Complex Data Management in OpenACC™ Programs

Technical Report TR-14-1

OpenACC-Standard.org

November, 2014

A preeminent problem blocking the adoption of OpenACC by many programmers is support for user-defined types: classes and structures in C/C++ and derived types in Fortran. This problem is particularly challenging for data structures that involve pointer indirection, since transferring these data structures between the disjoint host and accelerator memories found on most modern accelerators requires deep-copy semantics. Even as architectures begin offering virtual or physical unified memory between host and device, the need to map and relocate complex data structures in memory is likely to remain important, since controlling data locality is fundamental to achieving high performance in the presence of complex memory hierarchies. This technical report presents a directive-based solution for describing the shape and layout of complex data structures. These directives are pending adoption in an upcoming OpenACC standard, where they will extend existing transfer mechanisms with the ability to relocate complex data structures.

This is a preliminary document and may be changed substantially prior to any release of the software implementing this standard.

Contents

1. Introduction

A preeminent problem blocking the adoption of OpenACC by many programmers is support for user-defined types: classes and structures in C/C++ and derived types in Fortran. User-defined types are an important part of many user applications, but the current OpenACC specification only allows for placing top-level variables in data clauses – there is no mechanism for explicitly transferring members of user-defined types. This problem is relatively minor for *flat* data types, which contain only contiguous, fixed-size members, since the data structure can be transferred with one contiguous data transfer using *shallow-copy* semantics. However, this problem is a major limitation for datatypes that contain *disjoint members*, which use pointer indirection, since this kind of data structure cannot be transferred with a single, contiguous transfer. Instead, it requires *deep-copy* semantics. Further, this problem affects nearly all interesting data structures, since it applies to any C/C++ structure with pointer members or any Fortran derived type with allocatable or pointer members. To address this issue this report presents a new set of proposed OpenACC directives and clauses to allow programmers to describe complex data structures, allowing an OpenACC implementation to automatically relocate entire disjoint data structures in memory.

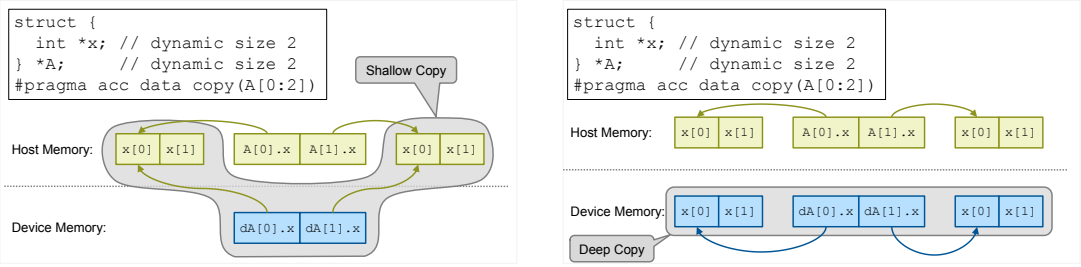
1.1. Problem

Flat data structures can be relocated with a single, contiguous memory transfer because they do not contain pointer indirection, but instead only contain only pure data. Pure data are interpreted the same on the host and device – therefore, an implementation may transfer such data to and from device memory as a simple memory transfer, without regard for the contents of that block of memory.[1]

On the other hand, data structures that contain pointers are interpreted differently on the host and device. A host pointer targets a location in host memory and a device pointer targets a location in device memory; either pointer may be legally dereferenced only on its corresponding device.[2] If an implementation transfers a data structure using a simple transfer, then pointers in that data structure will refer to invalid locations – this transfer policy is referred to as *shallow* copy, which is illustrated in Figure 1.1a. Instead, users often require deep-copy semantics, as illustrated in Figure 1.1b, where every object in a data

[1] Although it is technically possible for a host and device to interpret pure data differently, architectures that follow common conventions and standards (e.g., equivalent endianness, two's complement and IEEE floating point) are compatible.

[2] Pointers are functionally equivalent if a host and device share common memory. Even so, dereferencing such a non-local pointer will likely affect performance due to memory locality; thus, on these devices it may still be desirable to physically transfer memory targeted by pointers.



(a) Shallow copy

(b) Deep copy

Figure 1.1.: Copy semantics

structure is transferred. Deep copy requires recursively traversing pointer members in a data structure, transferring all disjoint memory locations, and translating the pointers to refer to the appropriate device location. This technique is also known as object serialization or marshalling, which is commonly used for sending complex data structures across networks or writing them to disk. For example, MPI has basic support for describing the layout of user-defined data types and sending user-defined objects between ranks [4].

A Fortran compiler could automatically perform deep copy, since Fortran pointers are self-describing dope vectors. In fact, deep copy is the language-specified behavior for intrinsic assignment to a derived type containing allocatable members. Unfortunately, a C/C++ compiler cannot automatically perform deep copy, since C/C++ pointers are raw pointers that do not contain shape information. Further, even for self-describing dope-vectors in Fortran, a user may desire to copy only some subset of deep members. Addressing these problems requires additional information from the programmer.

In C++, the `this` pointer poses an additional challenge. Any reference to a member variable inside of a member function implies a dereference of the `this` pointer, which under normal OpenACC semantics requires an implicit copy of the `this` object. As a result, any transfer of a member variable of pointer type requires deep-copy semantics, an implication that can be particularly confusing to programmers due to the hidden nature of the `this` pointer.

Finally, throughout the report we will distinguish between two types of members for user-defined types: *direct members* and *indirect members*. Direct members store their contents directly in the contiguous memory of the user-defined type. A scalar or statically-sized array member is a direct member. In contrast, *indirect members* store their contents outside of the contiguous memory of the user-defined type, where it is accessed through pointer indirection. Thus, an indirect member always has a corresponding direct member that stores the address or access mechanism of the indirect part. For example, a pointer member in C/C++ is a direct member, but the target of the pointer is an indirect member. In Fortran, an allocatable member is an indirect member, but the underlying dope vector for the allocatable data is a direct member. Since direct members are contained directly in an object, they are automatically transferred as part of that object. But indirect members are not contained directly in an object, and thus require deep-copy semantics to correctly transfer them.

(a) Host memory (b) Device memory

Figure 1.2.: Full deep copy

1.2. Desired capabilities

This chapter introduces several distinct capabilities that we can offer users to improve support for user-defined types with indirect members: member shaping, full deep-copy, selective member deep-copy, selective direction deep-copy, and mutable deep-copy. These various forms of deep-copy require that every sub-object on the device is accessible through a chain of parent objects originating at a single base object – this ensures that addressing calculations and access mechanisms are identical for both host and device code. Additionally, these techniques all assume that individual objects are transferred in whole and have an identical layout in both host and device memory.

Member shapes Allowing users to explicitly shape member pointers puts C/C++ on the same footing as Fortran: pointers become *self describing*. This first step is important because it makes automatic deep-copy possible. However, member shapes aren't useful alone, since they don't imply allocation or transfer – instead, they must be used with other data clauses to achieve deep-copy semantics.

In most cases, member shapes should be global properties of a user-defined type, accessible to all data regions that transfer objects of that type. However, member shapes could certainly be defined as a local property, only affecting data regions at the same program scope.

Full deep-copy With self-describing dope-vectors in Fortran and user-supplied shape declarations in C/C++, an implementation can automatically perform deep copy for user-defined types with indirect members. This approach is known as *full deep-copy*, since it results in a complete duplication of a host data structure in device memory. Figure 1.2 shows a full deep copy from (a) host memory to (b) device memory – the entire data structure is replicated.

In contrast to shallow copy, which only requires replicating an individual object, deep copy requires replicating all sub-objects in a data structure that are accessible from a common base object. Deep copy also requires replicating all object-to-object pointer relationships, which is done by initializing pointer members in the replicated objects.

(a) Host memory (b) Device memory

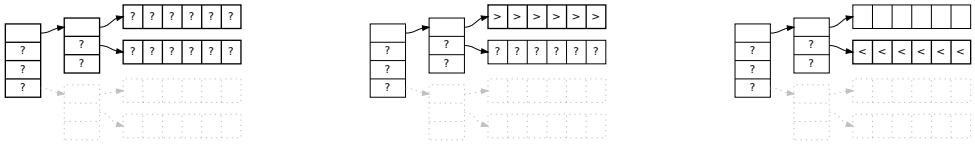
Figure 1.3.: Selective member deep copy

Although we describe deep-copy in terms of `copy`-clause behavior, the concept is applicable to all of the data clauses. Deep `create` replicates all objects in a data structure, but the data members need not be initialized. However, unlike shallow `create`, where no transfer is required, deep `create` requires transfers to properly initialize member pointers. Deep `delete` deallocates all objects in a data structure and deep `update` updates all objects in a data structure. Finally, deep `copyin` and deep `copyout` can be described as combinations of deep `create`, deep `update` and deep `delete`.

