'(A)') 'digraph {'
1554       node => self%first
1555       do while (associated(node))
1556          if (.true.) then
1557             call job_write_graph(node%p)
1558          else
1559             call node%p%graph%save_as_dot(unit, node%p%name)
1560          end if
1561          node => node%next
1562       end do
1563       write (unit,'(A)') '}'
1564
1565    contains
1566
1567       subroutine job_write_graph(job)
1568          class (type_job), intent(in) :: job
1569
1570          type (type_task),     pointer :: task, previous_task
1571          integer                       :: itask, icall
1572          character(len=8)              :: index
1573          type (type_job_node), pointer :: node
1574
1575          itask = 0
1576          previous_task => null()
1577          write (unit,'(A)') '  subgraph "cluster' // trim(job%name) // '" {'
1578          write (unit,'(A)') '    label="' // trim(job%name) // '";'
1579          task => job%first_task
1580          if (.not. associated(task)) write (unit,'(A)') '    "' // trim(job%name) // ':dummy" [style=invis];'
```

```
1581          do while (associated(task))
1582             itask = itask + 1
1583             write (index,'(i0)') itask
1584             write (unit,'(A)') '  subgraph "cluster' // trim(job%name) // ':' // trim(index) // '" {'
1585             write (unit,'(A)') '    label="' // trim(source2string(task%operation)) // '";style=filled;'
1586             write (unit,'(A)') '    node [color=black,style=filled];'
1587             do icall = 1, size(task%calls)
1588                write (unit,'(A)') '    ' // trim(task%calls(icall)%graph_node%as_dot()) // ';'
1589             end do
1590             if (size(task%calls) == 0) write (unit,'(A)') '    "' // trim(job%name) // ':' // index // ':dummy" [styl
     e=invis];'
1591             do icall = 2, size(task%calls)
1592                write (unit,'(A)') '    "' // trim(task%calls(icall - 1)%graph_node%as_string()) // '" -> "' // trim(t
     ask%calls(icall)%graph_node%as_string()) // '";'
1593             end do
1594             write (unit,'(A)') '  }'
1595             if (associated(previous_task)) write (unit,'(A)') '    ' // trim(get_endpoint_name(job, previous_task, .f
     alse.)) // ' -> ' // trim(get_endpoint_name(job, task, .true.)) // ';'
1596             previous_task => task
1597             task => task%next
1598          end do
1599          write (unit,'(A)') '  }'
1600
1601          node => job%previous%first
1602          do while (associated(node))
1603             task => node%p%first_task
1604             if (associated(task)) then
1605                do while (associated(task%next))
1606                   task => task%next
1607                end do
1608             end if
1609             write (unit,'(A)') '    ' // trim(get_endpoint_name(node%p, task, .false.)) // ' -> ' // trim(get_endpoin
     t_name(job, job%first_task, .true.)) // ';'
1610             node => node%next
1611          end do
1612       end subroutine
1613
1614       function get_endpoint_name(job, task, first) result(name)
1615          class (type_job),         intent(in) :: job
1616          type (type_task), pointer            :: task
1617          logical,                  intent(in) :: first
1618          character(len=attribute_length)      :: name
1619
1620          type (type_task), pointer :: ptask
1621          integer                   :: itask
1622          character(len=8)          :: index
1623
1624          if (.not. associated(task)) then
1625             name = '"' // trim(job%name) // ':dummy"'
1626          elseif (size(task%calls) == 0) then
1627             ! No calls in this task - we need to use a dummy node name.
1628             ! First find the index of the task within the job (that's part of dummy name)
1629             ptask => job%first_task
1630             itask = 1
1631             do while (.not. associated(ptask, task))
1632                itask = itask + 1
1633                ptask => ptask%next
1634             end do
1635             write (index,'(i0)') itask
1636             name = '"' // trim(job%name) // ':' // index // ':dummy"'
1637          elseif (first) then
1638             ! First call
1639             name = '"' // trim(task%calls(1)%graph_node%as_string()) // '"'
1640          else
1641             ! Last call
1642             name = '"' // trim(task%calls(size(task%calls))%graph_node%as_string()) // '"'
1643          end if
1644       end function
1645
1646    end subroutine
1647
1648    subroutine job_connect(self, next)
1649       class (type_job), intent(inout), target :: self
1650       class (type_job), intent(inout), target :: next
1651
1652       integer,              pointer :: pmember
1653       type (type_job_node), pointer :: node
1654
1655       _ASSERT_(self%state <= job_state_created, 'job_connect','This job (' // trim(self%name) // ') has already start
     ed initialization; it is too late to specify its place in the call order.')
1656       !_ASSERT_(.not. associated(self%previous), 'job_connect','This job ('//trim(self%name)//') has already been con
     nected to a subsequent one.')
1657
1658       ! Note: for Cray 10.0.4, the comparison below fails for class pointers! Therefore we compare type member refere
     nces.
1659       pmember => self%state
1660       _ASSERT_(.not. associated(pmember, next%state), 'job_connect', 'Attempt to connect job ' // trim(self%name) //
     ' to itself.')
1661
1662       allocate(node)
1663       node%p => self
1664       node%next => next%previous%first
1665       next%previous%first => node
1666       call self%graph%connect(next%graph)
1667    end subroutine job_connect
1668
1669    function variable_register_add(self, variable, share_constants) result(i)
1670       type (type_variable_register), intent(inout) :: self
```

