OpenMP Technical Report 5: Memory Management Support for OpenMP 5.0

This Technical Report augments the OpenMP TR 4 document with language features for managing memory on systems with heterogeneous memories.

EDITORS

Alejandro Duran (alejandro.duran@intel.com)

Christian Terboven (terboven@itc.rwth-aachen.de)

OTHER AUTHORS

Jonathan Beard (ARM)

Bronis de Supinski (LLNL)

Deepak Eachempati (Cray)

Alexandre Eichenberger (IBM)

Ian Karlin (LLNL)

Kelvin Li (IBM)

Stephen Olivier (SNL)

Ravi Narayanaswamy (Intel)

John Pennycook (Intel)

Alejandro Rico (ARM)

Jeff Sandoval (Cray)

Tom Scogland (LLNL)

Jason Sewall (Intel)

Xinmin Tian (Intel)

and other members of the Affinity Subcommittee

January 5th, 2017

Expires January 4th, 2018

We actively solicit comments. Please provide feedback on this document either to the Editor directly or in the OpenMP Forum at openmp.org

End of Public Comment Period: March 5th, 2017

OpenMP Architecture Review Board
www.openmp.org info@openmp.org
OpenMP ARB – Ravi S. Rao, c/o Intel Corporation, 1300 S MoPac Express Way, Austin, TX 78746, USA

This technical report describes possible future directions or extensions to the OpenMP Specification.

The goal of this technical report is to build more widespread existing practice for an expanded OpenMP. It gives advice on extensions or future directions to those vendors who wish to provide them possibly for trial implementation, allows OpenMP to gather early feedback, support timing and scheduling differences between official OpenMP releases, and offers a preview to users of the future directions of OpenMP with the provision stated in the next paragraph.

This technical report is non-normative. Some of the components in this technical report may be considered for standardization in a future version of OpenMP, but they are not currently part of any OpenMP Specification. Some of the components in this technical report may never be standardized, others may be standardized in a substantially changed form, or it may be standardized as is in its entirety.

Memory Management support for OpenMP

The OpenMP Affinity Subcommittee

1 Motivation and Background

System performance is often dependent on memory performance. Over the past decades the bandwidth of the standard memory technology (DRAM) has scaled slower than the increase in CPU computational throughput. System builders traditionally addressed this problem by adding more memory channels to maintain system balance. However, recently, bandwidth and capacity are scaling slower than compute and vendors have not been able to maintain system balance with DRAM-only solutions. To address this problem, emerging systems feature multiple types of memories with different optimization points. Examples are systems that combine off-package DRAM with higher bandwidth technologies integrated on package to increase memory bandwidth, non-volatile high-density memories to increase capacity, and on-chip scratchpad memories with low-latency access.

Compute systems with such a tiered memory solution present a unique challenge to programmers. With the fastest resources typically having limited capacity, placement choices present performance tradeoffs in applications. Also, traditional first touch placement strategies used in Linux do not allow users to differentiate among memories with different properties. Vendors provide their own programming approaches to differentiate different memories, e.g., CUDA and Memkind, but these approaches and the alternative low-level programming approaches are non-portable. In response, to enable portability across platforms, the OpenMP committee is developing a more consistent and portable interface for memory placement in tiered memory systems.

The proposal in this document is designed to abstract the myriad of choices from the user. The goal is to enable portability, while providing the user with enough control to allow a runtime to manage allocations for user-defined properties such as latency, bandwidth and capacity. We aim to use properties and traits rather than the specific memory types of today to help future proof the interface against emerging and changing technology trends.

This document represents current directions being discussed within the OpenMP Affinity Subcommittee and is designed to engage the community, solicit feedback and reflect the current thoughts of the committee on this topic. The proposal is a start of a larger document that will include controls to cover additional memory types and features, such as persistent memory and constant memory. This document is not a promise that the interface will be adopted into the specification. Instead, it represents the subcommittee's best estimate of a portion of an interface that will be adopted, assuming that the OpenMP community agrees that the interface can be extended to fully support the range of architectures of interest.

2 High-level overview

A platform-agnostic integration of memory management support into OpenMP is necessary to avoid the separation of code paths for different platforms and also different kinds of memory within each platform. As a de-facto standard, OpenMP has to support all current kinds of memory and has to be capable of supporting future memory kinds and platform configuration without significant changes to both the specification and any code using the OpenMP memory management. This is achieved by introducing the following new concepts into the OpenMP API:

- Memory spaces and allocators: A *memory space* refers to a memory resource available in the system at the time the OpenMP program is executed. Each space has certain characteristics depending on the kind of the physical memory and the current state of the system. An *allocator* is an object that allocates (and frees) memory from an associated memory space.
- Memory allocation API: The `omp_alloc()` and `omp_free()` API routines are provided for C/C++ to allocate and deallocate memory using an allocator.
- `allocate` directive and clause: The new `allocate` directive and clause allow the allocation of variables without the explicit use of the aforementioned API, and can be used in both Fortran and C/C++. They support several modifiers to influence their behavior.

In order to work with memory spaces and allocators, an API is provided to manage (i.e., create and destroy) both types of objects. The programmer must explicitly use this API to enable the use of memory types other than the default type with OpenMP.

The mixture of run-time and compile-time functionality is necessary to handle the different types of memory allocations, namely a `malloc()`-like interface for dynamic (heap) allocations in C/C++ and directives for static and stack allocations in both Fortran and C/C++. A mixture of runtime and compiler support is also necessary to support certain kinds of memory that need special (machine) instructions to access or modify data.

With respect to future architectural developments, it must be assumed that hardware will develop at a faster rate than the OpenMP specification can match. In consequence, the options to express certain memory properties are not tied to current systems. Instead, the options aim to be broadly applicable by referring to certain characteristics of memory resources, and they are intended to be extended by vendors with the introduction of additional traits.

2.1 Memory spaces and allocators

A *memory space* represents a storage resource that is available in the system. For example, almost all contemporary HPC systems contain a DDR-based main memory, which could be the only available memory space. Additional new memory types include those with enhanced performance (e.g., high-bandwidth memory) or functionality (e.g., non-volatile memory).

Both could be additional memory spaces in a single system, and numerous combinations are possible.

A memory space is represented by the `omp_memspace_t` C/C++ dataype (`omp_memspace_kind` in Fortran). Before first use, it has to be initialized via the corresponding initialization function `omp_init_memspace`, which accepts a set of memory traits (see next paragraph) as the argument. The instance of a memory space is itself passed as an argument in the construction of an allocator. After last use, the memory space must be destroyed via `omp_destroy_memspace`.

Memory traits describe the characteristics of memory spaces and as such allow for queries, identification and description of the different memory spaces of a system. This proposal contains a base set of memory traits described below, others may be added in the future or as vendor-specific extensions. Memory traits can either be prescriptive, meaning an exact match is required, or descriptive, meaning the runtime is requested to select the optimal type of memory based on the requested properties.

Prescriptive traits include the location of memory (with possible values core, socket or device), a certain optimization characteristic of the underlying memory technology (with possible values bandwidth or latency or capacity), and support for certain page sizes or read/write permission. Descriptive traits include the relative distance relative to the task performing the request (with possible values near or far) and the relative bandwidth and latency of the memory space with respect to other memories in the system (with possible values highest and lowest).

A memory trait is represented by the `omp_memtrait_t` datatype and support for sets of memory traits is represented by `omp_memtrait_set_t` in C/C++, with corresponding Fortran types/kinds. The `omp_init_memtrait_set` API routine is available to construct a memory trait set from a given list of memory traits. The trait set is used as an argument to `omp_init_memspace`, with `omp_default_memtraits` representing the default memory as selected by the runtime. Traits to request a minimum total capacity and available capacity are also available. Associated routines include `omp_destroy_memtrait_set`, to destroy the memtrait set, `omp_add_memtraits`, to add a memory trait to a memory traits set, and `omp_merge_memtraits`, to merge two memory traits sets.

An *allocator* is an object performing allocations of contiguous memory chunks from a given memory space. *Allocator traits* can be employed to customize the behavior of an allocator. This includes the behavior in case the allocation is not successful – the standard behavior in case of failure is to fall back to the default memory, based on the `omp_default_memtraits` specified at initialization of the memory space. On many systems that would be DDR main memory. Further allocator traits specify the thread model (with possible values shared or exclusive) and the options to specify alignment and the request for pinned memory.

An allocator is represented by the `omp_allocator_t` datatype (`omp_allocator_kind` in Fortran). Before first use, it has to be initialized via the corresponding initialization function `omp_init_allocator`, which accepts a memory space and a set of allocator traits as arguments. API routines for the management of allocator traits are similar to those for memory traits. After last use, the allocator has to be destroyed via `omp_destroy_allocator`.

2.2 Memory allocation API for C/C++

Two new API routines are provided to allocate and deallocate memory using an allocator in C/C++. Allocations are performed with the `omp_alloc` routine, which takes the requested size as the first argument and an OpenMP allocator as the second argument and returns a pointer to the allocated memory. The additional `omp_alloc_safe_align` routine requests an aligned allocation. Similarly, the `omp_free` routine frees memory and also takes an OpenMP allocator as the second argument. When memory of a given size is requested, memory of at least that size is allocated, and it must be freed with the corresponding function using the corresponding allocator.

The separation of the API and the allocators allows the programmer to write portable code because only the allocator definition must be modified when the code is changed to target a different kind of memory on a different platform, while all the individual allocations in the code can remain unmodified.

2.3 Allocate directive and clause

The new `allocate` directive enables the programmer to influence the allocation of variables without the explicit use of the aforementioned API. It also integrates the memory management concept with the other directives and constructs in the OpenMP API. The effect of using the `allocate` directive is that for all variables in the list the storage location is determined by the application of the given allocator object. The allocator can be specified via the `allocator` clause. If no allocator is given, an implicit allocator is constructed from the memory and allocator traits specified with the directive via the `memtraits` and `alloctraits` clauses, taking as arguments the corresponding trait sets as discussed above.

In Fortran, the `allocate` directive provides in addition to the semantics described above the ability to use the allocator functionality with variables declared as `ALLOCATABLE`. That means it ensures the following Fortran `ALLOCATE` statement is performed with the OpenMP allocator specified either explicitly or constructed implicitly from the provided trait sets.

For directives supporting the new `allocate` clause, it specifies the allocation and memory traits of the storage used for private variables of a directive.

2.4 Default allocator

The new def-allocator-var ICV determines the allocator to be used by allocation routines, directives and clauses when an allocator is not specified by the user. The new corresponding API routines `omp_get_default_allocator` and `omp_set_default_allocator` are introduced, along with the new environment variable `OMP_ALLOCATOR`.

3 Changes to the OpenMP specification

In this section we present the necessary changes to be enacted to OpenMP TR4 document to enable our proposal. The new text that would be added is marked in blue and to simplify the presentation of the changes pages where the only changes are cross-references are not showed in this document.

3.1 Changes to Chapter 1

A private variable in a task region that eventually generates an inner nested **parallel** region is permitted to be made shared by implicit tasks in the inner **parallel** region. A private variable in a task region can be shared by an explicit task region generated during its execution. However, it is the programmer's responsibility to ensure through synchronization that the lifetime of the variable does not end before completion of the explicit task region sharing it. Any other access by one task to the private variables of another task results in unspecified behavior.

1.4.2 Device Data Environments

When an OpenMP program begins, an implicit **target data** region for each device surrounds the whole program. Each device has a device data environment that is defined by its implicit **target data** region. Any **declare target** directives and the directives that accept data-mapping attribute clauses determine how an original variable in a data environment is mapped to a corresponding variable in a device data environment.

When an original variable is mapped to a device data environment and the associated corresponding variable is not present in the device data environment, a new corresponding variable (of the same type and size as the original variable) is created in the device data environment. The initial value of the new corresponding variable is determined from the clauses and the data environment of the encountering thread.

The corresponding variable in the device data environment may share storage with the original variable. Writes to the corresponding variable may alter the value of the original variable. The impact of this on memory consistency is discussed in Section 1.4.5 on page 21. When a task executes in the context of a device data environment, references to the original variable refer to the corresponding variable in the device data environment.