The `present` clause could imply either shallow or deep semantics. Shallow `present` implies a present check only for the top-level object in a data structure – it is programmer responsibility to ensure that all required sub-objects are also present and properly accessible through member pointers. On the other hand, deep `present` implies a present check for all objects in a data structure. However, presence alone does not indicate that a data structure is available for use – object-to-object member pointers must also be properly configured. Ensuring that these pointer relationships are valid could be handled by the programmer or the implementation. In either case, member pointers must be properly initialized for the data structure to be usable as expected.

Full deep-copy is most appropriate for user-defined types that represent indivisible collections of members, commonly seen in pure object-oriented programming styles. Such an object is only useful when all or most of its members are available. In this case, full deep-copy can create a complete clone of the object, rendering that object fully usable on the device. As an example, support for member shapes and full deep-copy would make it possible to provide OpenACC-aware versions of the common C++ STL containers.

Selective members Full deep-copy is not appropriate for user-defined types that are used for coarse-grained organizational purposes. These objects are rarely treated as single entities, but instead various subsets of the members are used in different contexts. In this case, it is very inefficient to make a complete clone of the object, since many or most of the members are not needed at one time. Instead, it is preferable to allow users to select different members to include and exclude in different data regions, a capability that we refer to as *selective member* deep-copy. This capability requires a mechanism for specifying different deep-copy policies based on context.



(a) Deep `create` (b) Selective `copyin` (c) Selective `copyout`

Figure 1.4.: Selective direction deep copy

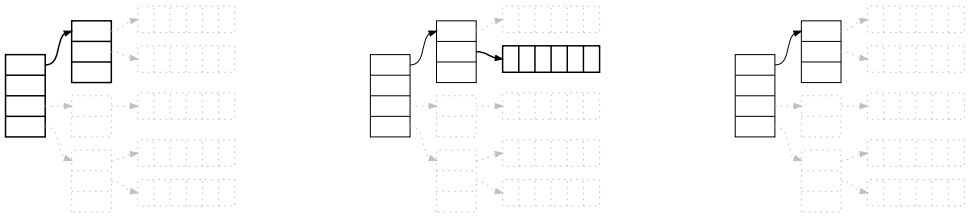
Selective member deep-copy differs from full deep-copy in that some pointers in a host data structure will not be traversed during a transfer, instead leaving invalid pointers in the device data structure. Of course, a user requesting selective deep-copy is responsible for only accessing members that are transferred – accessing a member that is not transferred is considered a programmer error, just like accessing an out-of-bounds value. Figure 1.3 shows selective member deep-copy, where the greyed-out objects are not transferred to device memory. Selective member deep-copy saves device memory and transfer bandwidth by skipping some sub-objects that are not referenced on the device. However, every present sub-object must be accessible through a chain of objects originating at a single base object – this ensures that addressing calculations and access mechanisms are identical for both host and device code.

Selective member deep-copy in an inherently local operation, since different contexts in a program may require different deep-copy behaviors. This differentiation can be achieved in various ways, however. One possible solution is to use local member shapes to shape member pointers differently in different contexts. This requires a means of canceling or overriding any global member shapes that may be visible. Another possible solution is to allow global *named policies* to be declared for a particular user-defined type. Then different contexts may request policies for the user-defined type. Ultimately, the end result is the same for both solutions – a data structure is partially replicated in device memory.

Selective directionality For scalars and arrays of primitive type, a single data-clause behavior (e.g., `create`, `copyin`, `copyout`, etc.) is usually sufficient for an entire object. However, because user-defined types are a collection of individual members, it is reasonable to expect that different members will require different data-clause behavior. For example, consider an object where one member is read-only and another is write-only. Transferring this entire object with a `copy` clause is functionally sufficient, but using a `copyin` clause for the read-only member and a `copyout` clause the write-only member could significantly reduce transfer bandwidth. As a result, it may be useful to provide mechanisms for specifying different members in different data clauses. We refer to this as *selective direction* deep-copy, since different members will transfer in different directions.

Conceptually, selective direction deep-copy is a multi-step process[3], as illustrated in Figure 1.4. First, a data structure is allocated on the device with deep-`create` semantics, only

[3]In practice, the steps at the start of the data region can be merged into a single operation, as can the steps at the end of the data region.



(a) Selective member deep-copy

(b) Attach to parent

(c) Detach from parent

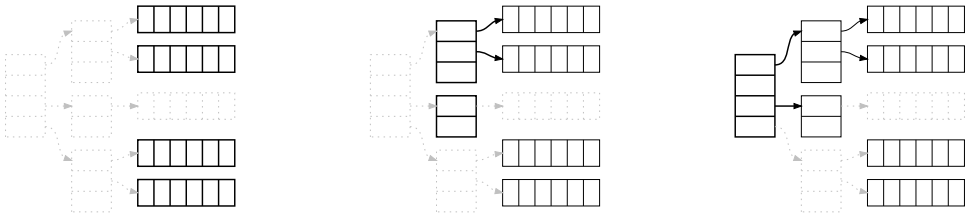
Figure 1.5.: Top-down mutable deep-copy

initializing the required pointers and leaving all other elements uninitialized. This step is shown in Figure 1.4a, with "`?`" labels indicating uninitialized data. Next, the `copyin` sub-objects are transferred to device memory, shown in Figure 1.4b with > labels. Then, at the end of the data region the `copyout` sub-objects are transferred back to host memory, shown Figure 1.4c with < labels. Only the parts of the data structure that are used on the device need to be initialized or transferred – other than for active pointer members, the two parent objects remain uninitialized throughout the data region.

Note that selective directionality is not required for functionality. A user can always use the data-clause behavior that satisfies all references to all members and sub-objects. For example, using a `copy` clause is sufficient for a user-defined object that has one read-only member and another write-only member. However, this approach is coarse grained and may issue unnecessary transfers in both directions.

Moreover, note that selective directionality can always be achieved through `update` directives. Accomplishing this requires two steps: (1) issue a deep `create` to allocate an uninitialized clone of a data structure and (2) issue individual `update` directives to move specific members in the desired direction. When the data structure is no longer needed it will be deep deallocated. The only caveat with using `update` directives is that they are unconditional, so using them with a `present_or_create` clause can be awkward.

Mutable deep-copy Even with selective member and direction, deep copy treats a user-defined object as a single entity, particularly with respect to allocation and deallocation. That is, a base object and all required sub-objects are allocated and deallocated at the same time – the composition of a data structure is immutable throughout its lifetime. However, there may be times when a user does not want to assemble or disassemble a data structure in a single step; instead, they may want to assemble or disassemble it over a period of time, using a series of distinct steps. For example, a data structure may be so large that only a subset of its members fit in device memory at one time. A user may want to copy one group of sub-objects for one part of a data region and another group of sub-objects for another part of the same data region, all while leaving a third group of sub-objects on the device for the entire data region. Because it allows modifying the composition of a data structure during its lifetime, we call this capability *mutable deep-copy*.



(a) Copy sub-objects

(b) Attach to existing children

(c) Attach to existing children

Figure 1.6.: Bottom-up mutable deep-copy

Mutable deep-copy allows a user to *attach* objects to and *detach* them from one another, independently of allocating or transferring those objects. That is, an object may attach to an existing parent object or an existing sub-object. This capability differs from other deep-copy capabilities, which require the composition of a data structure to remain fixed throughout a data region.

Pointer translation for mutable deep-copy can be either top-down or bottom-up. Top-down translation occurs when an object is attached to its existing parent object – this paradigm allows a user to assemble a data structure in a top-down manner. Figure 1.5 illustrates a three-step top-down mutable deep-copy. First, selective deep-copy transfers a base object and a sub-object; then, another sub-object is attached to that data structure; finally, that sub-object is detached from the data structure, leaving only the original base object and single sub-object. In contrast, bottom-up translation occurs when an object is attached to its existing sub-objects – this paradigm allows a user to assemble a data structure in a bottom-up manner. Figure 1.6 illustrates bottom-up mutable deep-copy. First, a set of sub-objects are transferred to the device, independent of their parent objects. Next, another set of sub-objects are transferred to the device, some of which attach to other existing sub-objects. Finally, the top-level base object is transferred to the device and attaches to existing sub-objects.

