```
#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

```

```fortran
   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 contribution 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 /= source_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 commands')
      _ASSERT_(n_hz == sum(self%calls%ncopy_hz), 'task_process_indices', 'mismatch in count of horizontal copy commands')
      _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 for 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 for 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.
      call create_prefill_commands(self%prefill,    domain_interior)
      call create_prefill_commands(self%prefill_hz, domain_horizontal)

      ! Create read cache load instructions for all input variables.
      call create_load_commands(self%load,        domain_interior)
      call create_load_commands(self%load_hz,     domain_horizontal)
      call create_load_commands(self%load_scalar, domain_scalar)

   contains

      integer function get_copy_command_count(call_node, domain)
         type (type_call), intent(in) :: call_node
         integer,          intent(in) :: domain

         type (type_output_variable_set_node), pointer :: variable
         integer                                       :: read_index

         get_copy_command_count = 0
         variable => call_node%graph_node%outputs%first
         do while (associated(variable))
            if (variable%p%copy_to_cache .and. iand(variable%p%target%domain, domain) /= 0) then
               _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.')
               read_index = variable%p%target%read_indices%value
               if (associated(variable%p%target%write_owner)) read_index = variable%p%target%write_owner%read_indices%value
               _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.')
               get_copy_command_count = get_copy_command_count + 1
            end if
            variable => variable%next
         end do
      end function

      subroutine create_cache_copy_commands(call_node, commands, domain, n)
         type (type_call),               intent(in)    :: call_node
         type (type_cache_copy_command), intent(inout) :: commands(:)
         integer,                        intent(in)    :: domain
         integer,                        intent(inout) :: n

         type (type_output_variable_set_node), pointer :: variable
         integer                                       :: max_write_index, read_index, write_index

         ! We will order by source index (i.e., the index in the write cache) to hopefully increase cache hits.
         ! To allow this ordering, first determine the maximum source index.
         max_write_index = -1
         variable => call_node%graph_node%outputs%first
         do while (associated(variable))
            if (variable%p%copy_to_cache .and. iand(variable%p%target%domain, domain) /= 0) then
               max_write_index = max(max_write_index, variable%p%target%write_indices%value)
            end if
            variable => variable%next
         end do

         ! Now process all possible source indices in order, and create cache copy commands where required.
         do write_index = 1, max_write_index
            variable => call_node%graph_node%outputs%first
            do while (associated(variable))
               if (variable%p%copy_to_cache .and. iand(variable%p%target%domain, domain) /= 0 .and. variable%p%target%write_indices%value == write_index) then
                  n = n + 1
                  read_index = variable%p%target%read_indices%value
                  if (associated(variable%p%target%write_owner)) read_index = variable%p%target%write_owner%read_indices%value
                  commands(n)%read_index = read_index
                  commands(n)%write_index = write_index
                  exit
               end if
               variable => variable%next
            end do
         end do
      end subroutine create_cache_copy_commands

      subroutine create_prefill_commands(prefill, domain)
         integer, intent(out), allocatable :: prefill(:)
         integer, intent(in)               :: domain

         integer                                       :: ilast
         integer                                       :: icall
         type (type_output_variable_set_node), pointer :: output_variable
         type (type_variable_node),            pointer :: variable_node

         ! Find the last write cache index
         ilast = 0
         do icall = 1, size(self%calls)
            output_variable => self%calls(icall)%graph_node%outputs%first
            do while (associated(output_variable))
               if (output_variable%p%target%prefill /= prefill_none .and. iand(output_variable%p%target%domain, domain) /= 0) then
                  _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.')

                  _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_index /= store_index_none), 'create_prefill_commands','Variable ' // trim(output_variable%p%target%name) // ' has prefill==previous value, but it does not have data.')
                  ilast = max(ilast, output_variable%p%target%write_indices%value)
               end if
               output_variable => output_variable%next
            end do
         end do
         variable_node => self%write_cache_preload%first
         do while (associated(variable_node))
            if (iand(variable_node%target%domain, domain) /= 0) then
               _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.')
               _ASSERT_(variable_node%target%source == source_external .or. variable_node%target%source == source_state .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.')
               ilast = max(ilast, variable_node%target%write_indices%value)
            end if
            variable_node => variable_node%next
         end do

         allocate(prefill(ilast))
         prefill(:) = prefill_none

         if (ilast == 0) return

         do icall = 1, size(self%calls)
            output_variable => self%calls(icall)%graph_node%outputs%first
            do while (associated(output_variable))
               if (output_variable%p%target%prefill /= prefill_none .and. iand(output_variable%p%target%domain, domain) /= 0) then
                  ilast = output_variable%p%target%write_indices%value
                  if (output_variable%p%target%prefill == prefill_previous_value) then
                     if (associated(output_variable%p%target%write_owner)) then
                        prefill(ilast) = output_variable%p%target%write_owner%catalog_index
                     else
                        prefill(ilast) = output_variable%p%target%catalog_index
                     end if
                  else
                     prefill(ilast) = output_variable%p%target%prefill
                  end if
               end if
               output_variable => output_variable%next
            end do
         end do
         variable_node => self%write_cache_preload%first
         do while (associated(variable_node))
            if (iand(variable_node%target%domain, domain) /= 0) then
               ilast = variable_node%target%write_indices%value
               if (associated(variable_node%target%write_owner)) then
                  prefill(ilast) = variable_node%target%write_owner%catalog_index
               else
                  prefill(ilast) = variable_node%target%catalog_index
               end if
            end if
            variable_node => variable_node%next
         end do
      end subroutine create_prefill_commands

      subroutine create_persistent_store_commands(commands, domain)
         integer, allocatable, intent(out) :: commands(:)
         integer,              intent(in)  :: domain

         integer                                       :: ilast
         integer                                       :: icall
         type (type_output_variable_set_node), pointer :: variable_node

         ! First find the last index in persistent storage that will be written to.
         ilast = 0
         do icall = 1, size(self%calls)
            variable_node => self%calls(icall)%graph_node%outputs%first
            do while (associated(variable_node))
               if (variable_node%p%copy_to_store .and. iand(variable_node%p%target%domain, domain) /= 0 .and. self%calls(icall)%graph_node%source /= source_constant) then
                  _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.')