```
      type (type_internal_variable), target        :: variable
      logical,                       intent(in)    :: share_constants
      integer :: i

      _ASSERT_(.not. self%frozen, 'variable_register_add', 'Cannot add ' // trim(variable%name) // '; register has been frozen.')
      select case (variable%domain)
      case (domain_interior)
         call add(self%interior)
      case (domain_horizontal, domain_surface, domain_bottom)
         call add(self%horizontal)
      case (domain_scalar)
         call add(self%scalar)
      end select

   contains

      subroutine add(list)
         type (type_variable_list), intent(inout) :: list

         type (type_variable_node), pointer :: node

         if (share_constants .and. variable%source == source_constant) then
            ! This is a constant. See if there is already another constant with the same value in the register.
            ! If there is, reuse that entry instead of creating a new one.
            i = 0
            node => list%first
            do while (associated(node))
               i = i + 1
               if (node%target%source == source_constant .and. node%target%prefill_value == variable%prefill_value) return
               node => node%next
            end do
         end if
         call list%append(variable, index=i)
      end subroutine

   end function variable_register_add

   subroutine variable_register_finalize(self)
      class (type_variable_register),intent(inout) :: self
      call self%interior%finalize()
      call self%horizontal%finalize()
      call self%scalar%finalize()
   end subroutine

   recursive subroutine global_variable_register_add_to_store(self, variable)
      class (type_global_variable_register), intent(inout) :: self
      type (type_internal_variable), target                :: variable

      type (type_variable_node), pointer :: variable_node

      ! If this variable has already been added to the persistent store, we are done: return.
      if (variable%store_index /= store_index_none) return

      ! If this variable is contributing to a variable (e.g., by adding to a sum), that other variable
      ! takes control. That controlling variable will then propagate its store index to all contributors.
      if (associated(variable%write_owner)) then
         call self%add_to_store(variable%write_owner)
         return
      end if

      ! Add the variable to the store and obtain its index.
      variable%store_index = variable_register_add(self%store, variable, share_constants=.true.)