The relationship between the value of the original variable and the initial or final value of the corresponding variable depends on the *map-type*. Details of this issue, as well as other issues with mapping a variable, are provided in Section 2.17.6.1 on page 256.

The original variable in a data environment and the corresponding variable(s) in one or more device data environments may share storage. Without intervening synchronization data races can occur.

1.4.3 Memory management

The host device, and target devices that an implementation may support, have attached storage resources where program variables are stored. These resources can be of different kinds and of different traits. A memory space in an OpenMP program represents one of these resources. Memory spaces have different traits that define them and a single resource may be exposed as multiple memory spaces with different traits. In any device at least one memory space is guaranteed to exist.

An OpenMP program can use an allocator to allocate storage for its variables. Allocators are associated with a memory space when created and use storage in that memory space to allocate variables. Allocators are also used to deallocate variables and free the storage in the memory space. When an OpenMP allocator is not used variables can be allocated in any memory space. The behavior of a memory management construct, modifier or API is unspecified if the variable that is applied to was not allocated with an OpenMP allocator.

1.4.4 The Flush Operation

The memory model has relaxed-consistency because a thread's temporary view of memory is not required to be consistent with memory at all times. A value written to a variable can remain in the thread's temporary view until it is forced to memory at a later time. Likewise, a read from a variable may retrieve the value from the thread's temporary view, unless it is forced to read from memory. The OpenMP flush operation enforces consistency between the temporary view and memory.

The flush operation is applied to a set of variables called the *flush-set*. The flush operation restricts reordering of memory operations that an implementation might otherwise do. Implementations must not reorder the code for a memory operation for a given variable, or the code for a flush operation for the variable, with respect to a flush operation that refers to the same variable.

If a thread has performed a write to its temporary view of a shared variable since its last flush of that variable, then when it executes another flush of the variable, the flush does not complete until the value of the variable has been written to the variable in memory. If a thread performs multiple writes to the same variable between two flushes of that variable, the flush ensures that the value of the last write is written to the variable in memory. A flush of a variable executed by a thread also causes its temporary view of the variable to be discarded, so that if its next memory operation for that variable is a read, then the thread will read from memory when it may again capture the value in the temporary view. When a thread executes a flush, no later memory operation by that thread for a variable involved in that flush is allowed to start until the flush completes. The completion of a flush of a set of variables executed by a thread is defined as the point at which all writes to those variables performed by the thread before the flush are visible in memory to all other threads and that thread's temporary view of all variables involved is discarded.

The flush operation provides a guarantee of consistency between a thread's temporary view and memory. Therefore, the flush operation can be used to guarantee that a value written to a variable by one thread may be read by a second thread. To accomplish this, the programmer must ensure that the second thread has not written to the variable since its last flush of the variable, and that the following sequence of events happens in the specified order:

1. The value is written to the variable by the first thread.

3.2 Changes to Chapter 2

Fortran

A *list item* is a variable, array section or common block name (enclosed in slashes). An *extended list item* is a *list item* or a procedure name. A *locator list item* is a *list item*.

When a named common block appears in a *list*, it has the same meaning as if every explicit member of the common block appeared in the list. An explicit member of a common block is a variable that is named in a **COMMON** statement that specifies the common block name and is declared in the same scoping unit in which the clause appears.

Although variables in common blocks can be accessed by use association or host association, common block names cannot. As a result, a common block name specified in a data-sharing attribute, a data copying or a data-mapping attribute clause must be declared to be a common block in the same scoping unit in which the clause appears.

Fortran

For all base languages, a *list item* or an *extended list item* is subject to the restrictions specified in Section 2.4 on page 48 and in each of the sections describing clauses and directives for which the *list* or *extended-list* appears.

The clauses of the **allocate** directive accept a key-value list. A key-value list is a comma-separated list of key-value pairs. A key-value pair has the form of key=value. The allowed keys and values depend on each clause.

3.2.1 Changes to ICVs descriptions

- *bind-var* - controls the binding of OpenMP threads to places. When binding is requested, the variable indicates that the execution environment is advised not to move threads between places. The variable can also provide default thread affinity policies. There is one copy of this ICV per data environment.

The following ICVs store values that affect the operation of loop regions.

- *run-sched-var* - controls the schedule that the **`runtime`** schedule clause uses for loop regions. There is one copy of this ICV per data environment.
- *def-sched-var* - controls the implementation defined default scheduling of loop regions. There is one copy of this ICV per device.

The following ICVs store values that affect program execution.

- *stacksize-var* - controls the stack size for threads that the OpenMP implementation creates. There is one copy of this ICV per device.
- *wait-policy-var* - controls the desired behavior of waiting threads. There is one copy of this ICV per device.
- *cancel-var* - controls the desired behavior of the **`cancel`** construct and cancellation points. There is one copy of this ICV for the whole program.
- *default-device-var* - controls the default target device. There is one copy of this ICV per data environment.
- *max-task-priority-var* - controls the maximum priority value that can be specified in the **`priority`** clause of the **`task`** construct. There is one copy of this ICV for the whole program.

The following ICVs store values that affect the operation of the tool interface.

- *tool-var* - determines whether an OpenMP implementation will try to register a tool. There is one copy of this ICV for the whole program.
- *tool-libraries-var* - specifies a list of absolute paths to tool libraries for OpenMP devices. There is one copy of this ICV for the whole program.

The following ICVs store values that affect default memory allocation.

- def-allocator-var - determines the allocator to be used by allocation routines, directives and clauses when an allocator is not specified by the user.

2.3.2 ICV Initialization

Table 2.1 shows the ICVs, associated environment variables, and initial values.

TABLE 2.1: ICV Initial Values

ICV	Environment Variable	Initial value
dyn-var	**OMP_DYNAMIC**	See description below
nest-var	**OMP_NESTED**	*false*
nthreads-var	**OMP_NUM_THREADS**	Implementation defined
run-sched-var	**OMP_SCHEDULE**	Implementation defined
def-sched-var	(none)	Implementation defined
bind-var	**OMP_PROC_BIND**	Implementation defined
stacksize-var	**OMP_STACKSIZE**	Implementation defined
wait-policy-var	**OMP_WAIT_POLICY**	Implementation defined
thread-limit-var	**OMP_THREAD_LIMIT**	Implementation defined
max-active-levels-var	**OMP_MAX_ACTIVE_LEVELS**	See description below
active-levels-var	(none)	*zero*
levels-var	(none)	*zero*
place-partition-var	**OMP_PLACES**	Implementation defined
cancel-var	**OMP_CANCELLATION**	*false*
default-device-var	**OMP_DEFAULT_DEVICE**	Implementation defined
max-task-priority-var	**OMP_MAX_TASK_PRIORITY**	*zero*
tool-var	**OMP_TOOL**	*enabled*
tool-libraries-var	**OMP_TOOL_LIBRARIES**	*empty string*
def-allocator-var	**OMP_ALLOCATOR**	Implementation defined

Description

- Each device has its own ICVs.
- The value of the *nthreads-var* ICV is a list.
- The value of the *bind-var* ICV is a list.
- The initial value of *dyn-var* is implementation defined if the implementation supports dynamic adjustment of the number of threads; otherwise, the initial value is *false*.

TABLE 2.2: Ways to Modify and to Retrieve ICV Values

ICV	Ways to modify value	Ways to retrieve value
dyn-var	**omp_set_dynamic()**	**omp_get_dynamic()**
nest-var	**omp_set_nested()**	**omp_get_nested()**
nthreads-var	**omp_set_num_threads()**	**omp_get_max_threads()**
run-sched-var	**omp_set_schedule()**	**omp_get_schedule()**
def-sched-var	(none)	(none)
bind-var	(none)	**omp_get_proc_bind()**
stacksize-var	(none)	(none)
wait-policy-var	(none)	(none)
thread-limit-var	**thread_limit** clause	**omp_get_thread_limit()**
max-active-levels-var	**omp_set_max_active_levels()**	**omp_get_max_active_levels()**
active-levels-var	(none)	**omp_get_active_level()**
levels-var	(none)	**omp_get_level()**
place-partition-var	(none)	See description below
cancel-var	(none)	**omp_get_cancellation()**
default-device-var	**omp_set_default_device()**	**omp_get_default_device()**
max-task-priority-var	(none)	**omp_get_max_task_priority()**
tool-var	(none)	(none)
tool-libraries-var	(none)	(none)
def-allocator-var	**omp_set_default_allocator()**	**omp_get_default_allocator()**

Description

- The value of the *nthreads-var* ICV is a list. The runtime call **omp_set_num_threads()** sets the value of the first element of this list, and **omp_get_max_threads()** retrieves the value of the first element of this list.
- The value of the *bind-var* ICV is a list. The runtime call **omp_get_proc_bind()** retrieves the value of the first element of this list.
- Detailed values in the *place-partition-var* ICV are retrieved using the runtime calls **omp_get_partition_num_places()**, **omp_get_partition_place_nums()**, **omp_get_place_num_procs()**, and **omp_get_place_proc_ids()**.

TABLE 2.3: Scopes of ICVs

ICV	Scope
dyn-var	data environment
nest-var	data environment
nthreads-var	data environment
run-sched-var	data environment
def-sched-var	device
bind-var	data environment
stacksize-var	device
wait-policy-var	device
thread-limit-var	data environment
max-active-levels-var	device
active-levels-var	data environment
levels-var	data environment
place-partition-var	implicit task
cancel-var	global
default-device-var	data environment
max-task-priority-var	global
tool-var	global
tool-libraries-var	global
def-allocator-var	data environment

Description

- There is one copy per device of each ICV with device scope
- Each data environment has its own copies of ICVs with data environment scope
- Each implicit task has its own copy of ICVs with implicit task scope

Calls to OpenMP API routines retrieve or modify data environment scoped ICVs in the data environment of their binding tasks.

table continued from previous page

ICV	construct clause, if used
def-sched-var	**`schedule`**
bind-var	**`proc_bind`**
stacksize-var	(none)
wait-policy-var	(none)
thread-limit-var	(none)
max-active-levels-var	(none)
active-levels-var	(none)
levels-var	(none)
place-partition-var	(none)
cancel-var	(none)
default-device-var	(none)
max-task-priority-var	(none)
tool-var	(none)
tool-libraries-var	(none)
def-allocator-var	(none)

Description

- The **`num_threads`** clause overrides the value of the first element of the *nthreads-var* ICV.
- If *bind-var* is not set to *false* then the **`proc_bind`** clause overrides the value of the first element of the *bind-var* ICV; otherwise, the **`proc_bind`** clause has no effect.

Cross References

- **`parallel`** construct, see Section 2.6 on page 54.
- **`proc_bind`** clause, Section 2.6 on page 54.
- **`num_threads`** clause, see Section 2.6.1 on page 59.
- Loop construct, see Section 2.8.1 on page 66.
- **`schedule`** clause, see Section 2.8.1.1 on page 74.

3.2.2 Memory spaces and allocators

2.5 Memory Spaces and Allocators

2.5.1 Memory Spaces

OpenMP memory spaces represent storage where variables are defined. A set of memory traits and the value that those traits have define the characteristics of each memory space. Table 2.5 shows the supported memory traits, the possible values each trait can take and their meaning. Trait values and their names are not case sensitive.

TABLE 2.5: Memory traits and their values

Memory trait	Matching rule	Allowed values	Description
`distance`	$\approx$	`near`, `far`	Specifies the relative physical distance of the memory space with respect to the task the request binds to.
`bandwidth`	$\approx$	`highest`, `lowest`	Specifies the relative bandwidth of the memory space with respect to other memories in the system.
`latency`	$\approx$	`highest`, `lowest`	Specifies the relative latency of the memory space with respect to other memories in the system.
`location`	$=$	see Table 2.6	Specifies the physical location of the memory space.

table continued on next page

table continued from previous page

Memory trait	Matching rule	Allowed values	Description
`optimized`	$=$	**`bandwidth`**, **`latency`**, **`capacity`**, **`none`**	Specifies if the memory space underlying technology is optimized to maximize a certain characteristic. The exact mapping of these values to actual technologies is implementation defined.
`pagesize`	$=$	positive integer	Specifies the size of the pages used by the memory space.
`permission`	$=$	**`r`**, **`w`**, **`rw`**	Specifies if read operations (r), write operations (w) or both (rw) are supported by the memory space.
`capacity`	$\geq$	positive integer	Specifies the physical capacity in bytes of the memory space.
`available`	$\geq$	positive integer	Specifies the current available capacity for new allocations in the memory space.