                  _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 storage index.')
                  ilast = max(ilast,variable_node%p%target%store_index)
               end if
               variable_node => variable_node%next
            end do
         end do

         ! Allocate the commands array (go up to the last written-to index in persistent storage only)
         allocate(commands(ilast))
         commands(:) = 0

         ! Associate indices in persistent storage with the index in the write cache at which the source variable will be found.
         do icall = 1, size(self%calls)
            variable_node => self%calls(icall)%graph_node%outputs%first
            do while (associated(variable_node))
               if (variable_node%p%copy_to_store .and. iand(variable_node%p%target%domain,domain) /= 0 .and. self%calls(icall)%graph_node%source /= source_constant) &
                  commands(variable_node%p%target%store_index) = variable_node%p%target%write_indices%value
               variable_node => variable_node%next
            end do
         end do
      end subroutine create_persistent_store_commands

      subroutine create_load_commands(load, domain)
```

```
         integer, allocatable, intent(out) :: load(:)
         integer,              intent(in)  :: domain

         integer                            :: ilast
         type (type_variable_node), pointer :: input_variable

         ! First determine the maximum index to preload from/to. That determines the length of the array with preloading instructions.
         ilast = 0
         input_variable => self%read_cache_preload%first
         do while (associated(input_variable))
            _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.')
            _ASSERT_(input_variable%target%read_indices%value /= -1, 'create_load_commands', 'Variable ' // trim(input_variable%target%name) // ' is marked for preloading but has no valid read index.')
            _ASSERT_(input_variable%target%source == source_external .or. input_variable%target%source == source_state .or. input_variable%target%store_index /= -1, 'create_load_commands', 'Variable ' // trim(input_variable%target%name) // ' is marked for preloading but has no valid data.')
            if (iand(input_variable%target%domain, domain) /= 0) ilast = max(ilast, input_variable%target%read_indices%value)
            input_variable => input_variable%next
         end do

         ! Allocate array with preloading instructions, and initialize these to "do not preload"
         allocate(load(ilast))
         load(:) = 0

         ! Flag variables that require preloading
         input_variable => self%read_cache_preload%first
         do while (associated(input_variable))
            if (iand(input_variable%target%domain, domain) /= 0) load(input_variable%target%read_indices%value) = input_variable%target%catalog_index
            input_variable => input_variable%next
         end do
      end subroutine create_load_commands

   end subroutine task_process_indices

   subroutine job_print(self, unit, specific_variable)
      class (type_job),                                 intent(in) :: self
      integer,                                          intent(in) :: unit
      type (type_internal_variable), target, optional, intent(in) :: specific_variable

      type (type_task), pointer :: task

      write (unit,'(a,a)') 'Job: ',trim(self%name)
      task => self%first_task
      do while (associated(task))
         write (unit,'(a,a)') '- TASK WITH OPERATION = ',trim(source2string(task%operation))
         call task%print(unit, indent=2, specific_variable=specific_variable)
         task => task%next
      end do
   end subroutine job_print

   subroutine job_finalize(self)
      class (type_job), intent(inout) :: self

      type (type_task),             pointer :: task, next_task
      type (type_variable_request), pointer :: variable_request, next_variable_request
      type (type_call_request),     pointer :: call_request, next_call_request

      task => self%first_task
      do while (associated(task))
         next_task => task%next
         call task%finalize()
         deallocate(task)
         task => next_task
      end do
      self%first_task => null()

      variable_request => self%first_variable_request
      do while (associated(variable_request))
         next_variable_request => variable_request%next
         call variable_request%output_variable_set%finalize(owner=.false.)
         deallocate(variable_request)
         variable_request => next_variable_request
      end do
      self%first_variable_request => null()

      call_request => self%first_call_request
      do while (associated(call_request))
         next_call_request => call_request%next
         deallocate(call_request)
         call_request => next_call_request
      end do
      self%first_call_request => null()

      call self%read_cache_loads%finalize()
      call self%store_prefills%finalize()
      call self%previous%finalize()
      call self%graph%finalize()
   end subroutine job_finalize

   subroutine print_output_variable(variable, unit, indent)
      type (type_output_variable), intent(in) :: variable
      integer,                     intent(in) :: unit
      integer, optional,           intent(in) :: indent

      integer                             :: read_index
      type (type_node_set_member),pointer :: pnode

      read_index = variable%target%read_indices%value
      if (associated(variable%target%write_owner)) read_index = variable%target%write_owner%read_indices%value

      write (unit,'(a,"   - ",a,", write@",i0)',advance='no') repeat(' ',indent), trim(variable%target%name), variable%target%write_indices%value
      if (variable%copy_to_cache) write (unit,'(", cache@",i0)',advance='no') read_index
      if (variable%copy_to_store) write (unit,'(", store@",i0)',advance='no') variable%target%store_index
      write (unit,*)
      pnode => variable%dependent_nodes%first
      do while (associated(pnode))
         if (associated(pnode%p%model)) then
            write (unit,'(a,"     <- ",a,": ",a)') repeat(' ',indent), trim(pnode%p%model%get_path()), trim(source2string(pnode%p%source))
         else
            write (unit,'(a,"     <- host")') repeat(' ',indent)
         end if
         pnode => pnode%next
      end do
   end subroutine print_output_variable

   subroutine print_input_variable(variable, unit, indent)
      type (type_internal_variable), intent(in) :: variable
      integer,                       intent(in) :: unit
      integer,optional,              intent(in) :: indent