1.3. Data layout

Programmers often misunderstand the distinction between direct and indirect members. For example, they don't always realize that direct members are automatically included when transferring an object of user-defined type. Instead, they sometimes want to selectively transfer a subset of direct members of a user-defined type, a capability that at first glance appears very similar to selective-member deep-copy. But beneath the surface these capabilities are vastly different.

Because a direct member is contained directly in the contiguous memory of a user-defined type, it is always accessed at a fixed offset from the start of an object. If direct members were allowed to be placed on the device selectively, then any missing direct members would

skew the offsets for subsequent members. We refer to this capability as *data layout*.

Changing the data layout of a variable essentially changes the type of that variable, since it removes some members and affects the offsets and addressing calculations for the remaining members. Although not impossible, supporting data layout for anything more than simple cases is difficult and beyond the scope of this report.

On the other hand, deep-copy applies to indirect members. Every object and sub-object involved in a deep-copy operation is transferred in whole, including all direct members. This preserves equivalent offsets and address calculations between host and device. Selective-member deep-copy simply makes some indirect members unavailable, but the corresponding direct member (i.e., the pointer) is still available, even if it may not be legally dereferenced.

The primary motivation for data layout is for large user-defined types, containing many large direct members. If only a small fraction of those members are needed on the device, then programmers would prefer to save the memory and bandwidth that allocating and transferring them would consume. Although the memory cannot be saved without data layout, the transfer bandwidth can be. For example, an object can always be allocated in full but selectively transferred, allowing the programmer to reduce transfer bandwidth by skipping members that are not needed.

1.4. Relevance

Today OpenACC implementations predominantly target heterogeneous systems comprising independent host and device processors with distinct memory available to each – all communication between host and device processors must be performed through explicit data transfers. In this environment directive-based deep-copy is a functional requirement for supporting disjoint data structures in OpenACC; without it, data structures or applications must be rewritten to work with OpenACC.

However, industry trends suggest that future system designs will relax the strict functional need for deep-copy. For example, CUDA [5] and HSA [1] provide some form of *unified memory*, which allows software to treat physically distinct memories as one logical memory with a shared virtual address space. In this environment deep-copy is no longer necessary for functionality, since a pointer is treated the same on both host and device. That is, dereferencing a pointer that targets a location in the unified address space produces the same functional behavior regardless of whether the dereference occurs on the host or device. In theory data transfers can become no-ops in OpenACC, allowing complex data structures to be easily shared between host and device. Indeed, this capability is a very compelling solution for porting applications to OpenACC. However, this approach can severely limit performance due to locality implications of the distinct physical memories. Thus, once functionality is established and performance optimization is desired, developers will need to use explicit data transfers to improve data locality with respect to computation locality. Just as the directives described in this report will allow transferring complex data structures between distinct physical memories for functionality, they will also allow relocating complex data structures within a unified address space for performance.

Heterogeneous systems that combine fast scalar processors with relatively slow parallel processors face a trade off in memory system design. A fast scalar processor performs best with a very low memory latency, but a massively parallel processor performs best with very high memory bandwidth. As a result, most heterogeneous systems designed within reasonable cost parameters are likely to employ distinct physical memories, each tailored to the type of processor which they serve. Thus, achieving high performance on these kinds of systems will require careful attention to data locality.

It is likely that even homogeneous systems of the future will face similar design trade offs, especially as memory bandwidth requirements track the growth of on-chip parallelism. An ideal system would provide large-capacity, high-bandwidth memory; but, a realistic system will provide limited-capacity, high-bandwidth memory paired with high-capacity, conventional memory. In some sense this trend is already here, in the form of a hierarchy of disk storage, main memory, processor cache, and registers. But this hierarchy is likely to expand to include new levels of main memory and secondary storage. Again, achieving high performance will require careful attention to data locality, in some cases requiring initial placement and subsequent relocation of complex data structures within the memory hierarchy.

Finally, the the new directives and clauses presented in this report apply to member of user-defined types, allowing programmers to describe shapes and relationships of objects in their data structures. This capability can be applied beyond deep-copy. For example, OpenMP currently defines shallow-copy semantics for privatization of pointers and pointer members [6]. But if the data types are properly annotated then an entire data structure could be automatically privatized by a compiler.

2. Requirements

After the completion of OpenACC 2.0, the dominant user request heard by the OpenACC language committee has been support for deep copy. This chapter presents high-level survey results illustrating the deep-copy requirements for a range of HPC applications, followed by a deeper analysis of two real-world examples (one Fortran and one C++) where deep-copy is required for porting to OpenACC.

2.1. User survey results

We conducted a survey of applications that run on different HPC centers (NCCS, CSCS, TU-Dresden, etc.) and found that the majority of applications are written in C++ and Fortran, as shown on Figure 2.1.

We found that 65% of applications use data structures with single levels of pointers. That means data structures that contain pointers to memory regions. 55% of the applications contains pointers to other data structures with multiple levels of pointers. A small number of applications use recursive data types, pointers to unions, etc. as shown in Figure 2.2.

From the survey, 60% of the users want to use sibling members to describe the shape of the pointers, 35% wants to use functions, and 25% want to use global variables as shown in Figure 2.3.

We also found that the majority of the users want a mechanism to toggle the default of full recursive deep copy or shallow copies of data structures as show in Figure 2.4.

For C++ applications we found that 35% use STL vectors, 15% map, etc. We also found that STL complex type is used in scientific applications, as shown in Figure 2.5.

2.2. Fortran: ICON

ICON, a climate code developed by the German Weather Service (DWD) and the Max Planck Institute for Meteorology (MPI-M) [2], uses derived types that must be made available on the device.

For example, one derived type is `t_nh_state`, declared in Figure 2.6. The shallow size of this derived type is roughly 19KB, but the deep size of this derived type is much larger because the member `metrics` contains roughly 84 pointer/allocatable members and the member `diag` contains roughly 86 pointer/allocatable members. Moreover, only the members depicted in Figure 2.7 are needed on the device at one time.

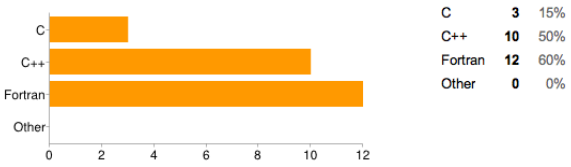


Figure 2.1.: Programming languages used at different HPC centers

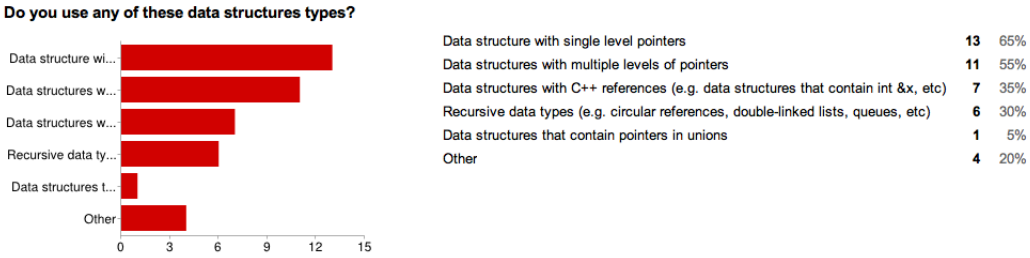


Figure 2.2.: Types of data structures used in applications

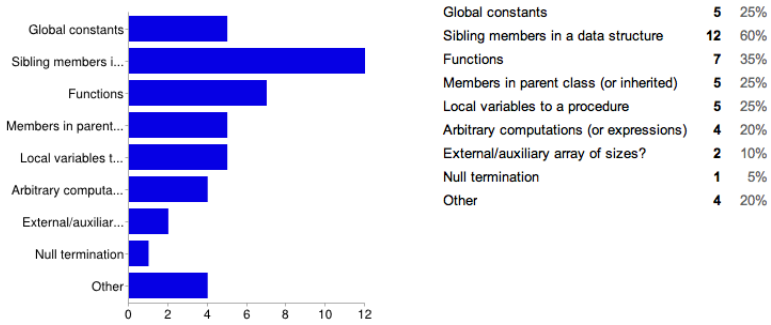


Figure 2.3.: How applications would like to describe the shape of pointers

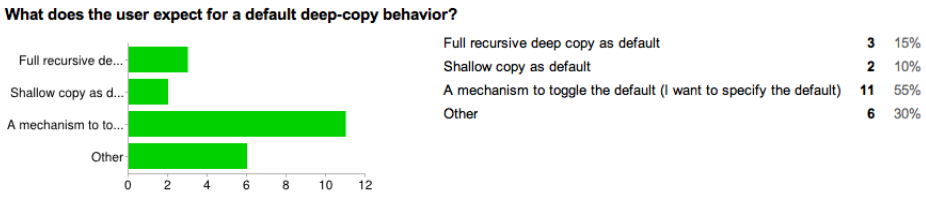


Figure 2.4.: Programming languages used at different HPC centers

Because a relatively small number of pointer/allocatable members are actually referenced on the device, this data structure is a good candidate for selective-member deep-copy – OpenACC developers of this code agree with this assessment. Full deep-copy is a functionally suitable option, though it will result in unnecessary memory allocation and data transfers.