Table 2.6 shows the possible values for the **`location`** memory trait and their description. The values are not case sensitive. In addition, the **`location`** memory trait may accept other implementation specific values.

TABLE 2.6: Allowed values for the **`location`** memory trait

Location	Description
`core`	The memory space corresponds to a memory that is located within a core and might only accessible by the hardware threads of that core.
`socket`	The memory space corresponds to a memory that is located within a socket and might only be accessible by the hardware threads of that socket.
`device`	The memory space corresponds to a memory that is located within the device and is accessible by any hardware thread of that device.

Certain constructs and API routines will try to find a memory space that matches a list of pairs of memory traits and values. A memory space matches a list if every trait in the list matches the corresponding trait in the memory spaces according to the following rules:

- An empty list of memory traits matches any memory space.
- Traits with the $\geq$ matching rule match if the value of the trait in the memory space is greater or equal than the value in the list.
- Traits with the $=$ matching rule match if the value of the trait in the memory space is the same as the one in the list. For the **`location`** trait, for the matching to succeed it requires in addition that the task that the matching process binds to can access the memory space.
- Traits with the $\approx$ matching rule match if the value of the trait in the memory space compared to the value of the trait in other candidate memory spaces results in the value in the list.
- The matching process selects first memory spaces that match the $\geq$ and $=$ rules. From those selected in the previous step, it will select those that match the $\approx$ rules.

If more than one memory space would match a memory trait specification it is unspecified which memory space will be returned by the matching process. If a list contains more than a pair with the same memory trait it is unspecified which memory space, if any, will be matched.

2.5.2 How Allocation Works

Allocations are made through requests to an allocator. Allocators can be either explicit, those created with the API calls defined in Section 3.5, or implicit, those logically created because of a construct. When an allocator receives a request to allocate storage of a certain size, it will try to return an allocation of logically consecutive virtual memory in its associated memory space of at least the size being requested. The behavior of the allocation process can be affected by the allocator traits that the user specifies. Table 2.7 shows the allowed allocator traits, their possible values and the default value of each trait. Trait names and their values are not case sensitive.

TABLE 2.7: Allocator traits and their values

Allocator trait	Allowed values	Default value
`threadmodel`	`shared`, `exclusive`	`shared`
`alignment`	`0`, power of two integer	`0`
`pinned`	`true`, `false`	`false`
`fallback`	`null_fb`, `abort_fb`, `allocator_fb`, `default_fb`	`default_fb`
`fb_data`	an allocator handle	-

When an allocator `threadmodel` trait is defined to be `exclusive` the implementation can assume that no operation will be performed on the allocator by more than one thread at a time.

If either the allocator `alignment` trait or the allocation alignment of the request is greater than zero the allocated memory will be byte aligned to the maximum of the two values.

When an allocator `pinned` trait is defined to be `true` then the allocated memory must be pinned to physical pages. If the `pinned` trait is defined to be `false` then the allocated memory needs not to be pinned to physical pages.

The `fallback` trait specifies how the allocator behaves when it cannot fulfil the allocation request. If the `fallback` trait is set to `null_fb` the allocator returns the value zero if fails to allocate the memory. If the `fallback` trait is set to `abort_fb` the program execution will be terminated if the allocation fails. If the `fallback` trait is set to `allocator_fb` then when an allocation fails the request will be delegated to the allocator specified in the `fb_data` trait. If the `fallback` trait is set to `default_fb` then when an allocation fails another allocation will be tried in a memory space with the `omp_default_memspace_traits` memory traits assuming all allocator traits to be set to their default values except for `fallback` which will be set to `null_fb`.

3.2.3 Changes to existing directives

Fortran

If any operation of the base language causes a reallocation of an array that is allocated with an explicit or implicit OpenMP allocator then that allocator will be used to release the current memory and to allocate the new memory.

Fortran

2.6 `parallel` Construct

Summary

This fundamental construct starts parallel execution. See Section 1.3 on page 15 for a general description of the OpenMP execution model.

Syntax

C / C++

The syntax of the **parallel** construct is as follows:

```
#pragma omp parallel [clause[ [,] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

- **if(**[**parallel :**] *scalar-expression***)**
- **num_threads(***integer-expression***)**
- **default(shared | none)**
- **private(***list***)**
- **firstprivate(***list***)**
- **shared(***list***)**
- **copyin(***list***)**
- **reduction(***reduction-identifier* **:** *list***)**
- **proc_bind(master | close | spread)**
- allocate(modifiers:list)

C / C++

Fortran

The syntax of the **parallel** construct is as follows:

```
!$omp parallel [clause[ [,] clause] ... ]
    structured-block
!$omp end parallel
```

where *clause* is one of the following:

if([**parallel :**] *scalar-logical-expression***)**

num_threads(*scalar-integer-expression***)**

default(private | firstprivate | shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

copyin(*list***)**

reduction(*reduction-identifier* **:** *list***)**

proc_bind(master | close | spread)

allocate(modifiers:list)

The **end parallel** directive denotes the end of the **parallel** construct.

Fortran

Binding

The binding thread set for a **parallel** region is the encountering thread. The encountering thread becomes the master thread of the new team.

2.8.1 Loop Construct

Summary

The loop construct specifies that the iterations of one or more associated loops will be executed in parallel by threads in the team in the context of their implicit tasks. The iterations are distributed across threads that already exist in the team executing the **parallel** region to which the loop region binds.

Syntax

C / C++

The syntax of the loop construct is as follows:

```
#pragma omp for [clause[ [,] clause] ... ] new-line
    for-loops
```

where clause is one of the following:

private(*list***)**

firstprivate(*list***)**

lastprivate(*[lastprivate-modifier* **:** *] list***)**

linear(*list[* **:** *linear-step]***)**

reduction(*reduction-identifier* **:** *list***)**

schedule(*[modifier [*, *modifier]*:*]kind[*, *chunk_size]***)**

collapse(*n***)**

ordered*[***(***n***)** *]*

nowait

allocate(modifiers:list)

The **for** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form* (see Section 2.7 on page 62).

C / C++

Fortran

The syntax of the loop construct is as follows:

```
!$omp do [clause[ [,] clause] ... ]
    do-loops
[!$omp end do [nowait]]
```

where *clause* is one of the following:

- **private(***list***)**
- **firstprivate(***list***)**
- **lastprivate(***[lastprivate-modifier* **:** *] list***)**
- **linear(***list[* **:** *linear-step]***)**
- **reduction(***reduction-identifier* **:** *list***)**
- **schedule(***[modifier [*, *modifier]***:** *]kind[*, *chunk_size]***)**
- **collapse(***n***)**
- **ordered***[* **(***n***)** *]*
- allocate(modifiers:list)

If an **end do** directive is not specified, an **end do** directive is assumed at the end of the *do-loops*.

Any associated *do-loop* must be a *do-construct* or an *inner-shared-do-construct* as defined by the Fortran standard. If an **end do** directive follows a *do-construct* in which several loop statements share a **DO** termination statement, then the directive can only be specified for the outermost of these **DO** statements.

If any of the loop iteration variables would otherwise be shared, they are implicitly made private on the loop construct.

Fortran

Binding

The binding thread set for a loop region is the current team. A loop region binds to the innermost enclosing **parallel** region. Only the threads of the team executing the binding **parallel** region participate in the execution of the loop iterations and the implied barrier of the loop region if the barrier is not eliminated by a **nowait** clause.

2.8.2 `sections` Construct

Summary

The **`sections`** construct is a non-iterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team. Each structured block is executed once by one of the threads in the team in the context of its implicit task.

Syntax

C / C++

The syntax of the **`sections`** construct is as follows:

```
#pragma omp sections [clause[ [,] clause] ... ] new-line
   {
   [#pragma omp section new-line]
      structured-block
   [#pragma omp section new-line
      structured-block]
   ...
   }
```

where *clause* is one of the following:

`private(`*list***`)`**

`firstprivate(`*list***`)`**

`lastprivate(`*[lastprivate-modifier* **`:`** *] list***`)`**

`reduction(`*reduction-identifier* **`:`** *list***`)`**

`nowait`

allocate(modifiers:list)

C / C++

Fortran

The syntax of the **sections** construct is as follows:

```
!$omp sections [clause[ [,] clause] ... ]
   [!$omp section]
       structured-block
   [!$omp section
       structured-block]
   ...
!$omp end sections [nowait]
```

where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

lastprivate(*[lastprivate-modifier* **:** *] list***)**

reduction(*reduction-identifier* **:** *list***)**

allocate(modifiers:list)

Fortran

Binding

The binding thread set for a **sections** region is the current team. A **sections** region binds to the innermost enclosing **parallel** region. Only the threads of the team executing the binding **parallel** region participate in the execution of the structured blocks and the implied barrier of the **sections** region if the barrier is not eliminated by a **nowait** clause.

Description

Each structured block in the **sections** construct is preceded by a **section** directive except possibly the first block, for which a preceding **section** directive is optional.

The method of scheduling the structured blocks among the threads in the team is implementation defined.

There is an implicit barrier at the end of a **sections** construct unless a **nowait** clause is specified.

2.8.3 `single` Construct

Summary

The **`single`** construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task. The other threads in the team, which do not execute the block, wait at an implicit barrier at the end of the **`single`** construct unless a **`nowait`** clause is specified.

Syntax

C / C++

The syntax of the single construct is as follows:

```
#pragma omp single [clause[ [,] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

`private(`*list***`)`**

`firstprivate(`*list***`)`**

`copyprivate(`*list***`)`**

`nowait`

allocate(modifiers:list)

C / C++

Fortran

The syntax of the **`single`** construct is as follows:

```
!$omp single [clause[ [,] clause] ... ]
    structured-block
!$omp end single [end_clause[ [,] end_clause] ... ]
```

where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

allocate(modifiers:list)

and *end_clause* is one of the following:

copyprivate(*list***)**

nowait

Fortran

Binding

The binding thread set for a **single** region is the current team. A **single** region binds to the innermost enclosing **parallel** region. Only the threads of the team executing the binding **parallel** region participate in the execution of the structured block and the implied barrier of the **single** region if the barrier is not eliminated by a **nowait** clause.

Description

The method of choosing a thread to execute the structured block is implementation defined. There is an implicit barrier at the end of the **single** construct unless a **nowait** clause is specified.

Events

The *single-begin* event occurs after an **implicit task** encounters a **single** construct but before the task starts the execution of the structured block of the **single** region.

The *single-end* event occurs after a **single** region finishes execution of the structured block but before resuming execution of the encountering implicit task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback for each occurrence of *single-begin* and *single-end* events in that thread. The callback has type signature **ompt_callback_work_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_work_single_executor** or **ompt_work_single_other** as its *wstype* argument.

2.10 Tasking Constructs

2.10.1 `task` Construct

Summary

The **task** construct defines an explicit task.