      write (unit,'(a,"   - ",a,", read@",i0)') repeat(' ',indent), trim(variable%name), variable%read_indices%value
   end subroutine print_input_variable

   subroutine task_print(self, unit, indent, specific_variable)
      class (type_task),                               intent(in) :: self
      integer,                                         intent(in) :: unit
      integer,optional,                                intent(in) :: indent
      type (type_internal_variable), target, optional, intent(in) :: specific_variable

      integer                                       :: indent_
      integer                                       :: i
      integer                                       :: icall
      type (type_output_variable_set_node), pointer :: output_variable
      logical                                       :: show
      type (type_input_variable_set_node),  pointer :: input_variable
      logical                                       :: header_written
      logical                                       :: subheader_written

      indent_ = 0
      if (present(indent)) indent_ = indent

      if (size(self%load) > 0 .or. size(self%load_hz) > 0 .or. size(self%load_scalar) > 0) write (unit,'(a,a)') repeat(' ', indent_), 'Read cache prefilling:'
      do i = 1, size(self%load)
         if (self%load(i) /= 0) write (unit,'(a,"  - interior[",i0,"]")') repeat(' ', indent_), i
      end do
      do i = 1, size(self%load_hz)
         if (self%load_hz(i) /= 0) write (unit,'(a,"  - horizontal[",i0,"]")') repeat(' ', indent_), i
      end do
      do i = 1, size(self%load_scalar)
         if (self%load_scalar(i) /= 0) write (unit,'(a,"  - scalar[",i0,"]")') repeat(' ', indent_), i
      end do

      if (size(self%prefill) > 0 .or. size(self%prefill_hz) > 0) write (unit,'(a,a)') repeat(' ', indent_), 'Write cache prefilling:'
      do i = 1, size(self%prefill)
         select case (self%prefill(i))
         case (prefill_none)
         case (prefill_constant)
            write (unit,'(a,"  - interior[",i0,"] = constant value")') repeat(' ', indent_), i
         case default
            write (unit,'(a,"  - interior[",i0,"] = previous value")') repeat(' ', indent_), i
         end select
      end do
      do i = 1, size(self%prefill_hz)
         select case (self%prefill_hz(i))
         case (prefill_none)
         case (prefill_constant)
            write (unit,'(a,"  - horizontal[",i0,"] = constant value")') repeat(' ', indent_), i
         case default
            write (unit,'(a,"  - horizontal[",i0,"] = previous value")') repeat(' ', indent_), i
         end select
      end do

      do icall = 1, size(self%calls)
         header_written = .false.
         if (.not. present(specific_variable)) call write_header()

         subheader_written = .false.
         input_variable => self%calls(icall)%graph_node%inputs%first
         do while (associated(input_variable))
            show = .true.
            if (present(specific_variable)) show = associated(input_variable%p%target, specific_variable)
            if (show) then
               call write_header()
               if (.not. subheader_written) then
                  write (unit,'(a,"   ",a)') repeat(' ', indent_), 'inputs:'
                  subheader_written = .true.
               end if
               call print_input_variable(input_variable%p%target, unit, indent_)
            end if
            input_variable => input_variable%next
         end do

         subheader_written = .false.
         output_variable => self%calls(icall)%graph_node%outputs%first
         do while (associated(output_variable))
            show = .true.
            if (present(specific_variable)) show = associated(output_variable%p%target, specific_variable)
            if (show) then
               call write_header()
               if (.not. subheader_written) then
                  write (unit,'(a,"   ",a)') repeat(' ', indent_), 'outputs:'
                  subheader_written = .true.
               end if
               call print_output_variable(output_variable%p, unit, indent_)
            end if
            output_variable => output_variable%next
         end do
      end do

   contains

      subroutine write_header()
         if (.not. header_written) then
            if (associated(self%calls(icall)%model)) then
               write (unit,'(a,a,": ",a)') repeat(' ', indent_), trim(self%calls(icall)%model%get_path()), trim(source2string(self%calls(icall)%source))
            else
               write (unit,'(a,"host")') repeat(' ', indent_)
            end if
            header_written = .true.
         end if
      end subroutine

   end subroutine task_print

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

      call self%read_cache_preload%finalize()
      call self%write_cache_preload%finalize()
   end subroutine task_finalize

   subroutine job_request_variable(self, variable, store)
      class (type_job),target,       intent(inout)         :: self
      type (type_internal_variable), intent(inout), target :: variable
      logical, optional,             intent(in)            :: store

      type (type_variable_request), pointer :: variable_request

      _ASSERT_(self%state >= job_state_created, 'job_request_variable', 'Job has not been created yet.')
      _ASSERT_(self%state <= job_state_created, 'job_request_variable', 'Job "' // trim(self%name) // '" has already begun initialization; variables can no longer be requested.')

      ! Make sure this variable will not be merged (thus variable request must be filed before starting to merge variables!)
      variable%write_operator = ior(variable%write_operator, operator_merge_forbidden)

      _ASSERT_(.not. associated(variable%write_owner), 'job_request_variable','BUG: requested variable is co-written.')

      allocate(variable_request)
      variable_request%variable => variable
      if (present(store)) variable_request%store = store
      variable_request%next => self%first_variable_request
      self%first_variable_request => variable_request
   end subroutine job_request_variable

   subroutine job_request_call(self, model, source)
      class (type_job),target,intent(inout) :: self
      class (type_base_model),intent(in),target :: model
      integer,                intent(in)    :: source

      type (type_call_request), pointer :: call_request

      _ASSERT_(self%state >= job_state_created, 'job_request_call', 'Job has not been created yet.')
      _ASSERT_(self%state <= job_state_created, 'job_request_call', 'Job "' // trim(self%name) // '" has already begun initialization; calls can no longer be requested.')