All of the members referenced represent a single level of indirection except for `p_nh_state(:)%prog(:)%vn(:,:,:)`, which represents a selective deep copy of derived type `type(t_nh_prog)`. This size of this type is roughly 1KB per element, plus the additional selective deep copy of member `vn`. Selective directionality could be helpful for this code, since some members are read-only and others are read-write.

Additionally, this data structure contains many *convenience pointers*, which are designed to alias other members in the data structure. This aliasing introduces an ordering complexity, since the aliases must be processed after the members to which the aliases point.

Finally, the ICON data structure contains embedded linked lists. Fortunately, the developers indicate that these linked lists are not currently needed on the device, since they are only used for I/O. Even so, it is not unreasonable to expect that other applications will require transferring recursive data structures to an accelerator. As a result, it is prudent to keep this use-case in mind when designing deep-copy support.

2.3. C++ STL vector

The most commonly reported obstacle for porting C++ applications to OpenACC is lack of support for the Standard Template Library (STL) containers, particularly `std::vector` [3]. Although the `std::vector` isn't an application itself, we consider it a very important real-world example because it is so widely used in C++ applications and it exhibits all of the same deep-copy challenges that appear in application codes. Conceptually this data structure is quite simple, but because it is defined as a class with pointer member variables transferring it to device memory requires deep-copy semantics. This section identifies the challenges with supporting `std::vector` in OpenACC.

The STL defines a standard interface for vector, but it does not define a standard implementation. In theory, different vendors can provide different implementations that have different deep-copy requirements. However, since many vendors have adopted the open-source GNU implementation for STL containers we will examine that implementation in detail. In practice, the deep-copy requirements will be similar for all implementations. The code in Figure 2.8 illustrates the relevant parts of the GNU `std::vector` declaration.

The top-level `std::vector` class doesn't itself contain any member variables, but it inherits from the `_Vector_base` class. The `_Vector_base` class contains a single public member variable, `_M_impl`, which is a class-private struct named `_Vector_impl`. It is within `_Vector_impl` that we find three member variables, all of type pointer: `_M_start`, `_M_finish`, and `_M_end_of_storage`. These pointer member variables cause the data structure to require deep-copy semantics.

A vector holds a dynamically-sized contiguous array of elements, and the three pointers alias different offsets into this array: `_M_start` points to the first element, `_M_finish`

points to the next element after the last one in use, and `_M_end_of_storage` points to the next element after the end of reserved space (i.e., the first out-of-bounds element). Thus, deep-copy semantics require transferring the contiguous array of elements once and then translating all three pointers with respect to the array of elements. The main complication with translating the `_M_end_of_storage` pointer is that it targets an out-of-bounds memory location that is not allocated on either the host or device.

There is a distinction between the last valid element and the end of storage to allow for appending additional elements without triggering a reallocation – this behavior is required by the STL specification to guarantee that an append operation will complete in amortized constant time. As a result, a deep-copy transfer of `std::vector` may either transfer the valid elements or the reserved elements. The appropriate choice depends on the way in which an application uses the vector – a vector that is not resized on the device needs to only transfer the valid elements, while a vector that is extended on the device needs to transfer (or at least allocate) the reserved elements. A reasonable default may be to transfer the reserved elements, but to allow a mechanism for transferring only the valid elements. More generally, some applications may only need to transfer a subset of the valid elements – in this case it would be useful to parameterize the deep-copy behavior with an array slice.

Another challenge with supporting deep-copy for `std::vector` is that it is defined as a template type, where the template parameter determines the array element type. This is particularly challenging if the template type itself is one that requires deep-copy, such as a pointer or class containing a pointer. Supporting this kind of data structure requires that the deep-copy behavior be supported and defined for the template type. This could be accomplished by defining a default deep-copy behavior for the template type; or, it could be accomplished by parameterizing the `std::vector` deep-copy behavior with the deep-copy behavior desired for the template type. Fortunately, most uses of `std::vector` in high-performance computing use basic data types for the template parameter, ones that don't require deep-copy.

From a programmer's perspective, the `std::vector` declarations are provided in a read-only header file. If a programmer wants to define a deep-copy behavior for `std::vector` then it must be placed in an application header file, outside of the official declaration of `std::vector`. However, because the internal implementation details of `std::vector` are private (and may differ between vendors), it may not even be reasonable to expect a programmer to define the deep-copy behavior. Instead, the OpenACC spec should define the expected deep-copy behavior for `std::vector` and vendors should ensure that their particular implementation conforms to the spec. In essence, the author of an opaque data type should define the deep-copy behavior for it, allowing that behavior to be encapsulated within the data type.

A final challenge with supporting `std::vector` in OpenACC is handing operations that resize the array of elements, mainly because resizing can trigger a reallocation. This challenge is not specific to deep-copy – OpenACC does not currently support reallocating an object while it is mapped in an active data region. As long as this limitation persists, `std::vector` may not be resized on the device[1]

[1] This limitation does not apply to vectors that are entirely local to the device, which may be freely allocated and resized.

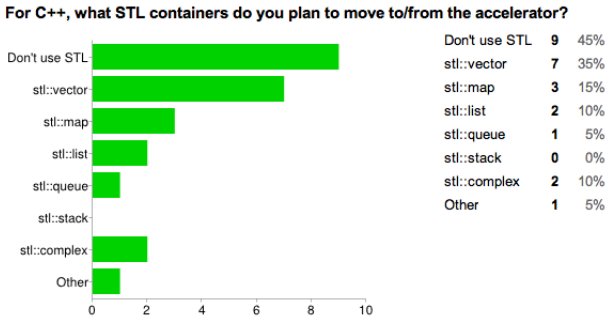


Figure 2.5.: STLs used in C++ applications

```
TYPE t_nh_state

  !array of prognostic states at different timelevels
  TYPE(t_nh_prog),  ALLOCATABLE :: prog(:)       !< shape: (timelevels)
  TYPE(t_var_list), ALLOCATABLE :: prog_list(:)  !< shape: (timelevels)

  TYPE(t_nh_diag)     :: diag
  TYPE(t_var_list)    :: diag_list

  TYPE(t_nh_ref)      :: ref
  TYPE(t_var_list)    :: ref_list

  TYPE(t_nh_metrics) :: metrics
  TYPE(t_var_list)    :: metrics_list

  TYPE(t_var_list), ALLOCATABLE :: tracer_list(:) !< shape: (timelevels)

END TYPE t_nh_state
```

Figure 2.6.: ICON data structure

```
p_nh_state(:)%metrics%rayleigh_w(:)
p_nh_state(:)%metrics%rayleigh_vn(:)
p_nh_state(:)%diag%vn_ie(:,:,:)
p_nh_state(:)%diag%vt(:,:,:)
p_nh_state(:)%diag%dvn_ie_ubc(:,:)
p_nh_state(:)%diag%e_kinh(:,:,:)
p_nh_state(:)%diag%w_concorr_c(:,:,:)
p_nh_state(:)%prog(:)%vn(:,:,:)
```

Figure 2.7.: ICON device members

```
namespace std {
  template<typename _Tp, typename _Alloc>
  struct _Vector_base
  {
    typedef typename _Alloc::template rebind<_Tp>::other _Tp_alloc_type;
    struct _Vector_impl : public _Tp_alloc_type
    {
      typename _Tp_alloc_type::pointer _M_start;
      typename _Tp_alloc_type::pointer _M_finish;
      typename _Tp_alloc_type::pointer _M_end_of_storage;
      // ... no more member variables
    };
  public:
    _Vector_impl _M_impl;
    // ... no more member variables
  };

  template<typename _Tp, typename _Alloc = std::allocator<_Tp> >
  class vector : protected _Vector_base<_Tp, _Alloc>
  {
    // ... no more member variables
  };
}
```

Figure 2.8.: `std::vector` declarations

3. Directives

This chapter introduces proposed new directives and clauses for supporting deep-copy in OpenACC. The syntax is designed in a graduated manner – simple behavior can be achieved with simple syntax, while more sophisticated behavior requires more complex syntax. This approach frees programmers from needing complex syntax for common functionality, but it gives them the option to employ the more complex syntax when the added expressiveness is necessary.