Syntax

C / C++

The syntax of the **task** construct is as follows:

```
#pragma omp task [clause[ [,] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

if([**task :**] *scalar-expression***)**

final(*scalar-expression***)**

untied

default(shared | none)

mergeable

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

in_reduction(*reduction-identifier* **:** *list***)**

depend(*dependence-type* **:** *locator-list***)**

priority(*priority-value***)**

allocate(modifiers:list)

C / C++

Fortran

The syntax of the **task** construct is as follows:

```
!$omp task [clause[ [,] clause] ... ]
      structured-block
!$omp end task
```

where *clause* is one of the following:

- **if(**[**task :**] *scalar-logical-expression***)**
- **final(***scalar-logical-expression***)**
- **untied**
- **default(private | firstprivate | shared | none)**
- **mergeable**
- **private(***list***)**
- **firstprivate(***list***)**
- **shared(***list***)**
- **in_reduction(***reduction-identifier* **:** *list***)**
- **depend(***dependence-type* **:** *locator-list***)**
- **priority(***priority-value***)**
- allocate(modifiers:list)

Fortran

Binding

The binding thread set of the **task** region is the current team. A **task** region binds to the innermost enclosing **parallel** region.

num_tasks (*num-tasks*)

collapse (*n*)

final (*scalar-expr*)

priority (*priority-value*)

untied

mergeable

nogroup

allocate(modifiers:list)

The **taskloop** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have canonical loop form (see Section 2.7 on page 62).

C / C++

Fortran

The syntax of the **taskloop** construct is as follows:

```
!$omp taskloop [clause[[,] clause] ...]
      do-loops
[!$omp end taskloop]
```

where *clause* is one of the following:

if ([**taskloop :**] *scalar-logical-expr*)

shared (*list*)

private (*list*)

firstprivate (*list*)

lastprivate (*list*)

reduction (*reduction-identifier* **:** *list*)

in_reduction (*reduction-identifier* **:** *list*)

default (**private** | **firstprivate** | **shared** | **none**)

grainsize (*grain-size*)

num_tasks (*num-tasks*)

collapse (*n*)

final (*scalar-logical-expr***)**

priority (*priority-value***)**

untied

mergeable

nogroup

allocate(modifiers:list)

If an **end taskloop** directive is not specified, an **end taskloop** directive is assumed at the end of the *do-loops*.

Any associated *do-loop* must be *do-construct* or an *inner-shared-do-construct* as defined by the Fortran standard. If an **end taskloop** directive follows a *do-construct* in which several loop statements share a **DO** termination statement, then the directive can only be specified for the outermost of these **DO** statements.

If any of the loop iteration variables would otherwise be shared, they are implicitly made private for the loop-iteration tasks generated by the **taskloop** construct. Unless the loop iteration variables are specified in a **lastprivate** clause on the **taskloop** construct, their values after the loop are unspecified.

Fortran

Binding

The binding thread set of the **taskloop** region is the current team. A **taskloop** region binds to the innermost enclosing **parallel** region.

Description

The **taskloop** construct is a *task generating construct*. When a thread encounters a **taskloop** construct, the construct partitions the associated loops into explicit tasks for parallel execution of the loops' iterations. The data environment of each generated task is created according to the data-sharing attribute clauses on the **taskloop** construct, per-data environment ICVs, and any defaults that apply. The order of the creation of the loop tasks is unspecified. Programs that rely on any execution order of the logical loop iterations are non-conforming.

By default, the **taskloop** construct executes as if it was enclosed in a **taskgroup** construct with no statements or directives outside of the **taskloop** construct. Thus, the **taskloop** construct creates an implicit **taskgroup** region. If the **nogroup** clause is present, no implicit **taskgroup** region is created.

3.2.4 Allocate directive and clause

As another example, if a lock acquire and release happen in different parts of a task region, no attempt should be made to acquire the same lock in any part of another task that the executing thread may schedule. Otherwise, a deadlock is possible. A similar situation can occur when a **`critical`** region spans multiple parts of a task and another schedulable task contains a **`critical`** region with the same name.

The use of threadprivate variables and the use of locks or critical sections in an explicit task with an **`if`** clause must take into account that when the **`if`** clause evaluates to *false*, the task is executed immediately, without regard to *Task Scheduling Constraint* 2.

Events

The *task-schedule* event occurs in a thread when the thread switches tasks at a task scheduling point; no event occurs when switching to or from a merged task.

Tool Callbacks

A thread dispatches a registered **`ompt_callback_task_schedule`** callback for each occurrence of a *task-schedule* event in the context of the task that begins or resumes. This callback has the type signature **`ompt_callback_task_schedule_t`**. The argument *prior_task_status* is used to indicate the cause for suspending the prior task. This cause may be the completion of the prior task region, the encountering of a **`taskyield`** construct, or the encountering of an active cancellation point.

Cross References

- **`ompt_callback_task_schedule_t`**, see Section 4.6.2.10 on page 409.

2.11 Memory Management Directives

2.11.1 `allocate` Directive

Summary

The **`allocate`** directive specifies how a set of variables are allocated.

C / C++

The **`allocate`** directive is a declarative directive.

C / C++

Fortran

The **allocate** directive is a declarative directive if it is not associated with an **allocate** statement.

Fortran

Syntax

C / C++

The syntax of the **allocate** directive is as follows:

```
#pragma omp allocate(list) [clause[ [ [,] clause] ... ]] new-line
```

where clause is one of the following:

allocator(allocator)

memspace(memspace)

alloctraits(alloctrait-list)

memtraits(memtrait-list)

safe_align(alignment)

where allocator is an expression of the **omp_allocator_t** type.

where memspace is an expression of the **omp_memspace_t** type.

where alloctrait-list is a key-value list where the allowed keys are the allocator traits keys and the allowed values are the accepted values for each key.

where memtrait-list is a key-value list where the allowed keys are the memory traits keys and the allowed values are the accepted values for each key.

where alignment is an integer expression that must evaluate to a power of two.

C / C++

Fortran

The syntax of the **allocate** directive is as follows:

```
!$omp allocate(list) [clause[ [ [,] clause] ... ]]
```

or

```
!$omp allocate[(list)] clause[ [ [,] clause] ... ]
    allocate statement
```

where clause is one of the following:

allocator(allocator)

memspace(memspace)

alloctraits(alloctrait-list)

memtraits(memtrait-list)

safe_align(alignment)

where allocator is an integer expression of the **omp_allocator_kind** kind.

where memspace is an integer expression of the **omp_memspace_kind** kind.

where alloctrait-list is a key-value list where the allowed keys are the allocator traits keys and the allowed values are the accepted values for each key.

where memtrait-list is a key-value list where the allowed keys are the memory traits keys and the allowed values are the accepted values for each key.

where alignment is an integer expression that must evaluate to a power of two.

Fortran

Description

If the directive is not associated with a Fortran **allocate** statement, the storage for each list item that appears in the directive will be provided by an allocation through an allocator. If no clause is specified then the allocator specified by the def-allocator-var ICV will be used. If the allocator clause is specified, the allocator specified in the clause will be used. Otherwise, the allocation will be provided as if using an allocator that had been built with the specified allocator traits, memory traits and/or the memspace memory space. If the **safe_align** clause is specified, then the allocation alignment of the request will the value of the **safe_align** clause.

The scope of this allocation is that of the list item in the base language. When the allocation reaches the end of the scope it will be deallocated through the specified allocator or as if using an allocator that had been built with the specified allocator traits, memory traits and/or the memspace memory space. If the execution leaves the scope in a manner not supported by the base language it is unspecified whether the deallocation happens or not.

Fortran

If the directive is associated with a Fortran **allocate** statement, the allocation of the specified list items will be provided through an allocator. If no clause is specified then the allocator specified by the def-allocator-var ICV will be used. If the allocator clause is specified, the allocator specified in the clause will be used. Otherwise, the allocation will be provided as if using an allocator that had been built with the specified allocator traits, memory traits and/or the memspace memory space. If no list item is specified then all variables allocated by the **allocate** statement will be provided by the allocator.

Fortran

For allocations that arise from this directive the **null_fb** value of the **fallback** allocator trait will behave as if the **abort_fb** had been specified.

Restrictions

- A variable that is part of another variable (as an array or structure element) cannot appear in an **allocate** directive.
- The directive must appear in the same scope of the list item declaration and before its first use.
- If the **allocator** clause is present, no other clause must be specified.
- If the **allocator** clause is present, the allocator must be an allocator returned by the **omp_init_allocator** routine.
- At most one **allocator** clause can appear on the **allocate** directive.
- If the **memspace** clause is present, the **memtraits** clause must not be specified.
- If the **memspace** clause is present, the memspace must be a memory space returned by the **omp_init_memspace** routine.
- At most one **memspace** clause can appear on the **allocate** directive.
- If the **safe_align** clause is present, its value must a power of two.

C / C++

- If a list item has a static storage type, the **allocator** and the **memspace** clauses must not be specified.
- If a list item has a static storage type, the **fallback** allocator trait must not have the **allocator_fb** value.

C / C++

Fortran

- List items specified in the **allocate** directive must not have the **ALLOCATABLE** attribute unless the directive is associated with an **allocate** statement.
- List items specified in an **allocate** directive that is associated with an **allocate** statement must be **ALLOCATABLE** variables allocated by the **allocate** statement.

Fortran

Cross References

- Memory spaces, allocators and their traits, see Section 2.5 on page 50.

C / C++

- **omp_memspace_t** and **omp_allocator_t**, see Section 3.5.1 on page 328.

C / C++

Fortran

- **omp_memspace_kind** and **omp_allocator_kind**, see Section 3.5.1 on page 328.

Fortran

2.11.2 The allocate Clause

Summary

The **allocate** clause specifies the allocation and memory traits of the storage used for private variables of a directive.

Syntax

The syntax of the **`allocate`** clause is as follows:

```
allocate([modifiers:] list)
```

where modifiers is a comma separated list of one or more of the following:

- allocator(allocator)
- memspace(memspace)
- alloctraits(alloctrait-list)
- memtraits(memtrait-list)
- safe_align(alignment)

C / C++

where allocator is an integer expression of the **`omp_allocator_t`** type.

where memspace is an integer expression of the **`omp_memspace_t`** type.

C / C++

Fortran

where allocator is an integer expression of the **`omp_allocator_kind`** kind.

where memspace is an integer expression of the **`omp_memspace_kind`** kind.

Fortran

where alloctrait-list is a key-value list where the allowed keys are the allocator traits keys and the allowed values are the accepted values for each key.

where memtrait-list is a key-value list where the allowed keys are the memory traits keys and the allowed values are the accepted values for each key.

where alignment is an integer expression that must evaluate to a power of two.

Description

The storage for new list items that arise from list item that appear in the directive will be provided by an allocation through an allocator. If no modifier is specified then the allocator specified by the def-allocator-var ICV will be used. If the allocator modifier is specified, the allocator specified in the clause will be used. Otherwise, the allocation will be provided as if using an allocator that had been built with the specified allocator traits, memory traits and/or the memspace memory space. For allocations that arise from this clause the **null_fb** value of the **fallback** allocator trait will behave as if the **abort_fb** had been specified. If the **safe_align** modifier is specified, then the allocation alignment of the request will be the value of the **safe_align** modifier.

Restrictions

- List items specified in the **allocate** clause must also be specified in a **private**, **firstprivate**, **lastprivate**, **linear** or **reduction** clause in the same directive.
- If the **allocator** modifier is present, no other modifier must be specified.
- If the **allocator** modifier is present, the allocator must be an allocator returned by the **omp_init_allocator** routine.
- At most one **allocator** modifier can appear on the **allocate** clause.
- If the **memspace** modifier is present, the **memtraits** modifier must not be specified.
- If the **memspace** modifier is present, the memspace must be a memory space returned by the **omp_init_memspace** routine.
- At most one **memspace** modifier can appear on the **allocate** modifier.

Cross References

- Memory spaces, allocators and their traits, see Section 2.5 on page 50.

C / C++

- **omp_memspace_t** and **omp_allocator_t**, see Section 3.5.1 on page 328.

C / C++

Fortran

- **omp_memspace_kind** and **omp_allocator_kind**, see Section 3.5.1 on page 328.

Fortran

3.3 Changes to Chapter 3

C / C++

3.5 Memory Management Routines

This section describes routines that support management of memory on the current device.

Instances of OpenMP memory management types must be accessed only through the routines described in this section; programs that otherwise access OpenMP instances of these types are non-conforming.

3.5.1 Memory Management Types

The following type definitions are used by the memory management routines:

C / C++

The type **`omp_uintptr_t`** must be defined as an unsigned integer that is capable of storing an address.