      if (.not. model%implements(source)) return
      allocate(call_request)
      call_request%model => model
      call_request%source = source
      call_request%next => self%first_call_request
      self%first_call_request => call_request
   end subroutine job_request_call

   subroutine job_create_graph(self, variable_register)
      class (type_job), target,             intent(inout) :: self
      type (type_global_variable_register), intent(inout) :: variable_register

      type (type_job_node),        pointer :: job_node
      type (type_variable_request),pointer :: variable_request
      type (type_call_request),    pointer :: call_request, next_call_request
      type (type_node),            pointer :: graph_node

      _ASSERT_(self%state >= job_state_created, 'job_create_graph', 'This job has not been created yet.')
      _ASSERT_(self%state < job_state_graph_created, 'job_create_graps', trim(self%name) // ': graph for this job has already been created.')

      ! If we are linked to an earlier called job, make sure its graph has been created already.
      ! This is essential because we can skip calls if they appear already in the previous job - we determine this by exploring its graph.
      job_node => self%previous%first
      do while (associated(job_node))
         _ASSERT_(job_node%p%state >= job_state_graph_created, 'job_create_graph', trim(self%name) // ': graph for previous job (' // trim(job_node%p%name) // ') has not been created yet.')
         job_node => job_node%next
      end do

      ! Construct the dependency graph by adding explicitly requested variables and calls.
      ! We clean up [deallocate] the variable and call requests at the same time.
      variable_request => self%first_variable_request
      do while (associated(variable_request))
         call self%graph%add_variable(variable_request%variable, variable_request%output_variable_set, copy_to_store=variable_request%store)
         if (variable_request%store) then
            ! FABM must be able to provide data for this variable across the entire spatial domain.
            ! If this variable is a constant, explicitly request a data field for it in the persistent store.
            ! If it is not a constant, the above call to add_variable will ensure that if the variable needs explicit computation,
            ! its value will also be copied to the persistent store.
            if (variable_request%variable%source == source_constant) call variable_register%add_to_store(variable_request%variable)
         else
            if (.not. associated(variable_request%output_variable_set%first)) call variable_register%add_to_write_cache(variable_request%variable)
         end if
         variable_request => variable_request%next
      end do

      call_request => self%first_call_request
      do while (associated(call_request))
         next_call_request => call_request%next
         graph_node => self%graph%add_call(call_request%model, call_request%source)
         deallocate(call_request)
         call_request => next_call_request
      end do
      self%first_call_request => null()

      !self%graph%frozen = .true.

      self%state = job_state_graph_created
   end subroutine job_create_graph

   subroutine job_create_tasks(self, log_unit)
      class (type_job), target, intent(inout) :: self
      integer,                  intent(in)    :: log_unit

      type (type_job_node),                  pointer :: job_node
      type (type_step), allocatable                  :: steps(:)
      integer                                        :: itask
      type (type_task),                      pointer :: task
      integer                                        :: ncall
      integer                                        :: icall
      type (type_graph_subset_node_pointer), pointer :: pnode

      _ASSERT_(self%state < job_state_tasks_created, 'job_create_tasks', trim(self%name) // ': tasks for this job have already been created.')
      _ASSERT_(self%state >= job_state_graph_created, 'job_create_tasks', trim(self%name) // ': the graph for this job have not been created yet.')

      ! If we are linked to an earlier called job, make sure its task list has already been created.
      ! This is essential if we will try to outsource our own calls to previous tasks/jobs.
      job_node => self%previous%first
      do while (associated(job_node))
         _ASSERT_(job_node%p%state >= job_state_tasks_created, 'job_create_tasks', trim(self%name) // ': tasks for previous job (' // trim(job_node%p%name) // ') have not been created yet.')
         job_node => job_node%next
      end do

      if (log_unit /= -1) write (log_unit,'(a)') trim(self%name)
      call find_best_order(self%graph, self%operation, log_unit, steps)

      ! Build task list by prepending
      do itask = size(steps), 1, -1
         ! Create the task and preprend it to the list.
         allocate(task)
         task%job => self
         task%next => self%first_task
         self%first_task => task
         task%operation = steps(itask)%operation

         ncall = 0
         pnode => steps(itask)%first
         do while (associated(pnode))
            ncall = ncall + 1
            pnode => pnode%next
         end do
         allocate(task%calls(ncall))

         ! Collect all calls for this task.
         ! Preserve the order in which calls appear in the "step",
         ! as this also represents the desired call order.
         pnode => steps(itask)%first
         do icall = 1, ncall
            task%calls(icall)%graph_node => pnode%p%graph_node
            task%calls(icall)%model      => pnode%p%graph_node%model
            task%calls(icall)%source     =  pnode%p%graph_node%source
            _ASSERT_(associated(task%calls(icall)%model), 'create_tasks', 'Call node does not have a model pointer.')
            _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.')
            pnode => pnode%next
         end do

         ! Clean-up array with processed calls.
         call steps(itask)%finalize()
      end do

      if (self%outsource_tasks) then
         task => self%first_task
         do while (associated(task))
            do icall = 1, size(task%calls)
               !call move_call_backwards(self, task, task%calls(icall))
            end do
            task => task%next
         end do
      end if

      if (associated(self%first_task)) then
         _ASSERT_(self%operation == source_unknown .or. .not. associated(self%first_task%next), 'job_select_order', 'Multiple tasks created while only one source was acceptable.')
      end if

      self%state = job_state_tasks_created

   !contains

      !subroutine move_call_backwards(job,task,call_node)
      !   class (type_job),target :: job
      !   type (type_task),target :: task
      !   type (type_call),target :: call_node
      !
      !   class (type_job),pointer :: current_job
      !   type (type_task),pointer :: current_task, target_task
      !   type (type_call),pointer :: current_call
      !   integer :: operation_after_merge
      !   logical :: compatible
      !
      !   write (*,*) 'moving '//trim(call_node%graph_node%as_string())
      !   current_job => job
      !   current_task => task
      !   target_task => null()
      !   compatible = .true.