The simple *shape* syntax allows defining global deep-copy behavior for all objects of a particular type, but it is limited to full and selective-member deep-copy. The *policy* syntax enables defining global selective direction behavior for all objects of a particular type (see Section 1.2 for a description of these capabilities). Next, building on the shape and policy syntax, the *nest* syntax allows defining local deep-copy behavior, which applies to one or more specific objects in a data region. Finally, the *convenience* syntax provides a very intuitive shorthand for the nest syntax, but it only applies to simple use cases.

3.1. Shape

OpenACC already provides a syntax for shaping arrays and pointers. This syntax is used to specify what elements are being referred to in the clause. For example, `copy(p[0:5])` specifies to copy the first 5 elements of the array `p`. This section introduces a syntax for specifying which members are being referred to for user-defined types. The syntax allows expressing which members are needed on the device as well as providing shapes for those members.

3.1.1. Shape directive

Summary The shape directive is used to specify which members to include when processing data or update directives and providing a shape for any members that require one. A shape directive is associated with a user-defined data type, and it may appear either inside or outside of the declaration for that data type. When specified outside of a type declaration, a `type` clause is used to specify the associated data type. Data clauses on the shape directive apply to members of the associated type. The shape directive provides full deep-copy and selective-member capabilities (see Section 1.2).

Syntax In C and C++, the syntax of the OpenACC shape directive is

`#pragma acc shape`[`(` *shape-name-string* `)`] *shape-clause-list*

and in Fortran, the syntax is

```
!$acc shape[( shape-name-string )] shape-clause-list
```

where *shape-clause* is one of the following:

```
type( type-specification )
include( member-shape-list )
exclude( member-shape-list )
init_needed( member-shape-list )
default( none | include | exclude )
```

Description The shape directive is used to specify which members to include when processing data or update directives and providing a shape for any members that require one. When an object matching the data type appears in a data clause, the shape defined for that data type determines the behavior of the transfer. A member that is included in the shape will be made available on the device, using the same data clause that the containing object appears in. A member that is excluded is not available on the device. Accessing an excluded member on the device will result in unspecified behavior.

When an object matching the data type appears in an update clause, the shape defined for that data type determines the behavior of the update. By default a member that is included will be updated. Members that are excluded will not be updated.

If an explicit default shape is not provided for a type, an implicit default shape is used. This implicit shape is equivalent to:

```
#pragma acc shape default( include ) (C/C++)
```

```
!$acc shape default( include ) (Fortran)
```

A shape directive without a name becomes the default shape of the type. Only one default shape may be specified for a type.

Type clause The `type` clause is used to specify the data type to which a shape is associated, which is useful when the shape directive cannot be placed in the type declaration (e.g., the header file containing the type declaration may be read only). If a shape directive appears within a type declaration then the directive has an implicit type clause specifying the nearest enclosing type declaration, making the type clause optional. A `type` clause may still be provided, but it must specify the nearest enclosing type or a type that is declared within the nearest enclosing type. If a shape directive appears outside of a type declaration then an explicit `type` clause is required. At most one `type` clause is allowed per shape directive.

Include clause The `include` clause is used to specify which members to include when processing objects of this type. It can also be used to provide a shape for members.

Init_needed clause The `init_needed` clause is a special form of the include clause. It specifies both that the members are included, and that they require initialization when used in a data clause. A member that is specified in an `init_needed` clause will be treated as `copyin` when processed for a `create` clause and `copy` when processed for a `copyout` clause.

Default clause The `default` clause is used to specify whether members are treated as `include` or `exclude` if they are not specified in any clause in the shape directive. If a shape directive does not specify an explicit `default` clause then it receives an implicit `default(include)` clause.

3.1.2. Specifying a non-default shape

A shape directive that specifies a shape name is considered a non-default shape. This shape will only be used when explicitly requested. The syntax for requesting a non-default shape is:

clause`<`*shape-name-string*`>`

This syntax is allowed on any data clause, update clause, include clause, and init_needed clause. A non-default shape extends a default shape, overriding the inclusion or exclusion of members that appear in both the default and non-default shape. See Figures A.1, A.2 and A.4 in Appendix A for examples of how to use the `shape` directive.

3.2. Policy

Although the shape syntax provides good encapsulation and simplicity for management of user defined types, it applies the same top-level data clause behavior (`create`, `copyin`, ...) to all included members, recursively. That is, it supports selective-member semantics, but not selective-direction (see Section 1.2). This section introduces a syntax for specifying more complex policies that allow the behavior of each member to be controlled independently.

3.2.1. Policy directive

Summary The policy directive is used to control the behavior of each member. This behavior can be either data clauses (in the case of a data policy) or update clauses (in the case of an update policy). Just like with the shape directive, the policy directive is associated with a user-defined data type, and it may appear either inside or outside of the declaration for that data type. When specified outside of a type declaration, a `type` clause is used to specify the associated data type. Data clauses on the policy directive apply to members of the associated type.

Syntax In C and C++, the syntax of the OpenACC policy directive is

`#pragma acc policy(` *policy-name-string* `)` *policy-clause-list*

and in Fortran, the syntax is

`!$acc policy(` *policy-name-string* `)` *policy-clause-list*

Where *policy-clause* is one of the following:

`type(` *type-specification* `)`
`use(` *policy-name-string-list* `)`
`shape(` *shape-name-string* `)`
`exclude(` *member-shape-list* `)`
data-policy-clause
update-policy-clause

Where *data-policy-clause* is one of the following:

data-clause`(` *member-shape-list* `)`
`default( none | copy | copyin | copyout | create | present | delete | exclude )`

Where *update-policy-clause* is one of the following:

`update(` *member-shape-list* `)`
`default( none | update | exclude )`

Description The policy directive is used to control the behavior of each member. This behavior can be either data clauses (data policy) or update clauses (update policy).

The behavior of a policy always builds upon a shape. The default shape will be used unless a `shape` clause appears, in which case the shape policy named in the `shape` clause will be used. The policy may override the inclusion or exclusion of a member, as specified by the shape, with the exclude clause or by listing an excluded member in a data or update clause.

When a policy is used within a data directive it is called a data policy. The data policy specifies the data direction to use for each member and/or to exclude a member. This provides selective direction (see Section 1.2).

When a policy is used within an update directive it is called an update policy. The update policy specifies which members are updated and which are excluded from the update.

Type clause The `type` clause is used to specify the data type a policy is associated with, which is useful when the policy directive cannot be placed in the type declaration (e.g., the header file containing the type declaration may be read only). If a policy directive appears within a type declaration then the directive has an implicit type clause specifying the nearest enclosing type declaration, making the type clause optional. A type clause may still be provided, but it must specify the nearest enclosing type or a type that is declared within the nearest enclosing type. If a policy directive appears outside of a type declaration then an explicit `type` clause is required. At most one `type` clause is allowed per policy directive.

Use clause The `use` clause is used to specify another policy that should be invoked when the current policy is used. This allows policies to be composed from other policies. The behavior of a member in the current policy will override the behavior in any used policies.

Shape clause The `shape` clause is used to specify the shape to use for this policy. If no `shape` clause is specified then the default shape is used. At most one `shape` clause is allowed per policy directive.

Exclude clause The `exclude` clause is used to specify which members are not included in this policy. A member excluded in a data policy may not be accessed on the device. A member excluded in an update policy will not be updated.