```
typedef enum {
    OMP_MTK_DISTANCE,
    OMP_MTK_LOCATION,
    OMP_MTK_BANDWIDTH,
    OMP_MTK_LATENCY,
    OMP_MTK_OPTIMIZED
    OMP_MTK_PAGESIZE,
    OMP_MTK_PERMISSION,
    OMP_MTK_CAPACITY,
    OMP_MTK_AVAILABLE
} omp_memtrait_key_t;

typedef enum {
    OMP_MTV_FALSE = 0,
    OMP_MTV_TRUE = 1,
    OMP_MTV_NEAR,
    OMP_MTV_FAR,
    OMP_MTV_CORE,
    OMP_MTV_SOCKET,
    OMP_MTV_NODE,
    OMP_MTV_HIGHEST,
```

```
        OMP_MTV_LOWEST,
        OMP_MTV_BANDWIDTH,
        OMP_MTV_LATENCY,
        OMP_MTV_CAPACITY,
        OMP_MTV_NONE,
        OMP_MTV_R,
        OMP_MTV_W,
        OMP_MTV_RW = OMP_MTV_R | OMP_MTV_W,
    } omp_memtrait_value_t;


    typedef struct {
        omp_memtrait_key_t key;
        omp_uintptr_t value;
    } omp_memtrait_t;


    typedef enum {
        OMP_ATK_THREADMODEL,
        OMP_ATK_ALIGNMENT,
        OMP_ATK_PIN,
        OMP_ATK_FALLBACK,
        OMP_ATK_FB_DATA
    } omp_alloctrait_key_t;


    typedef enum {
        OMP_ATV_FALSE = 0,
        OMP_ATV_TRUE = 1,
        OMP_ATV_SHARED,
        OMP_ATV_EXCLUSIVE,
        OMP_ATV_ABORT_FB,
        OMP_ATV_NULL_FB,
        OMP_ATV_ALLOCATOR_FB,
        OMP_ATV_DEFAULT_FB
    } omp_alloctrait_value_t;


    typedef struct {
        omp_alloctrait_key_t key;
        omp_uintptr_t value;
    } omp_alloctrait_t;
```

```
omp_memtrait_set_t;
const omp_memtrait_set_t omp_default_memspace_traits;
omp_memspace_t;
enum { OMP_NULL_MEMSPACE = NULL };


omp_alloctrait_set_t;
const omp_alloctrait_set_t omp_default_allocator_traits;
omp_allocator_t;
enum { OMP_NULL_ALLOCATOR = NULL };
```

C / C++

Fortran

```
integer parameter omp_memtrait_key_kind

integer(kind=omp_memtrait_key_kind), &
        parameter :: omp_mtk_distance
integer(kind=omp_memtrait_key_kind), &
        parameter :: omp_mtk_location
integer(kind=omp_memtrait_key_kind), &
        parameter :: omp_mtk_bandwidth
integer(kind=omp_memtrait_key_kind), &
        parameter :: omp_mtk_latency
integer(kind=omp_memtrait_key_kind), &
        parameter :: omp_mtk_optimized
integer(kind=omp_memtrait_key_kind), &
        parameter :: omp_mtk_pagesize
integer(kind=omp_memtrait_key_kind), &
        parameter :: omp_mtk_permission
integer(kind=omp_memtrait_key_kind), &
        parameter :: omp_mtk_capacity
integer(kind=omp_memtrait_key_kind), &
        parameter :: omp_mtk_available


integer parameter omp_memtrait_val_kind


integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_false = 0
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_true = 1
```

```
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_near
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_far
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_core
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_socket
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_node
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_highest
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_lowest
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_bandwidth
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_latency
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_capacity
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_none
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_r
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_w
integer(kind=omp_memtrait_val_kind), &
        parameter :: omp_mtv_rw = IOR(omp_mtv_r,omp_mtv_w)


type omp_memtrait
  integer(kind=omp_memtrait_key_kind) key
  integer(kind=omp_memtrait_val_kind) value
end type omp_memtrait


integer parameter omp_alloctrait_key_kind


integer(kind=omp_alloctrait_key_kind), &
        parameter :: omp_atk_threadmodel
integer(kind=omp_alloctrait_key_kind), &
        parameter :: omp_atk_alignment
```

```
integer(kind=omp_alloctrait_key_kind), &
            parameter :: omp_atk_pin
integer(kind=omp_alloctrait_key_kind), &
            parameter :: omp_atk_fallback
integer(kind=omp_alloctrait_key_kind), &
            parameter :: omp_atk_fb_data


integer parameter omp_alloctrait_val_kind


integer(kind=omp_alloctratit_val_kind), &
            parameter :: omp_atv_true = 0
integer(kind=omp_alloctratit_val_kind), &
            parameter :: omp_atv_false = 1
integer(kind=omp_alloctratit_val_kind), &
            parameter :: omp_atv_shared
integer(kind=omp_alloctratit_val_kind), &
            parameter :: omp_atv_exclusive
integer(kind=omp_alloctratit_val_kind), &
            parameter :: omp_atv_abort_fb
integer(kind=omp_alloctratit_val_kind), &
            parameter :: omp_atv_null_fb
integer(kind=omp_alloctratit_val_kind), &
            parameter :: omp_atv_allocator_fb
integer(kind=omp_alloctratit_val_kind), &
            parameter :: omp_atv_default_fb


type omp_alloctrait
   integer(kind=omp_alloctrait_key_kind) key
   integer(kind=omp_alloctrait_val_kind) value
end type omp_alloctrait


integer parameter omp_memtrait_set_kind
integer(kind=omp_memtrait_set_kind), &
            parameter :: omp_default_memspace_traits
integer parameter omp_memspace_kind
integer(kind=omp_memspace_kind), &
            parameter :: omp_null_memspace = 0
```

3.3.1 Routines for defining memory traits

```
integer parameter omp_alloctrait_set_kind
integer(kind=omp_alloctrait_set_kind), &
          parameter :: omp_default_allocator_traits
integer parameter omp_allocator_kind
integer(kind=omp_allocator_kind), &
          parameter :: omp_null_allocator = 0
```

Fortran

3.5.2 omp_init_memtrait_set

Summary

The **omp_init_memtrait_set** routine initializes an OpenMP memory traits set.

Format

C / C++

```
void omp_init_memtrait_set (omp_memtrait_set_t *set,
                            size_t ntraits,
                            omp_memtrait_t *traits);            (C)
void omp_init_memtrait_set (omp_memtrait_set_t *set,
                            size_t ntraits = 0,
                            omp_memtrait_t *traits = NULL);      (C++)
```

C / C++

Fortran

```
subroutine omp_init_memtrait_set ( set, ntraits, traits )
integer(kind=omp_memtrait_set_kind),intent(out) :: set
integer,intent(in) :: ntraits
type(omp_memtrait),intent(in) :: traits(*)
```

Fortran

Binding

The binding thread set for an **omp_init_memtrait_set** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Constraints on Arguments

If the ntraits argument is greater than zero, then there must be at least as many traits specified in the traits argument; otherwise, the behavior is unspecified.

Effect

The effect of the **omp_init_memtrait_set** routine is to initialize the memory trait set in the set argument to the memory traits specified in the traits argument. The number of traits to be included in the set is specified by the ntraits argument.

3.5.3 omp_destroy_memtrait_set

Summary

The **omp_destroy_memtrait_set** routine ensures that an OpenMP memory traits set is uninitialized.

Format

C / C++

```
void omp_destroy_memtrait_set (omp_memtrait_set_t *set);
```

C / C++

Fortran

```
subroutine omp_destroy_memtrait_set ( set )
integer(kind=omp_memtrait_set_kind),intent(inout) :: set
```

Fortran

Binding

The binding thread set for an **omp_destroy_memtrait_set** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The effect of the **omp_destroy_memtrait_set** routine is to uninitialize the memory traits set specified in the first argument.

3.5.4 omp_add_memtraits

Summary

The **omp_add_memtraits** routine adds a memory trait to the memory traits set.

Format

C / C++

```
void omp_add_memtraits (omp_memtrait_set_t *set,
                        size_t ntraits,
                        omp_memtrait_t *traits);
```

C / C++

Fortran

```
subroutine omp_add_memtraits ( set, ntraits, traits )
integer(kind=omp_memtrait_set_kind),intent(inout) :: set
integer,intent(in) :: ntraits
type(omp_memtrait),intent(in) :: traits(*)
```

Fortran

Constraints on Arguments

If the ntraits argument is greater than zero, then there must be at least as many traits specified in the traits argument; otherwise, the behavior is unspecified.

Binding

The binding thread set for an **omp_add_memtraits** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The effect of the **omp_add_memtraits** routine is that the ntraits specified in traits are added to the set of memory traits.

Cross References

- Memory traits in Section 2.5.1 on page 50

3.5.5 omp_merge_memtraits

Summary

The **omp_merge_memtraits** routine merges two memory traits sets.

Format

C / C++

```
void omp_merge_memtraits (omp_memtrait_set_t *dst,
                          const omp_memtrait_set_t *src,
                          int dst_priority );   (C)
void omp_merge_memtraits (omp_memtrait_set_t *dst,
                          const omp_memtrait_set_t *src,
                          bool dst_priority = true);   (C++)
```

C / C++

Fortran

```
subroutine omp_merge_memtraits ( dst, src, dst_priority )
integer(kind=omp_memtrait_set_kind),intent(inout) :: dst
integer(kind=omp_memtrait_set_kind),intent(in) :: src
logical :: dst_priority
```

Fortran

Binding

The binding thread set for an **omp_merge_memtraits** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

3.3.2 Routines for memory spaces

Effect

The effect of the **omp_merge_memtraits** routine is that the two memory traits sets dst and src are merged into dst. If the **available** trait appears in both sets the merged valued for the trait will be the result of adding the values in each set. If the **capacity** trait appears in both sets the merged value for the trait will be the greater of the values in either set. For any other trait, if the same memory trait appears in both sets, if the dst_priority argument evaluates to **true** the merged value will be that of the dst set; otherwise, it will the value of the src set.

3.5.6 omp_init_memspace

Summary

The **omp_init_memspace** routine returns handler to a memory space that matches the specified memory traits.

Format

C / C++

```
omp_memspace_t * omp_init_memspace(const omp_memtrait_set_t *traits);
```

C / C++

Fortran

```
integer(kind=omp_memspace_kind) function omp_init_memspace (traits)
integer(kind=omp_memtrait_set_kind),intent(in) :: traits
```

Fortran

Binding

The binding thread set for an **omp_init_memspace** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Constraints on Arguments

The traits argument must have been initialized with the **omp_init_memtrait_set** routine.

Effect

The `omp_init_memspace` routine returns a handler to a memory space in the current device that matches the memory traits specified in the traits set. If no memory space is found that matches the specified memory traits then the special value `OMP_NULL_MEMSPACE` is returned.

The traits in `omp_default_memspace_traits` must be defined in such a way that it guarantees that the `omp_init_memspace` routine will return a valid memory space that is always the same and that an allocation from that memory space is guaranteed to be accessible to all threads on that device without any special consideration.

Cross References

- Memory spaces in Section 2.5.1 on page 50

3.5.7 omp_destroy_memspace

Summary

The `omp_destroy_memspace` releases all resources associated with a memory space handler.

Format

C / C++

```
void omp_destroy_memspace (omp_memspace_t *memspace);
```

C / C++

Fortran

```
subroutine omp_destroy_memspace ( memspace )
integer(kind=omp_memspace_kind),intent(out) :: memspace
```

Fortran

3.3.3 Routines for defining allocator traits

Binding

The binding thread set for an **omp_destroy_memspace** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_destroy_memspace** routine releases resources associated with the memspace handler. Accessing allocators, or memory allocated by them, that have been associated through the memspace handler results in unspecified behavior.

3.5.8 omp_init_alloctrait_set

Summary

The **omp_init_alloctrait_set** initializes an OpenMP allocator traits set.

Format

C / C++

```
void omp_init_alloctrait_set (omp_alloctrait_set_t *set
                              size_t ntraits,
                              omp_alloctrait_t *traits);        (C)
void omp_init_alloctrait_set (omp_alloctrait_set_t *set
                              size_t ntraits = 0,
                              omp_alloctrait_t *traits = NULL); (C++)
```

C / C++

Fortran

```
subroutine omp_init_alloctrait_set ( set, ntraits, traits )
integer(kind=omp_alloctrait_set_kind),intent(out) :: set
integer,intent(in) :: ntraits
type(omp_alloctrait),intent(in) :: traits(*)
```

Fortran

Binding

The binding thread set for an **omp_init_alloctrait_set** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Constraints on Arguments

If the ntraits argument is greater than zero, then there must be at least as many traits specified in the traits argument. If there are fewer than ntraits traits the behavior is unspecified.

Effect

The effect of the **omp_init_alloctrait_set** routine is to initialize the allocator trait set in the set argument to the allocator traits specified in the traits argument. The number of traits to be included in the set is specified by the ntraits argument.

3.5.9 omp_destroy_alloctrait_set

Summary

The **omp_destroy_alloctrait_set** routine ensures that an OpenMP allocator traits set is uninitialized.

Format

C / C++