      !   do while (move_one_step_backwards(current_job,current_task,call_node))
      !      operation_after_merge = current_task%operation
      !      compatible = is_source_compatible(operation_after_merge,call_node%source)
      !      compatible = compatible .and. operation_after_merge==current_task%operation
      !      if (compatible) then
      !         write (*,*) '  new task is compatible'
      !         target_task => current_task
      !      end if
      !   end do
      !   if (associated(target_task)) then
      !      ! Remove node from original task
      !      if (associated(task%first_call,call_node)) then
      !         task%first_call => call_node%next
      !      else
      !         current_call => task%first_call
      !         do while (.not.associated(current_call%next,call_node))
      !            current_call => current_call%next
      !         end do
      !         current_call%next => call_node%next
      !      end if
      !
      !      ! Append to target task
      !      current_call => target_task%first_call
      !      do while (associated(current_call%next))
      !         current_call => current_call%next
      !      end do
      !      current_call%next => call_node
      !      call_node%next => null()
      !
      !      !call target_task%initialize()
      !   end if
      !end subroutine move_call_backwards
      !
      !function move_one_step_backwards(job, task, travelling_call) result(moved)
      !   class (type_job),pointer :: job
      !   type (type_task),pointer :: task
      !   type (type_call),intent(in) :: travelling_call
      !   logical :: moved
      !
      !   type (type_call),pointer             :: call_node
      !   type (type_node_set_member), pointer :: dependency
      !
      !   moved = .false.
      !
      !   ! First determine if we can leave the current task
      !   ! (we cannot if it also handles one or more of our dependencies)
      !   call_node => task%first_call
```

```fortran
      !   do while (associated(call_node))
      !      dependency => travelling_call%graph_node%dependencies%first
      !      do while (associated(dependency))
      !         if (associated(dependency%p,call_node%graph_node)) then
      !            write (*,*) '  cannot move past '//trim(call_node%graph_node%as_string())
      !            return
      !         end if
      !         dependency => dependency%next
      !      end do
      !      call_node => call_node%next
      !   end do
      !
      !   ! Move to previous task (if any)
      !   call get_previous_task(job,task)
      !   if (.not.associated(task)) return
      !
      !   moved = .true.
      !end function move_one_step_backwards

   end subroutine job_create_tasks

   function get_last_task(job) result(task)
      class (type_job),intent(in) :: job
      type (type_task),pointer :: task

      task => job%first_task
      if (.not. associated(task)) return
      do while (associated(task%next))
         task => task%next
      end do
   end function get_last_task

   recursive function find_responsible_task(job, output_variable) result(task)
      class (type_job), intent(in)        :: job
      type (type_output_variable), target :: output_variable
      type (type_task), pointer :: task
      type (type_job_node), pointer :: job_node

      task => job%first_task
      do while (associated(task))
         if (task_is_responsible(task, output_variable)) return
         task => task%next
      end do

      job_node => job%previous%first
      do while (associated(job_node) .and. .not. associated(task))
         task => find_responsible_task(job_node%p, output_variable)
         job_node => job_node%next
      end do
   end function

   logical function task_is_responsible(task, output_variable)
      class (type_task), intent(in)       :: task
      type (type_output_variable), target :: output_variable

      integer                                      :: icall
      type (type_output_variable_set_node), pointer :: output_variable_node

      task_is_responsible = .true.
      do icall = 1, size(task%calls)
         ! Loop over all outputs of this call
         output_variable_node => task%calls(icall)%graph_node%outputs%first
         do while (associated(output_variable_node))
            if (associated(output_variable_node%p, output_variable)) return
            output_variable_node => output_variable_node%next
         end do
      end do
      task_is_responsible = .false.
   end function

   logical function output_is_produced_before(task, reference_output_variable, output_variable)
      class (type_task), intent(in)       :: task
      type (type_output_variable), target :: reference_output_variable, output_variable

      integer                                      :: icall
      type (type_output_variable_set_node), pointer :: output_variable_node

      output_is_produced_before = .false.
      do icall = 1, size(task%calls)
         ! Loop over all outputs of this call
         output_variable_node => task%calls(icall)%graph_node%outputs%first
         do while (associated(output_variable_node))
            if (associated(output_variable_node%p, output_variable)) output_is_produced_before = .true.
            if (associated(output_variable_node%p, reference_output_variable)) return
            output_variable_node => output_variable_node%next
         end do
      end do
      _ASSERT_(.false., 'output_is_produced_before', 'reference output not found in task')
   end function output_is_produced_before

   subroutine job_finalize_prefill_settings(self)
      class (type_job), target, intent(inout) :: self

      type (type_task),                    pointer :: task, last_task
      class (type_job),                    pointer :: first_job
      type (type_variable_request),        pointer :: variable_request
      type (type_output_variable_set_node), pointer :: output_variable
      logical                                      :: responsible

      _ASSERT_(self%state < job_state_finalized_prefill_settings, 'job_finalize_prefill_settings', 'This job has already been initialized.')
      _ASSERT_(self%state >= job_state_tasks_created, 'job_finalize_prefill_settings', 'Tasks for this job have not been created yet.')

      first_job => self
      do while (associated(first_job%previous%first))
         first_job => first_job%previous%first%p
      end do

      ! Set copy-to-cache and copy-to-store based on dependencies between different calls/tasks.
      task => self%first_task
      do while (associated(task))
         call prepare_task(task)
         task => task%next
      end do

      last_task => get_last_task(self)

      variable_request => self%first_variable_request
      do while (associated(variable_request))
         if (.not. variable_request%store) then
            ! This variable needs to end up in the write cache
            ! Any contributions from tasks other than the last need to be saved in the store and loaded into the write cache by the last task.