Data clause A data clause is used to specify the direction for members in a data policy. Any data clause may be specified in a policy. A data clause cannot appear in an update policy.

Update clause The update clause is used to specify which members should be updated. An update clause cannot appear in a data policy.

Default clause The `default` clause is used to specify the default behavior of a member that does not appear in any other clause. This behavior will only apply to members specified in an `include` clause in the shape. Members in an `exclude` clause in the shape will by default still be excluded. The values allowed to appear in a default clause are different for data policies and update policies. A directive may have at most a single `default` clause. If a `default` clause is not specified on a directive then it receives an implicit `default(none)` clause.

3.2.2. Using a data policy

The behavior defined by a data policy is triggered by using the new `invoke` data clause:

`invoke<`*policy-name-string*`>(` *var-list* `)`

The `invoke` clause with a data policy can be specified wherever other data clauses are allowed including `data`, `end data`, `exit data` directives. It is an error for a variable to appear in an `invoke` clause if its type does not support a policy with the specified name.

It is intentional that the `invoke` clause does not imply a specific data clause or transfer direction. Instead, it specifies a policy, which effectively defines a custom transfer behavior. Because a policy has full control to specify explicit data clauses for all members in a user-defined type, possibly expressing selective-direction behavior, it does not make sense to apply a policy to an object that appears in an ordinary data clause. This differs from the way shapes are used. Shapes are limited to `include` and `exclude` clauses, which express member selection without a direction. Thus, the direction to use when applying a shape must come from the top-level data clause. In summary, shapes are applied to objects appearing in ordinary data clauses while policies are applied to objects appearing in an `invoke` clause. For shapes, the direction is defined by the top-level clause, not the shape; for policies, the direction is defined by the policy, not by the top-level clause.

3.2.3. Using an update policy

The behavior defined by an update policy is triggered by using a new update clause:

`invoke<`*policy-name-string*`>(` *direction*`:`*var-list* `)`

Where *direction* is one of the following:

```
self
device
```

The `invoke` clause with an update policy can be used on `update` directives. It is an error for a variable to appear in an invoke clause if its type does not support a policy with the specified name.

3.2.4. Using a policy for a member

The behavior defined by a data or update policy can be triggered for a member variable by using a new clause:

`invoke<` *policy-name-string* `>(` *member-shape-list* `)`

The invoke clause for a member can be specified on the policy directive.

It is an error for a member to appear in an invoke clause if its type does not support a policy with the specified name.

See Figures A.3 and A.5 in Appendix A for examples of how to use the `policy` directive.

3.3. Local nested syntax

Although the global shape and policy directives (see Sections 3.1 and 3.2) have advantages of code reuse and encapsulation, there are some cases where users might prefer to request deep-copy semantics locally, for a single data region. This section introduces syntax for specifying nested data clauses, which allow expressing deep-copy semantics within a single directive. Additionally, this local nested syntax can be treated as a common, low-level specification syntax into which global policy syntax can be lowered. This is useful for formally defining the behavior of policy syntax as well as for implementing it.

The nested syntax essentially allows defining anonymous shapes or policies that apply to one or more objects appearing in a data clause. The extensions for nested syntax are relatively simple – an anonymous shape is defined by following a data clause with a nest containing a list of data clauses:

data-clause[<[*shape-name-string*]>]`(`*var-list*`)`[`::{`*nest-clause-list*`}`]

where *data-clause* is one of the following:

```
copy
copyin
copyout
create
present
delete
include
self
device
```

When a nest follows a data clause all variables appearing in *var-list* must have the same data type. The nest defines a local shape for that particular data type; the local shape is active only for the current directive and applies only to the variables appearing in the data clause. A *nest-clause-list* may contain all of the clauses that are allowed on a `shape` directive, except the `type` clause:

```
include( member-shape-list )
exclude( member-shape-list )
init_needed( member-shape-list )
default( none )
default( include )
default( exclude )
```

Just as for clauses on the `shape` directive, clauses appearing in a nest apply to member variables of the associated datatype. In addition to local and global variables visible at the location of the data directive, shape expressions in a nest may also refer to member

variables or member functions of the type associated with the nest. That is, the scope of a reference is first resolved as though it appears in a member function of the datatype associated with the nearest enclosing nest, and if that fails then it is resolved in the usual manner (as though it appears at the location of the data directive).

An anonymous shape may be defined for a data type that has an existing default shape – the local shape simply extends the default shape and overrides the behavior for members that appear in both shapes. Similarly, a data clause with a nest may also specify a named shape, in which case the anonymous shape simply extends the named shape instead of the default shape. Finally, the shape name is optional – if the shape-name brackets are specified but the shape name is omitted (i.e., *data-clause*`<>::{`*nest-clause-list*`}`), then the anonymous shape expresses a completely new shape, without extending any global shapes (default or named).

Very similar to anonymous shapes, an anonymous policy is defined by following an `invoke` clause with a nest containing a list of data clauses:

`invoke<`[*policy-name-string*]`>(`*var-list*`)`[`::{`*nest-clause-list*`}`]

When a nest follows an `invoke` clause all variables appearing in *var-list* must have the same data type. The nest defines a local policy for that particular data type; the local policy is active only for the current directive and applies only to the variables appearing in the `invoke` clause. A *nest-clause-list* may contain all of the clauses that are allowed on a `policy` directive, except the `type` clause:

`use(` *policy-name-string-list* `)`
`shape(` *shape-name-string* `)`
`exclude(` *member-shape-list* `)`
data-policy-clause
update-policy-clause

Data clauses appearing in a nest may also have nests, if the variables appearing in the nested data clause also have aggregate type. This definition allows expressing deep-copy behavior for any non-cyclic data structures with a fixed depth from a single root object.

See Figures A.6 and A.7 in Appendix A for examples of how to create anonymous shapes and policies.

3.4. Convenience syntax

The local nested syntax provides a powerful tool for describing arbitrary shapes and policies that are local to a data region. To achieve this generality and expressiveness, local nests use a relatively complex syntax. For some simple cases, a more natural and intuitive syntax is possible. This syntax is provided as a convenience to the user and does not provide any additional functionality beyond what is possible with the local nested syntax. It is simply a more natural syntax for simple use-cases.

The convenience syntax allows specifying member variables directly in data clauses, either with an explicit base object (using the member syntax provided by the programming language) or an implicit base object of the `this` variable (in C++). A base may be any of the following:

- a simple scalar variable
- implicit `this` variable (in a C++ member function)

A base may be appear multiple times in the same directive, even in different clauses, allowing multiple members to be specified. Members can be shaped, if applicable.

The convenience syntax generates an equivalent anonymous local policy for the base variable. If the base variable itself appears in a data clause then that clause determines the `default` clause for the local policy; otherwise the local policy receives a `default(exclude)` clause. Each member that appears in a data clause with convenience syntax generates an equivalent nested clause in the local policy.

In C and C++, the convenience syntax is

data-clause`(`*base-variable*`.`*member-shape*`)`

and in Fortran, the syntax is

data-clause`(`*base-variable*`%`*member-shape*`)`

For all languages the convenience syntax produces the following nest syntax

`invoke<>(`*base-variable*`)::{ default(exclude)` *data-clause*`(`*member-shape*`) }`

The convenience syntax generalizes to allow the *base-variable* itself to appear in a data clause, and for it to appear multiple times for different *member-shapes* (possibly in different *data-clauses*). In C and C++ the generalized convenience syntax is (with clauses appearing in any order)

base-data-clause`(`*base-variable*`)`
data-clause-1`(`*base-variable*`.`*member-shape-1*`)`
data-clause-2`(`*base-variable*`.`*member-shape-2*`)`
...
data-clause-n`(`*base-variable*`.`*member-shape-n*`)`

and in Fortran, the syntax is (with clauses appearing in any order)

base-data-clause`(`*base-variable*`)`
data-clause-1`(`*base-variable*`%`*member-shape-1*`)`
data-clause-2`(`*base-variable*`%`*member-shape-2*`)`
...
data-clause-n`(`*base-variable*`%`*member-shape-n*`)`

For all languages the generalized convenience syntax produces the following nest syntax

```
invoke<>(base-variable)::{
    default(base-data-clause)
    data-clause-1(member-shape-1)
    data-clause-2(member-shape-2)
    ...
    data-clause-n(member-shape-n)  }
```

Finally, the convenience syntax supports an implicit base variable of the C++ `this` pointer. The following convenience syntax is allowed in a C++ member function

```
data-clause-1(member-shape-1)
    data-clause-2(member-shape-2)
    ...
    data-clause-n(member-shape-n)
```

The C++ member function convenience syntax lowers to the following nest syntax

```
invoke<>(this[0:1])::{
    default(exclude)
    data-clause-1(member-shape-1)
    data-clause-2(member-shape-2)
    ...
    data-clause-n(member-shape-n)  }
```

See Figure A.8 in Appendix A for examples of how to use the convenience syntax.