```
void omp_destroy_alloctrait_set (omp_alloctrait_set_t *set);
```

C / C++

Fortran

```
subroutine omp_destroy_alloctrait_set ( set )
integer(kind=omp_alloctrait_set_kind),intent(inout) :: set
```

Fortran

Binding

The binding thread set for an **omp_destroy_alloctrait_set** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The effect of the **omp_destroy_alloctrait_set** routine is to uninitialize the allocator traits set specified in the first argument.

3.5.10 omp_add_alloctraits

Summary

The **omp_add_alloctraits** routine adds an allocator trait to the allocator traits set.

Format

C / C++

```
void omp_add_alloctraits (omp_alloctrait_set_t *set,
                          size_t ntraits,
                          omp_alloctrait_t *traits);
```

C / C++

Fortran

```
subroutine omp_add_alloctraits ( set, ntraits, traits )
integer(kind=omp_alloctrait_set_kind),intent(inout) :: set
integer,intent(in) :: ntraits
type(omp_alloctrait),intent(in) :: traits(*)
```

Fortran

Binding

The binding thread set for an **omp_add_alloctrait** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Constraints on Arguments

If the ntraits argument is greater than zero, then there must be at least as many traits specified in the traits argument; otherwise, the behavior is unspecified.

Effect

The effect of the **omp_add_alloctraits** routine is that the ntraits specified in traits are added to the set of allocator traits.

Cross References

- Allocator traits in Section 2.5.2 on page 52

3.5.11 omp_merge_alloctraits

Summary

The **omp_merge_alloctraits** routine merges two allocator traits sets.

Format

C / C++

```
void omp_merge_alloctraits (omp_alloctrait_set_t *dst,
                            const omp_alloctrait_set_t *src,
                            int dst_priority); (C)
void omp_merge_alloctraits (omp_alloctrait_set_t *dst,
                            const omp_alloctrait_set_t *src,
                            bool dst_priority=true); (C++)
```

C / C++

Fortran

```
subroutine omp_merge_alloctraits ( dst, src, dst_priority )
integer(kind=omp_alloctrait_set_kind),intent(inout) :: dst
integer(kind=omp_alloctrait_set_kind),intent(in) :: src
logical :: dst_priority
```

Fortran

3.3.4 Routines for allocators

Binding

The binding thread set for an **omp_merge_alloctraits** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The effect of the **omp_merge_alloctraits** routine is that the two allocator traits sets dst and src are merged into dst. If the same memory trait appears in both sets, and the dst_priority argument evaluates to **true** the merged value will be that of the dst set; otherwise, it will the value of the src set.

3.5.12 omp_init_allocator

Summary

The **omp_init_allocator** initializes an allocator and associates it with a memory space.

Format

C / C++

```
omp_allocator_t * omp_init_allocator ( omp_memspace_t *memspace,
                                       const omp_alloctrait_set_t *traits);
```

C / C++

Fortran

```
integer(kind=omp_allocator_kind)
function omp_init_allocator ( memspace, traits )
integer(kind=omp_memspace_kind),intent(in) :: memspace
integer(kind=omp_alloctrait_set_kind),intent(in) :: traits
```

Fortran

Binding

The binding thread set for an **omp_init_allocator** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Constraints on Arguments

The memspace argument must be a memory space returned by the **omp_init_memspace** routine. The traits argument must have been initialized with the **omp_init_alloctrait_set** routine.

Effect

The **omp_init_allocator** routine creates a new allocator that is associated with the memory space represented by the memspace handler. The allocations done through the created allocator will behave according to the allocator traits specified in the traits argument. Specifying the same allocator trait more than once results in unspecified behavior. The routine returns a handler for the created allocator. If the traits argument is an empty set this routine will always return a handler to an allocator. If the traits argument is not empty and an allocator that satisfies the requirements cannot be created then the special value **OMP_NULL_ALLOCATOR** is returned.

The traits in **omp_default_allocator_traits** must be defined as an empty set of allocator traits.

Cross References

- Allocators in Section 2.5.2 on page 52

3.5.13 omp_destroy_allocator

Summary

The **omp_destroy_allocator** releases all resources and memory allocations associated to an allocator.

Format

C / C++

```
void omp_destroy_allocator (omp_allocator_t *allocator);
```

C / C++

Fortran

```
subroutine omp_destroy_allocator ( allocator )
integer(kind=omp_allocator_kind),intent(out) :: allocator
```

Fortran

Binding

The binding thread set for an **omp_destroy_allocator** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_destroy_allocator** routine releases resources that might be associated with the allocator handler. Also, any memory allocated by the allocator but not deallocated yet is deallocated by this routine.

3.5.14 omp_set_default_allocator

Summary

The **omp_set_default_allocator** sets the default allocator to be used by allocation calls, directives and clauses that use default allocation.

Format

C / C++

```
void omp_set_default_allocator (omp_allocator_t *allocator);
```

C / C++

Fortran

3.3.5 Routines for allocation/free

```
integer(kind=omp_allocator_kind)
function omp_get_default_allocator ()
```

Fortran

Binding

The binding task set for an **`omp_get_default_allocator`** region is the generating task.

Effect

The effect of this routine is to return the value of the def-allocator-var ICV of the current task.

Cross References

- def-allocator-var ICV, see Section 2.3 on page 39.
- **`omp_alloc`** routine, see Section 3.5.16 on page 347.

C / C++

3.5.16 `omp_alloc`

Summary

The **`omp_alloc`** requests a memory allocation to an allocator.

Format

```
void * omp_alloc (size_t size, omp_allocator_t *allocator);   (C)
void * omp_alloc (size_t size,
           omp_allocator_t *allocator=OMP_NULL_ALLOCATOR);  (C++)
```

Effect

The **omp_alloc** routine requests a memory allocation of size bytes from the specified allocator without specifying an allocation alignment. If value of the allocator argument is **OMP_NULL_ALLOCATOR** the allocator used by the routine will be the one specified by the def-allocator-var ICV. Upon success it returns a pointer to the allocated memory. Otherwise, the behavior of the call depends on the **fallback** trait of the allocator.

Cross References

- How Allocations Works, see Section 2.5.2 on page 52.

3.5.17 omp_alloc_safe_align

Summary

The **omp_alloc_safe_align** requests a memory allocation to an allocator with an allocation alignment.

Format

```
void * omp_alloc_safe_align (size_t size, size_t alignment,
                 omp_allocator_t *allocator);                 (C)
void * omp_alloc_safe_align (size_t size, size_t alignment,
                 omp_allocator_t *allocator=OMP_NULL_ALLOCATOR);(C++)
```

Constraints on Arguments

The allocator must be an allocator returned by the **omp_init_allocator** routine. Specifying an alignment argument that is not a power of two results in unspecified behavior.

Effect

The **omp_alloc_safe_align** routine requests a memory allocation of size bytes from the specified allocator where the allocation alignment of the request is alignment. If value of the allocator argument is **OMP_NULL_ALLOCATOR** the allocator used by the routine will be the one specified by the def-allocator-var ICV. Upon success it returns a pointer to the allocated memory. Otherwise, the behavior of the call depends on the **fallback** trait of the allocator.

Cross References

- How Allocations Works, see Section 2.5.2 on page 52.

3.5.18 omp_free

Summary

The **omp_free** routine deallocates previously allocated memory.

Format

```
void omp_free ( void * ptr, omp_allocator_t *allocator);          (C)
void omp_free ( void * ptr,
                omp_allocator_t *allocator = OMP_NULL_ALLOCATOR); (C++)
```

Effect

The **omp_free** routine deallocates the memory pointed by ptr. The ptr argument must point to memory previously allocated with an OpenMP allocator. If the allocator is specified it must be the allocator to which the allocation request was made. If the allocator argument is **OMP_NULL_ALLOCATOR** the implementation will find the allocator used to allocate the memory. Using **omp_free** on memory that was already deallocated results in unspecified behavior.

C / C++

3.4 Changes to Chapter 5

5.16 OMP_TOOL_LIBRARIES

The **OMP_TOOL_LIBRARIES** environment variable sets the *tool-libraries-var* ICV to a list of tool libraries that will be considered for use on a device where an OpenMP implementation is being initialized. The value of this environment variable must be a comma-separated list of dynamically-linked libraries, each specified by an absolute path.

If the *tool-var* ICV is not enabled, the value of *tool-libraries-var* will be ignored. Otherwise, if **ompt_start_tool**, a global function symbol for a tool initializer, isn't visible in the address space on a device where OpenMP is being initialized or if **ompt_start_tool** returns **NULL**, an OpenMP implementation will consider libraries in the *tool-libraries-var* list in a left to right order. The OpenMP implementation will search the list for a library that meets two criteria: it can be dynamically loaded on the current device and it defines the symbol **ompt_start_tool**. If an OpenMP implementation finds a suitable library, no further libraries in the list will be considered.

Cross References

- *tool-libraries-var* ICV, see Section 2.3 on page 39.
- Tool Interface, see Section 4 on page 364.
- **ompt_start_tool** routine, see Section 4.5.1 on page 396.

5.17 OMP_ALLOCATOR

The **OMP_ALLOCATOR** environment variable defines the memory and allocator traits to be used to create the allocator to be set as the initial value of the def-allocator-var ICV.

The value of this environment variable is a comma-separated list of key=value elements where each key is either a memory or allocator trait and value is one of the allowed values for the spceified trait.

Cross References

- memory and allocator traits, see Section 2.5 on page 50.
- def-allocator-var ICV, see Section 2.3 on page 39.
- **omp_set_default_allocator** routine, see Section 3.5.14 on page 345.
- **omp_get_default_allocator** routine, see Section 3.5.15 on page 346.

4 Examples

The examples presented in the section are intended to demonstrate how the proposed memory management APIs may be used in an OpenMP program. For each example, a C and Fortran version is presented. The example descriptions pertain to the C examples but apply to the corresponding Fortran examples unless otherwise noted. The first set of examples show how to use the APIs to perform dynamic memory allocation using default memory and allocator traits. The next set of examples demonstrate the APIs for explicitly specifying memory and allocator traits for dynamic memory allocation. The examples that follow show how variable declarations can be annotated with the declarative `allocate` directive. The final examples in this section show how allocation for private variables that arise from data-sharing clauses can be managed with the `allocate` clause.

4.1 Basic Allocation

First, we start with examples demonstrating how to use the memory management APIs to perform allocations with the default allocator. In the C example, `OMP_NULL_ALLOCATOR` is passed in to the `omp_alloc` call at line 10 indicating that the default allocator internally maintained by the implementation should be used. In the Fortran example, the same effect is achieved by annotating the `allocate` statement with an `allocate` directive without an `allocator` clause at line 8. Equivalently, the default allocator can be explicitly obtained and used in the code by using the `omp_get_default_allocator` routine.

The memory and allocator traits for the default allocator may be specified by using the `OMP_ALLOCATOR` environment variable or the `omp_set_default_allocator` routine; otherwise, its traits are implementation-defined. For example, suppose `OMP_ALLOCATOR` is set to "`optimized=bandwidth,fallback=abort_fb`" in the environment from which the program is executed and `omp_set_default_allocator` is not used. In this case, the allocation will occur from a bandwidth-optimized memory if it is available or else the program will abort.

C / C++

Example basic.1.c

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   int basic_default1(int n)
S-5   {
S-6       const int success=1, failure=0;
S-7       int retval;
S-8       double *buffer;
S-9
S-10      buffer = omp_alloc(n * sizeof(*buffer), OMP_NULL_ALLOCATOR);
S-11
S-12      if (buffer == NULL) {
S-13          fprintf("Could not allocate space using default allocator\n");
S-14          retval = failure;
```

```
    } else {
        do_work(buffer, n);
        omp_free(buffer, OMP_NULL_ALLOCATOR);
        retval = success:
    }

    return retval;
}
```

C / C++

Fortran

Example basic.1.f

```
function basic_default1(n) result(retval)
    use omp_lib
    integer :: n, retval
    integer, parameter :: success=1, failure=0
    double precision, allocatable :: buffer(:)
    integer :: alloc_status

    !$omp allocate
    allocate(buffer(n), stat=alloc_status)

    if (alloc_status /= 0) then
        print *, "Could not allocate using default allocator"
        retval = failure
    else
        call do_work(buffer, n)
        deallocate(buffer)
        retval = success
    end if
end function basic_default1
```

Fortran

The following examples shows how the proposed API can be used to perform memory allocation using default memory and allocator traits. The effect of using `omp_default_memtraits` is to request that the implementation assumes an implementation-defined set of default traits when selecting a memory for which a memory space object will be returned. The effect of using `omp_default_alloctraits` is to request that the implementation assumes the specified default values for each allocator trait when returning an allocator object, and it is therefore equivalent to setting up an allocator traits set object with zero added traits.