            ! If the variable is not written by anyone, it needs to be preloaded into the write cache by the last task
            output_variable => variable_request%output_variable_set%first
            if (.not. associated(output_variable) .and. variable_request%variable%source /= source_constant) &
               call last_task%write_cache_preload%add(variable_request%variable)
            do while (associated(output_variable))
               responsible = associated(last_task)
               if (responsible) responsible = task_is_responsible(last_task, output_variable%p)
               if (.not. responsible) then
                  output_variable%p%copy_to_store = .true.
                  call last_task%write_cache_preload%add(output_variable%p%target)
               end if
               output_variable => output_variable%next
            end do
         end if
         call link_cowritten_outputs(variable_request%output_variable_set, last_task)
         variable_request => variable_request%next
      end do

      self%state = job_state_finalized_prefill_settings

   contains

      subroutine link_cowritten_outputs(output_variable_set, requesting_task)
         type (type_output_variable_set), intent(in) :: output_variable_set
         type (type_task), pointer                   :: requesting_task

         type type_variable_and_task
            type (type_output_variable), pointer :: output_variable
            type (type_task),            pointer :: task
         end type

         type (type_output_variable_set_node), pointer :: output_variable
         logical                                       :: multiple_tasks
         type (type_task),                     pointer :: task
         integer                                       :: n
         type (type_variable_and_task), allocatable    :: variable_and_tasks(:)
         type (type_job_set)                           :: job_set
         class (type_job),                     pointer :: first_job
         type (type_task),                     pointer :: first_task

         if (.not. associated(output_variable_set%first)) return

         output_variable => output_variable_set%first
         task => find_responsible_task(self, output_variable%p)
         _ASSERT_(associated(task), 'job_finalize_prefill_settings', 'Task responsible for ' // trim(output_variable%p%target%name) // ' not found.')
         multiple_tasks = .false.
         output_variable => output_variable%next
         do while (associated(output_variable) .and. .not. multiple_tasks)
            if (.not. task_is_responsible(task, output_variable%p)) multiple_tasks = .true.
            output_variable => output_variable%next
         end do

         ! If all outputs are written by the same task, it can handle its own initialization
         ! and there is no need to temporarily store results and reload them into the write cache.
         ! Thus we are done.
         if (.not. multiple_tasks) return

         ! Build a list of output variables and responsible tasks
         n = 0
         output_variable => output_variable_set%first
         do while (associated(output_variable))
            n = n + 1
            output_variable => output_variable%next
         end do
         allocate(variable_and_tasks(n))
         n = 0
         output_variable => output_variable_set%first
         do while (associated(output_variable))
            task => find_responsible_task(self, output_variable%p)
            _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_variable%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 references.
         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

```

```fortran
   function job_set_find_first(job_set) result(first)
      class (type_job_set), intent(in) :: job_set
      class (type_job), pointer :: first

      type (type_job_node), pointer :: job_node

      first => null()

      job_node => job_set%first
      if (.not. associated(job_node)) return
      if (.not. associated(job_node%next)) then
         first => job_node%p
         return
      end if
      do while (associated(job_node))
         job_node%p%flag = .true.
         job_node => job_node%next
      end do

      job_node => job_set%first
      do while (associated(job_node))
         if (.not. has_flagged_ancestor(job_node%p)) then
            if (associated(first)) then
               first => null()
               exit
            end if
            first => job_node%p
         end if
         job_node => job_node%next
      end do

      job_node => job_set%first
      do while (associated(job_node))
         job_node%p%flag = .false.
         job_node => job_node%next
      end do

   contains

      recursive function has_flagged_ancestor(job) result(found)
         class (type_job), intent(in) :: job
         logical :: found

         type (type_job_node), pointer :: job_node

         found = .true.
         job_node => job%previous%first
         do while (associated(job_node))
            if (job_node%p%flag .or. has_flagged_ancestor(job_node%p)) return
            job_node => job_node%next
         end do
         found = .false.
      end function

   end function

   subroutine job_set_finalize(self)
      class (type_job_set), intent(inout) :: self

      type (type_job_node), pointer :: job_node, next

      job_node => self%first
      do while (associated(job_node))
         next => job_node%next
         deallocate(job_node)
         job_node => next
      end do
      self%first => null()
   end subroutine

   subroutine job_initialize(self, variable_register, schedules)
      class (type_job), target,             intent(inout) :: self
      type (type_global_variable_register), intent(inout) :: variable_register
      type (type_schedules),                intent(inout) :: schedules

      type (type_task), pointer :: task

      _ASSERT_(self%state < job_state_initialized, 'job_initialize', trim(self%name) // ': this job has already been initialized.')
      _ASSERT_(self%state >= job_state_finalized_prefill_settings, 'job_initialize', 'Prefill settings for this job have not been finalized yet.')

      if (associated(self%first_task)) call self%first_task%read_cache_preload%update(self%read_cache_loads)

      ! Initialize tasks
      task => self%first_task
      do while (associated(task))
         call task%initialize(variable_register, schedules)
         task => task%next
      end do

      self%state = job_state_initialized

   end subroutine job_initialize

   subroutine job_process_indices(self)
      class (type_job), target, intent(inout) :: self

      type (type_task), pointer :: task

      if (allocated(self%arg1_sources)) then
         _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.')
      end if
      if (allocated(self%arg2_sources)) then
         _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.')
      end if

      task => self%first_task
      do while (associated(task))
         call task_process_indices(task)
         task => task%next
      end do

      call gather_prefill(domain_interior, self%interior_store_prefill)
      call gather_prefill(domain_horizontal, self%horizontal_store_prefill)

   contains

      subroutine gather_prefill(domain, flags)
         integer, intent(in)  :: domain
         logical, allocatable :: flags(:)

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

         prefill_max = 0
         variable_node => self%store_prefills%first
         do while (associated(variable_node))
            if (iand(variable_node%target%domain, domain) /= 0) prefill_max = max(prefill_max, variable_node%target%store_index)
            variable_node => variable_node%next
         end do
         allocate(flags(prefill_max))
         flags(:) = .false.