3.5. Pointer translation

This section introduces new syntax for expressing pointer aliases, which must be translated for use on the device. This functionality is useful beyond deep-copy, but we include it in this report because it is needed for supporting a particular paradigm seen in common implementations of `std::vector`.

Pointer aliasing is expressed with the `present` clause, since by definition an alias is a reference to an existing location – that is, the target object must already exist on the device. This new syntax allows specifying the symbol `@` instead of providing an explicit pointer shape:

```
present(var[@[base-var]]) (C/C++)
present(var(@[base-var])) (Fortran)
```

The `@` symbol indicates that the pointer targets an object that is already present on the device – the lookup is performed and the pointer is translated for use on the device. For simple cases this is equivalent to a `present` clause on a pointer with an explicit shape, as long as the entire slice is indeed present (otherwise an absent or partially present slice indicates a runtime error). Thus, for simple cases the `@` symbol syntax is essentially a convenience for eliding a shape, instead implying a minimal shape of one byte.

A more compelling use case for the `@` syntax is for *out of bound* aliases, which commonly occur with iterators in C++. A `start` iterator targets the first element in a range while an `end` iterator targets one past the last element in a range. In this case two pointers logically alias the same underlying range of elements. The elements should be transferred once, but both pointers need to be translated. However, we cannot naively translate the `end` iterator, since it actually references one past the last element and is therefore not present. To address this case the `@` syntax allows specifying an optional *base-var* pointer that is present, indicating that pointer translation should be made with respect to the specified base. By specifying a relative base pointer we can now express out-of-bounds pointer aliases: `present(end[@start])`.

Another interesting use case for the `@` syntax is for pointer-to-pointer structures in C/C++. Consider an array of pointers that all alias elements in an already present array, a data structure often used for sparse-matrix representations. The array of pointers may be transferred with an explicit, though slightly misleading shape: `copyin(ptrs[0:10][0:1])`. The `@` syntax provides a much cleaner way to express the aliases: `copyin(ptrs[0:10][@])`. In this case the `@` syntax is allowed within a data clause other than `present` because the double indirection of pointers really implies a deep-copy. So, dimensions with explicit shapes receive `copyin` semantics, but dimensions with explicit aliasing receive `present` semantics.

4. Semantics

4.1. Attach and detach

Deep-copy semantics require pointer translation within transferred objects to establish correct pointer relationships – this action is referred to as *attachment*, since it attaches two objects through a pointer relationship. The inverse operation – reverting a pointer relationship back to the state prior to an attach operation – is referred to as *detachment*.

Attachment is necessary when transferring two or more objects in a data structure, since the data structure is probably not usable if the pointer relationship between the objects is not properly established. Attachment can occur between objects that are being allocated and transferred as part of the current directive, or it can occur between objects that are already present prior to the current directive.

Detachment is necessary when removing an object from the device, since the pointer relationship is no longer valid. In general, every pointer attachment must be followed by a subsequent pointer detachment. This requires an implementation to track all pointer attachments with a reference count, similar to the present table. Without proper detachment it is possible to inadvertently transfer an object containing a device pointer back to the host, where the device pointer may be invalid.

An implementation must also refer to its attachment table when performing updates. An object containing a pointer member that has been attached can not be updated with a simple transfer, since the pointer has been translated and the value differs in the host and device copies. Instead, an implementation must *maintain* all translated pointers when performing updates. An implementation may either split updates around translated pointers, excluding the range of memory containing the pointers, or it may include the translated pointers in the transfer and reverse-translate them after the transfer is complete[1]. This flexibility allows an implementation to choose the approach that offers the best performance for a particular update.

4.2. Automatic alias resolution

Prior versions of OpenACC do not define an execution order for multiple data clauses on a single directive. For most programs the order does not matter, since clauses that apply to different regions of memory are independent. Clause order only matters in the presence of aliasing, when two or more clauses refer to overlapping regions of memory – in this

[1]Reverse translation is only legal when the pointer has not changed in the host or device copy, but the initial specification will prohibit changing translated pointers in data regions

case changing the clause processing order can change the behavior in an observable way. Consider the following directive, which copies array `a` and indicates array `b` is present:

```
#pragma acc data copy(a[0:10]) present(b[0:10])
```

If the arrays alias one another then the copy must be performed first, otherwise the present lookup will fail. To workaround this issue a programmer must split the data region into multiple directives, effectively enforcing the necessary order.

Unfortunately, such a workaround is not possible when aliasing occurs within a data structure that is being transferred with deep-copy semantics since the directive cannot easily be split into parts. The OpenACC committee considered defining ordering semantics for data clauses, but a full specification would require defining both clause order and graph traversal order for data structures. Further, a fixed order is not sufficient for all possible data structures, which can have complex bi-directional aliasing relationships between sub-objects. Rather than providing programmers with mechanisms to specify clause and traversal order, the OpenACC committee decided to place the burden on implementations. We refer to this as *automatic alias resolution*.

Automatic alias resolution requires an implementation to detect and handle all aliasing that occurs within a single directive. In the absence of aliasing an implementation is free to process clauses in any order. In the presence of aliasing an implementation must process clauses in an order that properly resolves all aliases, as defined by the following rules:

1. Pre-process all data clauses, including ones generated by deep-copy semantics, building a complete set of clauses for the current directive
2. Process clauses for present objects:
   a) Issue a fatal runtime error if any objects are partially present prior to the current directive
3. Resolve alias conflicts for non-present objects:
   a) Issue a fatal runtime error if two object partially overlap
   b) Allocate objects; if one object is fully contained in another then only allocate the larger one
   c) Perform data movement; an implementation may optimize data movement if two clauses would move the same or adjacent data in the same direction (partially overlapping data is not allowed)
4. Increment reference counts for all clauses

4.3. Limitations and complications

The initial specification of the new directives and clauses described in this report will impose several key restrictions on programs using them. This section describes the meaning and rationale of each of these restrictions.

Mutable deep-copy Mutable deep copy presents challenges beyond that of selective member or direction (see Section 1.2). Bottom up mutable deep copy is easily handled. A simple present check will determine if the memory referenced by an indirect member is already present. Top down mutable deep copy is not as easily handled. Specifically it requires the ability to dynamically detect whether a member is already present and attached or not (included or excluded). With this capability, top down deep copy essentially just requires the ability to have a member that is excluded in an enclosing data region become included in a contained data region. Therefore the ability to detect exclusion/inclusion is the critical capability. There are several possibilities of varying complexity and costs for achieving this.

1. Use the present table to keep track of members (direct or indirect) that have been included and excluded. This could potentially create a very large number of present table entries for cases like arrays of objects where some members are excluded. This places the burden entirely on the runtime but could have a high runtime cost because of dramatic increases in the size of the present table.
2. Limit mutable deep copy to indirect members. With this limit, the present table can be used to check for the existence of the target memory. If that memory is not present, then the member was excluded. When a member is included, create an entry in an attachment table to keep track of this. Since this table can essentially be made private to each instance of an object, there will be no single large table that would need to checked. Note that this attachment idea could be extended to direct members, but doing so would require keeping an entry for every direct member that is included (which in the common case would be all members).
3. Restrict mutable deep copy to only certain marked members. In this case, an additional clause could be added to the shape directive (for example dynamic or mutable). This clause would mark a member as being able to dynamically change from included to excluded within a data region. The runtime could then add an attachment table like entry for each of these members to keep track of whether they are currently included or excluded.