The call at line 12 is guaranteed to return a non-NULL value, and likewise the call at line 13 is guaranteed to return a non-NULL value. The resulting allocator may then be used for default allocations without any traits specified explicitly.

C / C++

Example basic.2.c

```
#include <stdio.h>
#include <omp.h>

int basic_default2(int n)
{
    const int success=1, failure=0;
    int retval;
    omp_memspace_t  *my_mspace;
    omp_allocator_t *my_allocator;
    double *buffer;

    my_mspace    = omp_init_memspace(&omp_default_memtraits);
    my_allocator = omp_init_allocator(my_mspace, &omp_default_alloctraits);
    buffer       = omp_alloc(n * sizeof(*buffer), my_allocator);

    if (buffer == NULL) {
        fprintf("Could not allocate space using default traits\n");
        retval = failure;
    } else {
        do_work(buffer, n);
        omp_free(buffer, my_allocator);
        retval = success:
    }

    omp_destroy_allocator(my_allocator);
    omp_destroy_mspace(my_mspace);

    return retval;
}
```

C / C++

Fortran

Example basic.2.f

```
function basic_default2(n) result(retval)
    use omp_lib
    integer :: n, retval
    integer, parameter :: success=1, failure=0
    integer (kind=omp_memspace_kind)  :: my_mspace
    integer (kind=omp_allocator_kind) :: my_allocator
    double precision, allocatable     :: buffer(:)
    integer :: alloc_status

    my_mspace   = omp_init_memspace(omp_my_memtraits)
```

```
    my_allocator = omp_init_allocator(my_mspace, omp_my_alloctraits)

    !$omp allocate allocator(my_allocator)
    allocate(buffer(n), stat=alloc_status)

    if (alloc_status /= 0) then
        print *, "Could not allocate using default traits"
        retval = failure
    else
        call do_work(buffer, n)
        deallocate(buffer)
        retval = success
    end if

    call omp_destroy_allocator(my_allocator)
    call omp_destroy_memspace(my_mspace)
end function basic_default2
```

Fortran

4.2 Allocation with Traits

In the following examples, the program attempts to allocate out of the memory providing the highest bandwidth while also supporting 2 megabyte pages. At lines 12 through 17, a memory space object is requested with the `bandwidth` trait set to `highest` and the `pagesize` trait set to 2 megabytes. Using the `bandwidth` trait rather than the `optimized` trait means that the memory providing the highest bandwidth while supporting 2MB pages should be used, regardless of whether it is actually designated as "bandwidth-optimized." If the implementation is unable to return such a memory space object since a memory with a 2MB page size is unavailable, a memory space object with default traits is obtained. Next, the program requests an allocator object using the memory space object (pointed to by `mspace`) and default allocator traits.

The allocation is performed at line 19 using the obtained allocator. If the allocator is unable to allocate the requested number of bytes, then the implementation invokes the default fallback behavior – allocating, with default allocator traits, from a memory space with default memory traits. Even with this fallback behavior, it is possible that the allocation is ultimately unsuccessful. In this event the program returns from the function with a `failure` status.

C / C++

Example basic_traits.1.c

```
#include <stdio.h>
#include <omp.h>

int basic_traits1(int n)
{
    const int success=1, failure=0;
    const omp_memtrait_t mtrait_list[2] =
            { {OMP_MTK_BANDWIDTH, OMP_MTV_HIGHEST},
              {OMP_MTK_PAGESIZE, 2*1024*1024} };
    int retval = success;

    omp_memtrait_set_t mtraits;
    omp_init_memtrait_set(&mtraits, 2, mtrait_list);
    omp_memspace_t *my_mspace = omp_init_memspace(&mtraits);
    if (my_mspace == OMP_NULL_MEMSPACE) {
        my_mspace = omp_init_memspace(&omp_default_memtraits);
    }

    omp_allocator_t *my_allocator = omp_init_allocator(my_mspace,
                                                       &omp_default_alloctraits);

    double *buffer = omp_alloc(N * sizeof(*buffer), my_allocator);
    if (buffer == NULL) {
        fprintf(stderr, "Could not allocate using memory allocator\n");
        retval = failure;
    } else {
        do_work(buffer, n);
        omp_free(buffer, my_allocator);
        retval = success;
    }

    omp_destroy_allocator(my_allocator);
    omp_destroy_mspace(my_mspace);

    return retval;
}
```

C / C++

Fortran

Example basic_traits.1.f

```fortran
function basic_traits1(n) result(retval)
    use omp_lib
    integer :: n, retval
    integer, parameter :: success=1, failure=0
    type(omp_memtrait), parameter     :: mtrait_list(2) = &
                (/  omp_memtrait(omp_mtk_bandwidth, omp_mtv_highest), &
                    omp_memtrait(omp_mtk_pagesize, 2*1024*1024) /)
    integer (kind=omp_memtrait_set_kind) :: mtraits
    integer (kind=omp_memspace_kind)  :: my_mspace
    integer (kind=omp_allocator_kind) :: my_allocator
    double precision, allocatable     :: buffer(:)
    integer :: alloc_status

    call omp_init_memtrait_set(mtraits, 2, mtrait_list)
    my_mspace   = omp_init_memspace(mtraits)
    if (my_mspace == omp_null_memspace) then
        my_mspace = omp_init_memspace(omp_default_memtraits)
    end if

    my_allocator = omp_init_allocator(my_mspace, omp_default_alloctraits)

    !$omp allocate allocator(my_allocator)
    allocate(buffer(n), stat=alloc_status)
    if (alloc_status /= 0) then
        print *, "Could not allocate using memory allocator"
        retval = failure
    else
        call do_work(buffer, n)
        deallocate(buffer)
        retval = success
    end if

    call omp_destroy_allocator(my_allocator)
    call omp_destroy_memspace(my_mspace)
end function basic_traits1
```

Fortran

The next examples are similar to the previous ones, except here the program requires that the buffer is either allocated from a bandwidth-optimized (HBW) memory or returns from the function call with a `failure` status. At lines 19 through 22 the program explicitly requests an allocator having a fallback trait with the `null_fb` value. This means that if the allocator is unable to allocate the requested number of bytes at line 29 then a NULL value will be returned and the function will return with a `failure` status.

C / C++

Example basic_traits.2.c

```
#include <stdio.h>
#include <omp.h>

int basic_traits2(int n)
{
    const int success=1, failure=0;
    const omp_memtrait_t mtrait_list[1] =
            { {OMP_MTK_OPTIMIZED, OMP_MTV_BANDWIDTH} };
    omp_memtrait_set_t mtraits;
    omp_init_memtrait_set(&mtraits, 1, mtrait_list);
    omp_memspace_t *hbw_mspace = omp_init_memspace(&mtraits);
    int retval;

    if (hbw_mspace == OMP_NULL_MEMSPACE) {
        fprintf(stderr, "Could not create memspace object for HBW memory\n");
        retval = failure;
    } else {
        omp_alloctrait_set_t atraits;
        const omp_alloctrait_t atrait_list[1] =
        { {OMP_ATK_FALLBACK, OMP_ATV_NULL_FB} };
        omp_init_alloctrait_set(&atraits, 1, atrait_list);
        omp_allocator_t *hbw_allocator = omp_init_allocator(hbw_mspace, &atraits);

        if (hbw_allocator == OMP_NULL_ALLOCATOR) {
            fprintf(stderr, "Could not create allocator object for HBW memory\n");
            retval = failure;
        } else {

            double *buffer = omp_alloc(N * sizeof(*buffer), hbw_allocator);
            if (buffer == NULL) {
                fprintf(stderr, "Could not allocate using HBW memory allocator\n");
                retval = failure;
            } else {
                do_work(buffer, n);
                omp_free(buffer, hbw_allocator);
                retval = success;
            }
            omp_destroy_allocator(hbw_allocator);
        }
        omp_destroy_mspace(hbw_mspace);
    }

    return retval;
}
```

C / C++

Fortran

Example basic_traits.2.f

```
function basic_traits2(n), result(retval)
    use omp_lib
    integer :: n, retval
    integer, parameter :: success=1, failure=0
    type(omp_memtrait), parameter     :: mtrait_list(1) = &
                (/ omp_memtrait(omp_mtk_optimized, omp_mtv_bandwidth) /)
    integer (kind=omp_memtrait_set_kind) :: mtraits
    integer (kind=omp_memspace_kind)  :: hbw_mspace
    type(omp_alloctrait), parameter   :: atrait_list(1) = &
                (/ omp_alloctrait(omp_atk_fallback, omp_atv_null_fb) /)
    integer (kind=omp_alloctrait_set_kind) :: atraits
    integer (kind=omp_allocator_kind) :: hbw_allocator
    double precision, allocatable     :: buffer(:)
    integer :: alloc_status

    call omp_init_memtrait_set(mtraits, 1, mtrait_list)
    hbw_mspace = omp_init_memspace(mtraits)
    if (hbw_mspace == omp_null_memspace) then
        print *, "Could not create memspace object for HBW memory"
        retval = failure
    else
        call omp_init_alloctrait_set(atraits, 1, atrait_list)
        hbw_allocator = omp_init_allocator(hbw_mspace, atraits)

        if (hbw_allocator == omp_null_allocator) then
            print *, "Could not create allocator object for HBW memory"
            retval = failure
        else
            !$omp allocate allocator(hbw_allocator)
            allocate(buffer(n), stat=alloc_status)
            if (alloc_status /= 0) then
                print *, "Could not allocate using memory allocator"
                retval = failure
            else
                call do_work(buffer, n)
                deallocate(buffer)
                retval = success
            end if
            call omp_destroy_allocator(hbw_allocator)
        end if
```

```
        call omp_destroy_memspace(hbw_mspace)
    end if
end function basic_traits2
```

Fortran

4.3 Annotating Variable Declarations

In the following examples, a local array, `scratch`, is declared with length `n` and is used to perform local processing. Memory and allocator traits are explicitly specified on the `allocate` directive for `scratch`. The lifetime of the array is the duration of the call to `process_data`, as it would be if the `allocate` directive was not present. The implementation will therefore take care of performing the implicit deallocation of the array just prior to returning from the function.

C / C++

Example allocate_directive.1.c

```
#include <string.h>
#include <omp.h>

void process_data1(double *dat, size_t n)
{
    double scratch[n];
    #pragma omp allocate(scratch) memtraits(optimized=bandwidth) \
                                  alloctraits(fallback=fb_abort)

    memcpy(scratch, dat, n * sizeof(*dat));
    do_local_work(scratch, n);
    memcpy(dat, scratch, n * sizeof(*dat));
}
```

C / C++

Fortran

Example allocate_directive.1.f

```
subroutine process_data1(dat, n)
    use omp_lib
    double precision :: dat(*)
    integer :: n
    double precision :: scratch(n)
    !$omp allocate(scratch) memtraits(optimized=bandwidth) &
    !$omp&                  alloctraits(fallback=fb_abort)

    scratch(1:n) = dat(1:n)
    call do_local_work(scratch, n)
    dat(1:n) = scratch(1:n)
end subroutine process_data2
```

Fortran

In the next examples, again there is a local scratch array that is followed by an `allocate` directive. This time, an allocator object passed in as an argument is used to allocate `scratch`. The program requires that the local array be allocated in a bandwidth-optimized memory, and if it is unable to do so the program should abort.