         variable_node => self%store_prefills%first
         do while (associated(variable_node))
            if (iand(variable_node%target%domain, domain) /= 0) flags(variable_node%target%store_index) = .true.
            variable_node => variable_node%next
         end do
      end subroutine

   end subroutine job_process_indices

   subroutine job_manager_create(self, job, name, source, outsource_tasks, previous)
      class (type_job_manager), intent(inout) :: self
      class (type_job), target, intent(inout) :: job
      character(len=*),         intent(in)    :: name
      integer, optional,        intent(in)    :: source
      logical, optional,        intent(in)    :: outsource_tasks
      class (type_job), target, optional      :: previous

      type (type_job_node), pointer :: node

      _ASSERT_(job%state < job_state_created, 'job_manager_create','This job has already been created with name ' // trim(job%name) // '.')
      job%state = job_state_created

      job%name = name
      if (present(source)) then
         job%operation = source2operation(source)
         job%graph%operation = job%operation
      end if
      if (present(outsource_tasks)) job%outsource_tasks = outsource_tasks

      allocate(node)
      node%p => job
      node%next => self%first
      self%first => node

      if (present(previous)) call previous%connect(job)
   end subroutine job_manager_create

   subroutine check_graph_duplicates(self)
      class (type_job_manager), intent(in) :: self

      type (type_job_node),         pointer :: node
      type (type_node_list_member), pointer :: graph_node, graph_node2
      type (type_node_list)                 :: global_call_list

      node => self%first
      do while (associated(node))
         graph_node => node%p%graph%first
         do while (associated(graph_node))
            graph_node2 => global_call_list%find_node(graph_node%p%model, graph_node%p%source)
            _ASSERT_(.not. associated(graph_node2), 'job_manager_initialize', 'Call ' // trim(graph_node%p%as_string()) // ' appears multiple times in global graph.')
            call global_call_list%append(graph_node%p)
            graph_node => graph_node%next
         end do
         node => node%next
      end do
      call global_call_list%finalize()
   end subroutine check_graph_duplicates

   subroutine job_manager_initialize(self, variable_register, schedules, log_unit, finalize_job)
      class (type_job_manager),             intent(inout) :: self
      type (type_global_variable_register), intent(inout) :: variable_register
      type (type_schedules),                intent(inout) :: schedules
      integer,                              intent(in)    :: log_unit
      type (type_job),                      intent(inout) :: finalize_job

      type (type_job_node),                 pointer :: node, first_ordered
      type (type_node_list_member),         pointer :: graph_node
      type (type_input_variable_set_node),  pointer :: input_variable

      ! Order jobs according to call order.
      ! This ensures that jobs that are scheduled to run earlier are also initialized earlier.
      ! During initialization, a job can therefore expect any preceding jobs to have initialized completely.
      first_ordered => null()
      do while (associated(self%first))
         node => self%first
         self%first => self%first%next
         call add_to_order(node%p)
         deallocate(node)
      end do
      self%first => first_ordered

      ! Create all graphs. This must be done across all jobs before other operations that use graphs,
      ! since a job can add to graphs owned by other jobs.
      node => self%first
      do while (associated(node))
         call job_create_graph(node%p, variable_register)
         node => node%next
      end do

#ifndef NDEBUG
      ! Ensure each call appears exactly once in the superset of all graphs
      call check_graph_duplicates(self)
#endif

      ! Create tasks. This must be done for all jobs before job_finalize_prefill_settings is called, as this API operates across all jobs.
      node => self%first
      do while (associated(node))
         call job_create_tasks(node%p, log_unit)
         node => node%next
      end do

      ! Make sure all stale inputs are still calculated somewhere
      node => self%first
      do while (associated(node))
         graph_node => node%p%graph%first
         do while (associated(graph_node))
            input_variable => graph_node%p%inputs%first
            do while (associated(input_variable))
               if (.not. input_variable%p%update) then
                  call finalize_job%graph%add_variable(input_variable%p%target, input_variable%p%sources)
               end if
               input_variable => input_variable%next
            end do
            graph_node => graph_node%next
         end do
         node => node%next
      end do

      ! Finalize prefill settings (this has cross-job implications, so must be done for all jobs before they initialize (initialization uses prefill settings)
      node => self%first
      do while (associated(node))
         call job_finalize_prefill_settings(node%p)
         node => node%next
      end do

      ! Initialize all jobs. This add variables to the register (i.e., to the read and write caches and the persistent store)
      node => self%first
      do while (associated(node))
         call job_initialize(node%p, variable_register, schedules)
         node => node%next
      end do

      ! If we have unfulfilled dependneices, stop here and let the host/user deal with them.
      if (associated(variable_register%unfulfilled_dependencies%first)) return

      variable_register%read_cache%frozen = .true.
      variable_register%write_cache%frozen = .true.
      variable_register%store%frozen = .true.

      ! Create cache preload and copy instructions per task and call, and simultaneously check whether all dependencies are fulfilled.