      ! Propagate store index to any contributing variables.
      if (associated(variable%cowriters)) then
         variable_node => variable%cowriters%first
         do while (associated(variable_node))
            variable_node%target%store_index = variable%store_index
            variable_node => variable_node%next
         end do
      end if
   end subroutine global_variable_register_add_to_store

   recursive subroutine global_variable_register_add_to_read_cache(self, variable)
      class (type_global_variable_register), intent(inout) :: self
      type (type_internal_variable), target                :: variable

      integer :: index

      ! If this variable is required but has no data, register it as an unfulfilled dependency.
      if (variable%source == source_unknown .and. variable%presence /= presence_external_optional) &
         call self%unfulfilled_dependencies%add(variable)

      ! If this variable has no data or it has already been added to the read cache, we are done: return.
      if (variable%source == source_unknown .or. variable%read_indices%value /= -1) return

      ! NB line below commented out because the variables that contribute together to a "reduce" operation (e.g., summation)
      ! may be called in any order. Only the last one may have copy_to_cache set, and that last one is not necessarily the write_owner.
      !_ASSERT_(.not. associated(variable%write_owner), 'variable_register_add_read', 'called on variable with owner')

      ! Add the variable to the register and obtain its index.
      index = variable_register_add(self%read_cache, variable, share_constants=.true.)

```

```
      ! Assign the read index to the variable.
      call variable%read_indices%set_value(index)

      !variable_node => variable%cowriters%first
      !do while (associated(variable_node))
      !   call variable_node%target%read_indices%set_value(i)
      !   variable_node => variable_node%next
      !end do
   end subroutine global_variable_register_add_to_read_cache

   subroutine global_variable_register_add_to_catalog(self, variable)
      class (type_global_variable_register), intent(inout) :: self
      type (type_internal_variable), target                :: variable

      if (variable%catalog_index /= -1) return
      variable%catalog_index = variable_register_add(self%catalog, variable, share_constants=.false.)
   end subroutine global_variable_register_add_to_catalog

   recursive subroutine global_variable_register_add_to_write_cache(self, variable)
      class (type_global_variable_register), intent(inout) :: self
      type (type_internal_variable), target                :: variable

      integer :: index

      ! If this variable has already been added to the write cache, we are done: return.
      if (variable%write_indices%value /= -1) return

      ! Add the variable to the register and obtain its index.
      if (associated(variable%write_owner)) then
         ! This variable is contributing to a variable (e.g., by adding to a sum), that other variable
         ! takes control and determines the index
         call self%add_to_write_cache(variable%write_owner)
         index = variable%write_owner%write_indices%value
      else
         index = variable_register_add(self%write_cache, variable, share_constants=.true.)
      end if

      ! Assign the write index to the variable.
      call variable%write_indices%set_value(index)
   end subroutine global_variable_register_add_to_write_cache

   subroutine global_variable_register_print(self, unit)
      class (type_global_variable_register), intent(in) :: self
      integer,                               intent(in) :: unit

      call print_list('Interior catalog:', self%catalog%interior)
      call print_list('Horizontal catalog:', self%catalog%horizontal)
      call print_list('Scalar catalog:', self%catalog%scalar)
      call print_list('Interior store:', self%store%interior)
      call print_list('Horizontal store:', self%store%horizontal)
      call print_list('Interior read cache:', self%read_cache%interior)
      call print_list('Horizontal read cache:', self%read_cache%horizontal)
      call print_list('Scalar read cache:', self%read_cache%scalar)
      call print_list('Interior write cache:', self%write_cache%interior)
      call print_list('Horizontal write cache:', self%write_cache%horizontal)

   contains

      subroutine print_list(title, list)
         character(len=*),          intent(in) :: title
         type (type_variable_list), intent(in) :: list

         integer                              :: i
         type (type_variable_node), pointer :: variable_node

         write (unit,'(a)') title
         i = 0
         variable_node => list%first
         do while (associated(variable_node))
            i = i + 1
            write (unit,'("  ",i0,": ",a)') i, trim(variable_node%target%name)
            variable_node => variable_node%next
         end do
      end subroutine

   end subroutine global_variable_register_print

   subroutine global_variable_register_finalize(self)
      class (type_global_variable_register), intent(inout) :: self

      call self%catalog%finalize()
      call self%store%finalize()
      call self%read_cache%finalize()
      call self%write_cache%finalize()
      call self%unfulfilled_dependencies%finalize()
   end subroutine

end module fabm_job

!-----------------------------------------------------------------------
! Copyright under the GNU Public License - www.gnu.org
!-----------------------------------------------------------------------
```