C / C++

Example allocate_directive.2.c

```
#include <string.h>
#include <omp.h>

void process_data2(double *dat, size_t n, omp_allocator_t *my_allocator)
{
    double scratch[n];
    #pragma omp allocate(scratch) allocator(my_allocator)

    memcpy(scratch, dat, n * sizeof(*dat));
    do_local_work(scratch, n);
    memcpy(dat, scratch, n * sizeof(*dat));
}
```

C / C++

Fortran

Example allocate_directive.2.f

```fortran
subroutine process_data2(dat, n, my_allocator)
    use omp_lib
    double precision :: dat(*)
    integer :: n
    integer (kind=omp_allocator_kind) :: my_allocator
    double precision :: scratch(n)
    !$omp allocate(scratch) allocator(my_allocator)

    scratch(1:n) = dat(1:n)
    call do_local_work(scratch, n)
    dat(1:n) = scratch(1:n)
end subroutine process_data1
```

Fortran

The next examples show how the `allocator_fb` fallback trait can be used. This time, a pointer to a structure containing user-defined allocators is passed in as an arguments. The `allocate` directive is used to allocate the local array in bandwidth-optimized memory, and if that is not possible it says the array should be allocated as per the allocator pointed to by `allocators->lat_opt`. The calling function, `process_data`, initializes the allocators with a call to `init_allocators` (line 32), and subsequently destroys the allocators with a call to `destroy_allocators` (line 34). It is also necessary to keep track of the memory space objects corresponding to each allocator since the lifetime of an allocator must not extend past the lifetime of its memory space.

C / C++

Example allocate_directive.3.c

```c
S-1  #include <string.h>
S-2  #include <omp.h>
S-3
S-4  struct allocators_t {
S-5      omp_allocator_t *bw_opt;
S-6      omp_allocator_t *lat_opt;
S-7      omp_allocator_t *cap_opt;
S-8      omp_memspace_t  *bw_opt_mspace;
S-9      omp_memspace_t  *lat_opt_mspace;
S-10     omp_memspace_t  *cap_opt_mspace;
S-11 };
S-12
S-13 void process_data3(double *dat, size_t n, struct allocators_t *allocators)
S-14 {
S-15     double scratch[n];
S-16     #pragma omp allocate(scratch) memtraits(optimized=bandwidth) \
S-17                                   alloctraits(fallback=allocator_fb) \
```

```
                                  alloctraits(fb_data=allocators->lat_opt)


    memcpy(scratch, dat, n * sizeof(*dat));
    do_local_work(scratch, n);
    memcpy(dat, scratch, n * sizeof(*dat));
}

void init_allocators(struct allocators_t *allocators);
void destroy_allocators(struct allocators_t *allocators);

void process_data(double *dat, size_t n)
{
    struct allocators_t allocators;
    init_allocators(&allocators);
    process_data3(dat, n, &allocators);
    destroy_allocators(&allocators);
}

void init_allocators(struct allocators_t *allocators)
{
    omp_memtrait_set_t mtraits;
    omp_memtrait_t mtrait_list[1];

    mtrait_list[0].key = OMP_MTK_OPTIMIZED;

    /* create bandwidth-optimized allocator */
    mtrait_list[0].value = OMP_MTV_BANDWIDTH;
    omp_init_memtrait_set(&mtraits, 1, mtrait_list);
    const omp_memspace_t *bw_opt_mspace = omp_init_memspace(&mtraits);
    omp_destroy_memtrait_set(&mtraits);
    allocators->bw_opt_mspace = bw_opt_mspace;
    allocators->bw_opt = omp_init_allocator(bw_opt_mspace,
                                            &omp_default_alloctraits);

    /* create latency-optimized allocator */
    mtrait_list[0].value = OMP_MTV_LATENCY;
    omp_init_memtrait_set(&mtraits, 1, mtrait_list);
    const omp_memspace_t *lat_opt_mspace = omp_init_memspace(&mtraits);
    omp_destroy_memtrait_set(&mtraits);
    allocators->lat_opt_mspace = lat_opt_mspace;
    allocators->lat_opt = omp_init_allocator(lat_opt_mspace,
                                             &omp_default_alloctraits);

    /* create capacity-optimized allocator */
    mtrait_list[0].value = OMP_MTV_CAPACITY;
    omp_init_memtrait_set(&mtraits, 1, mtrait_list);
```

```
    const omp_memspace_t *cap_opt_mspace = omp_init_memspace(&mtraits);
    omp_destroy_memtrait_set(&mtraits);
    allocators->cap_opt_mspace = cap_opt_mspace;
    allocators->cap_opt = omp_init_allocator(cap_opt_mspace,
                                             &omp_default_alloctraits);
}

void destroy_allocators(struct allocators_t *allocators)
{
    omp_destroy_allocator(allocators->bw_opt);
    omp_destroy_memspace(allocators->bw_opt_mspace);
    omp_destroy_allocator(allocators->lat_opt);
    omp_destroy_memspace(allocators->lat_opt_mspace);
    omp_destroy_allocator(allocators->cap_opt);
    omp_destroy_memspace(allocators->cap_opt_mspace);
}

```

C / C++

Fortran

Example allocate_directive.3.f

```
module mo_allocators
    use omp_lib
    type allocators_type
        integer (omp_allocator_kind) :: bw_opt
        integer (omp_allocator_kind) :: lat_opt
        integer (omp_allocator_kind) :: cap_opt
        integer (omp_memspace_kind)    :: bw_opt_mspace
        integer (omp_memspace_kind)    :: lat_opt_mspace
        integer (omp_memspace_kind)    :: cap_opt_mspace
    end type
end module mo_allocators

subroutine process_data3(dat, n, allocators)
    use mo_allocators
    double precision :: dat(*)
    integer :: n
    type(allocators_type) :: allocators
    double precision :: scratch(n)
    !$omp allocate(scratch) memtraits(optimized=bandwidth) &
    !$omp&                  alloctraits(fallback=allocator_fb) &
    !$omp&                  alloctraits(fb_data=allocators%lat_opt)

    scratch(1:n) = dat(1:n)
    call do_local_work(scratch, n)
    dat(1:n) = scratch(1:n)
```

```
end subroutine process_data3

subroutine init_allocators(allocators)
    use omp_lib
    use mo_allocators
    type(allocators_type) :: allocators
    integer (kind=omp_memtrait_set_kind) :: mtraits
    type(omp_memtrait) :: mtrait_list(1)

    mtrait_list(1)%key = omp_mtk_optimized

    ! create bandwidth-optimized allocator
    mtrait_list(1)%value = omp_mtv_bandwidth
    call omp_init_memtrait_set(mtraits, 1, mtrait_list)
    allocators%bw_opt_mspace = omp_init_memspace(mtraits)
    call omp_destroy_memtrait_set(mtraits)
    allocators%bw_opt = omp_init_allocator(allocators%bw_opt_mspace, &
                                           omp_default_alloctraits)

    ! create latency-optimized allocator
    mtrait_list(1)%value = omp_mtv_latency
    call omp_init_memtrait_set(mtraits, 1, mtrait_list)
    allocators%lat_opt_mspace = omp_init_memspace(mtraits)
    call omp_destroy_memtrait_set(mtraits)
    allocators%lat_opt = omp_init_allocator(allocators%lat_opt_mspace, &
                                            omp_default_alloctraits)

    ! create capacity-optimized allocator
    mtrait_list(1)%value = omp_mtv_capacity
    call omp_init_memtrait_set(mtraits, 1, mtrait_list)
    allocators%cap_opt_mspace = omp_init_memspace(mtraits)
    call omp_destroy_memtrait_set(mtraits)
    allocators%cap_opt = omp_init_allocator(allocators%cap_opt_mspace, &
                                            omp_default_alloctraits)
end subroutine init_allocators

subroutine destroy_allocators(allocators)
    use mo_allocators
    type(allocators_type) :: allocators

    call omp_destroy_allocator(allocators%bw_opt)
    call omp_destroy_memspace(allocators%bw_opt_mspace)
    call omp_destroy_allocator(allocators%lat_opt)
    call omp_destroy_memspace(allocators%lat_opt_mspace)
    call omp_destroy_allocator(allocators%cap_opt)
    call omp_destroy_memspace(allocators%cap_opt_mspace)
end subroutine destroy_allocators
```

```
subroutine process_data(dat, n)
    use mo_allocators
    double precision :: dat(*)
    integer :: n
    type(allocators_type) :: allocators

    call init_allocators(allocators)
    call process_data3(dat, n, allocators)
    call destroy_allocators(allocators)
end subroutine process_data
```

Fortran

4.4 Memory Management for Privatized Variables

The following examples illustrate the use of the `allocate` clause. A parallel loop is used to perform an array reduction across rows of a 2-dimensional array, `b`, which has been allocated in bandwidth-optimized memory. Each private copy of the 1-dimensional array, `a`, resulting from the `reduction` clause is allocated according to the `allocate` clause. In this case, the program requests that each thread's private array is also allocated in bandwidth-optimized memory.

C / C++

Example allocate_clause.1.c

```
#include <stdio.h>

#define N 100
void init(int n, float (*b)[N]);

int main()
{
    int i, j;
    float a[N], b[N][N];
    #pragma allocate(a, b) memtraits(optimized=bandwidth)

    init(N, b);

    for (i = 0; i < N; i++) a[i] = 0.0e0;

    #pragma omp parallel for reduction(+:a) private(j) \
                allocate(memtraits(optimized=bandwidth):a)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            a[j] += b[i][j];
        }
```

```
    }

    printf(" a[0] a[N-1]: %f %f \n", a[0], a[N-1]);

    return 0;
}

```

C / C++

Fortran

Example allocate_clause.1.f

```
program array_red
    integer, parameter  :: n=100
    integer             :: j
    real                :: a(n), b(n,n)
    !$omp allocate(a, b) memtraits(optimized=bandwidth)

    call init(n,b)

    a(:) = 0.0e0

    !$omp parallel do reduction(+:a) allocate(memtraits(optimized=bandwidth):a)
    do j = 1, n
        a(:) = a(:) + b(:,j)
    end do

    print *, " a(1) a(n): ", a(1), a(n)
end program
```

Fortran

5 Next steps

This document outlines multiple additions to the OpenMP specification to augment it with an initial modern memory management interface that is capable of supporting the new and future memory technologies but we believe that more features are needed to fully cover all programmer needs. The following are the areas, in no particular order, in which we expect to continue to work targeting the future OpenMP 5.0 specification:

- **Host-device interaction**. The presented mechanisms can be used from within a target region to manage the device memory but do not allow to manage it from the host device. We envision two extensions in this direction:
  1. Allow the **allocate** clause to appear in target directives to affect the device allocations that arise from the **map** clauses.
  2. Extend the API to allow creation of device allocator and allocating memory using these allocators in a similarly to the existing **omp_target_alloc** routine.
- **Predefined trait sets**. We plan to provide a set of standard defined trait sets that encode requirements (e.g., high-bandwidth memory or scratchpad memories) and simplify for common cases of the API usage.
- **NUMA support**. We are exploring mechanisms that allow to distribute memory allocations across the different NUMA domains that could exist in a memory space.
- **Resource querying**. To enable maximum flexibility in looking for the appropriate memory spaces, we plan to develop an API that will allow to query which memory spaces exist in a system (and its attached devices) and which are the traits of each memory space.
- **C++ support**. We acknowledge that the current interface might not blend well with the usages of many C++ programmers and we intend to study how to improve this by providing either additional APIs that work with C++ types such as *std::vector* or redefined C++ operators and allocators.
- **Special code generation support**. Some existing and future memories require compilers to generate different code than for regular memories. Additional directives will be provided to guide the compiler in this process and to allow multiple versions of the same code to exist to work with different memories as necessary.
- **Static allocators**. In some cases in the current proposal we require users to provide an explicit list of traits instead of an allocator. This can get cumbersome and it goes against our principle of moving the decision away from the allocation place as the traits need to be repeated in each allocate directive or clause. To help overcome this problem we envision the ability to fully define allocators at compile time which can the be used in places where a dynamic decision is not possible.