      ! This requires all indices (catalog/store/read cache/write cache) to be set.
      node => self%first
      do while (associated(node))
         call job_process_indices(node%p)
         node => node%next
      end do

   contains

      recursive subroutine add_to_order(job)
         class (type_job), target :: job

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

         ! Make sure job is not yet in list
         node => first_ordered
         pmember => job%state
         do while (associated(node))
            ! Note: for Cray 10.0.4, the comparison below fails for class pointers! Therefore we compare type member references.
            if (associated(pmember, node%p%state)) return
            node => node%next
         end do

         ! Append any jobs that run earlier first
         node => job%previous%first
         do while (associated(node))
            call add_to_order(node%p)
            node => node%next
         end do

         ! Append to list
         if (associated(first_ordered)) then
            node => first_ordered
            do while (associated(node%next))
               node => node%next
            end do
            allocate(node%next)
            node => node%next
         else
            allocate(first_ordered)
            node => first_ordered
         end if
         node%p => job
      end subroutine add_to_order

   end subroutine job_manager_initialize

   subroutine job_manager_print(self, unit, specific_variable)
      class (type_job_manager),                        intent(in) :: self
      integer,                                         intent(in) :: unit
      type (type_internal_variable), target, optional, intent(in) :: specific_variable

      type (type_job_node), pointer :: node

      node => self%first
      do while (associated(node))
         call node%p%print(unit, specific_variable)
         node => node%next
      end do
   end subroutine job_manager_print

   subroutine job_manager_finalize(self)
      class (type_job_manager), intent(inout) :: self

      type (type_job_node), pointer :: job_node

      job_node => self%first
      do while (associated(job_node))
         call job_node%p%finalize()
         job_node => job_node%next
      end do
      call self%type_job_set%finalize()
   end subroutine

   subroutine job_manager_write_graph(self, unit)
      class (type_job_manager), intent(in) :: self
      integer,                  intent(in) :: unit

      type (type_job_node), pointer :: node

      write (unit,'(A)') 'digraph {'
      node => self%first
      do while (associated(node))
         if (.true.) then
            call job_write_graph(node%p)
         else
            call node%p%graph%save_as_dot(unit, node%p%name)
         end if
         node => node%next
      end do
      write (unit,'(A)') '}'

   contains

      subroutine job_write_graph(job)
         class (type_job), intent(in) :: job

         type (type_task),     pointer :: task, previous_task
         integer                       :: itask, icall
         character(len=8)              :: index
         type (type_job_node), pointer :: node

         itask = 0
         previous_task => null()
         write (unit,'(A)') '  subgraph "cluster' // trim(job%name) // '" {'
         write (unit,'(A)') '    label="' // trim(job%name) // '";'
         task => job%first_task
         if (.not. associated(task)) write (unit,'(A)') '    "' // trim(job%name) // ':dummy" [style=invis];'
         do while (associated(task))
            itask = itask + 1
            write (index,'(i0)') itask
            write (unit,'(A)') '  subgraph "cluster' // trim(job%name) // ':' // trim(index) // '" {'
            write (unit,'(A)') '    label="' // trim(source2string(task%operation)) // '";style=filled;'
            write (unit,'(A)') '    node [color=black,style=filled];'
            do icall = 1, size(task%calls)
               write (unit,'(A)') '    ' // trim(task%calls(icall)%graph_node%as_dot()) // ';'
            end do
            if (size(task%calls) == 0) write (unit,'(A)') '    "' // trim(job%name) // ':' // index // ':dummy" [style=invis];'
            do icall = 2, size(task%calls)
               write (unit,'(A)') '    "' // trim(task%calls(icall - 1)%graph_node%as_string()) // '" -> "' // trim(task%calls(icall)%graph_node%as_string()) // '";'
            end do
            write (unit,'(A)') '  }'
            if (associated(previous_task)) write (unit,'(A)') '    ' // trim(get_endpoint_name(job, previous_task, .false.)) // ' -> ' // trim(get_endpoint_name(job, task, .true.)) // ';'
            previous_task => task
            task => task%next
         end do
         write (unit,'(A)') '  }'

         node => job%previous%first
         do while (associated(node))
            task => node%p%first_task
            if (associated(task)) then
               do while (associated(task%next))
                  task => task%next
               end do
            end if
            write (unit,'(A)') '    ' // trim(get_endpoint_name(node%p, task, .false.)) // ' -> ' // trim(get_endpoint_name(job, job%first_task, .true.)) // ';'
            node => node%next
         end do
      end subroutine

      function get_endpoint_name(job, task, first) result(name)
         class (type_job),         intent(in) :: job
         type (type_task), pointer            :: task
         logical,                  intent(in) :: first
         character(len=attribute_length)      :: name

         type (type_task), pointer :: ptask
         integer                   :: itask
         character(len=8)          :: index

         if (.not. associated(task)) then
            name = '"' // trim(job%name) // ':dummy"'
         elseif (size(task%calls) == 0) then
            ! No calls in this task - we need to use a dummy node name.
            ! First find the index of the task within the job (that's part of dummy name)
            ptask => job%first_task
            itask = 1
            do while (.not. associated(ptask, task))
               itask = itask + 1
               ptask => ptask%next
            end do
            write (index,'(i0)') itask
            name = '"' // trim(job%name) // ':' // index // ':dummy"'
         elseif (first) then
            ! First call
            name = '"' // trim(task%calls(1)%graph_node%as_string()) // '"'
         else
            ! Last call
            name = '"' // trim(task%calls(size(task%calls))%graph_node%as_string()) // '"'
         end if
      end function

   end subroutine

   subroutine job_connect(self, next)
      class (type_job), intent(inout), target :: self
      class (type_job), intent(inout), target :: next

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

      _ASSERT_(self%state <= job_state_created, 'job_connect','This job (' // trim(self%name) // ') has already started initialization; it is too late to specify its place in the call order.')
      !_ASSERT_(.not. associated(self%previous), 'job_connect','This job ('//trim(self%name)//') has already been connected to a subsequent one.')

      ! Note: for Cray 10.0.4, the comparison below fails for class pointers! Therefore we compare type member references.
      pmember => self%state
      _ASSERT_(.not. associated(pmember, next%state), 'job_connect', 'Attempt to connect job ' // trim(self%name) // ' to itself.')

      allocate(node)
      node%p => self
      node%next => next%previous%first
      next%previous%first => node
      call self%graph%connect(next%graph)
   end subroutine job_connect

   function variable_register_add(self, variable, share_constants) result(i)
      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.)

```

```fortran
      ! 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
!-----------------------------------------------------------------------
```