Modifying pointers in data regions A major assumption throughout this report is that the *structure* of a deep data structure remains fixed throughout the entire data region that places it on the device. That is, the pointer relationships between objects in a data structure may not change. This limitation is similar to requiring the shape of a flat object to remain fixed throughout the data region that places it on the device. For example, if a variable is transferred to the device with `copy(x[0:n])`, then the variable `n` may not be altered prior to the end of the data region. Relaxing this constraint complicates the transfer at the end of a data region – should the transfer honor the original or new value of `n`? Likewise, altering a pointer relationship within a data region presents similar ambiguities – should the final deep-transfer and deep-delete honor the original or new pointer relationship? Honoring the new pointer could result in complex memory management issues, such as deleting the same pointer more than once or failing to delete a pointer at all (i.e., orphaning an object). Given these challenges, the initial support for deep-copy in OpenACC will prohibit modifying translated pointers within data regions. Future revisions might be able to relax this limitation in various ways. For example, allowing programs to modify simple

pointer aliases is relatively straightforward, since it just requires translating the pointer correctly at the end of the data region. Allowing programs to modify pointers in general may require introducing a clause or directive that notifies an implementation when such a change has occurred.

Memory management on the device An object may not be transferred or deleted at the end of a data region unless it was created at the start of some prior data region. Similarly, an object may not be transferred in an update directive unless it was created in an enclosing data region. The intent of this restriction is to prohibit managing memory on the device.

OpenACC defines a master-slave model where a thread on the host initiates all actions on the device. Conceptually, all device memory management is initiated by the host, and all device objects are created to mirror corresponding host objects.[2] It is unclear what behavior should result if a compute region attempts to allocate or free memory, particularly if that memory is previously or subsequently involved in a data transfer between host and device. Presumably a compute region could manage memory that is only active for the lifetime of that compute region – in this case, the memory would be allocated after the start of the compute region and deallocated before the end of the compute region, and there would be no need to allocate corresponding host memory. But, what if a compute region attempts to deallocate memory that mirrors a host object? Should an implementation detect this situation, perhaps also deallocating the host object at the end of the data region? Similarly, if pointer reassociation is allowed on the device, then a compute region could allocate an object and store the pointer in a deep data structure that will be copied back to the host at the end of the data region. Should an implementation detect this and allocate a corresponding host object at the end of the data region? Both of these cases deviate from the expected paradigm, where all device memory management is initiated by the host, and all device objects mirror host objects.

This question of device memory management arises for standard C++ containers, such as `std::vector`. Transferring a `vector` requires deep copy, but supporting a resize operation on the device requires more – it requires allowing pointers to change and allowing memory management on the device. Since both of these capabilities introduce significant complications, the initial version of deep-copy prohibits resizing standard containers. Application developers with whom we've been working seem willing to accept this limitation, as long as containers may be resized between data regions.

A possible workaround for `std::vector` is to reserve additional space prior to a data region. Then the deep-copy policy would need to transfer the reserved space, not just the occupied space, allowing the compute region to append into the pre-reserved space without triggering an allocation. However, this approach requires support for changing pointers on the device, since appending to a vector will increment its end-of-data pointer. That is, when transferring a vector back to the host, after appending to it on the device, the end-of-data pointer must be reverse translated.

[2] Technically the `device_resident` clause allows an implementation to elide a host copy of an object; but, the device copy is still a logical mirror of the host copy, even if the host copy hasn't been created.

Conditional clauses When a shape or policy is applied to an object, all of the clauses it defines are unconditionally applied. In some cases it may be useful to support conditional clauses, that only apply if some condition is met. This could be useful for objects with dynamically changing behavior, for pointers in unions, and for recursion. Without first-class support of conditional clauses, an awkward workaround is to shape a pointer member so that the size computes to zero when the clause should not apply.

C++ semantics C++ exposes object creation, deletion, and assignment through explicit constructors, destructors and operators. However, OpenACC does not specify whether transferring a C++ object will invoke these standard routines. The essential question is whether a device copy of C++ object is truly a new object, or whether it is a temporary clone of the original host object. If we view it as a new object, then creating and transferring such an object should invoke the appropriate constructors, destructors and operators. The advantage of this approach is that it exposes the transfer mechanism to programmers, potentially giving them a way to control an object's transfer behavior. The disadvantage is that the behavior will differ for systems with unified memory, where an actual transfer is not required.

On the other hand, if we view a device object as a temporary relocation of the original host object, then creating and transferring an object should not imply language-level creation and copying of objects. In this case, the two objects are logically the same object, they just appear in different physical and temporal localities. The language of the current OpenACC specification supports this view, where object transfer is not subject to ordinary language semantics. Initial support for complex data management will not change this view.

However, there may be a middle ground where future specifications could expose transfer operations to programmers that want it. For example, a special member function could be called when an object is transferred. This solution still views host and device objects as a single logical object, but it exposes relocation operations as explicit events.

Pointers in unions This report does not provide a separate mechanism for scoping a pointer that appears as a member of a C/C++ union. The pointer can be shaped the same as if it were a member of a struct, but a pointer in a union may or may not be valid, depending on the dynamic manner in which the union is used. A programmer may always shape the pointer with a conditional size expression, where the size evaluates to 0 when the pointer is not valid, but the syntax for this workaround may be awkward. If pointers in unions turn out to be more important than our original polling suggests, then a future specification could add support for conditional clauses. Then a pointer in a union could be conditionally shaped, allowing the shape to only apply when the pointer is in use.

Unstructured data regions Unstructured data directives present complications when `enter` and `exit` deep-copy policies do not match. This problem also exists in the absence of deep-copy, where an `enter` and `exit` directive use different shapes. The main complication for deep-copy is that pointer traversal and translation could differ for the two directives.

The attachment table can at least catch errors. For example, if a shallow-`copyout` follows

a deep-`copyin` then an implementation could detect that an object being deleted contains attached pointers. It could detach the sub-objects, and optionally delete those sub-objects if the reference count drops to 0. Similarly, if a deep-`copyout` follows a shallow-`copyin` then an implementation could issue an error that the object is not attached.

Untranslated pointers How should pointers that aren't translated be handled? Should they remain as host pointers? Should they be set to null? Several users have indicated that setting them to null could be a useful debugging aid, or even a mechanism to check for presence of a sub-object. The disadvantage of setting them to null is that it causes the translated pointer table to grow larger. Also, behavior differs for systems with unified memory. Thus, the easiest solution is to make it undefined behavior to access an untranslated pointer member. An implementation could then provide various debug options to help users detect problems.

C++ templates To be useful in C++ the directives and clauses presented in this report must be applicable to template types, possibly even declaring policies outside of them. The main complication with doing this is that a template parameter itself could be a pointer or an object that requires deep copy. For example, `std::vector<int*> vec` declares a vector of pointers to integer. The policy syntax in this report does not provide a clean mechanism to shape the elements in a template container since the template type is not known until it is instantiated. If the template parameter is a type with an implicit policy then it will be applied when transferring a container of objects of that type. But programmers have no mechanism to define policies for raw pointers. One option is to allow defining different policies for different template specializations. Another option is to support syntax for *template policies*, which accept a policy as a template parameter – this would allow users to specify an explicit policy to use for transferring elements in a container. In any case, this functionality would come in a later OpenACC specification.

Polymorphism Polymorphic types present a unique challenge to deep-copy. Namely, the exact size of a polymorphic type may not be known statically. Thus, polymorphism is not supported by the directives described in this report.

The most natural way to support polymorphic types is to support *virtual policies*, similar to virtual function calls. A base class defines a virtual policy, and all sub-classes override that policy. At runtime, a virtual policy is resolved to that of the proper concrete type.

However, polymorphism presents additional challenges in OpenACC. For example, virtual function tables must be translated to target the device copy of each member function. Given these additional complexities, full support for polymorphism will likely take some time. Fortunately, high-performance codes often avoid polymorphism, so this functionality is not urgently needed.

Recursive data structures The initial version of deep-copy support does not permit recursive data structures. The nested syntax does not naturally support recursion, since it requires explicitly specifying a nest for every sub-object – adding support for recursion

requires additional syntax. The policy syntax does naturally support recursive types, but even so cyclic data structures pose a termination problem that require additional syntax to control conditional deep traversal.

Asynchronous deep-copy Achieving high performance with OpenACC often requires asynchronous transfers and computation. In theory, a deep-copy transfer is perfectly capable of running asynchronously. However, performing pointer translations asynchronously adds an additional level of complexity. Fortunately, an initial implementation can legally synchronize to ensure correctness, later adding true asynchronous deep-copy in future versions.

Scaling In the absence of deep-copy, every variable specified in a data clause triggers a single memory allocation and transfer. Even large, dynamically-sized arrays corresponded to a single block of contiguous memory. This implies that the number of independent objects that an implementation must manage is relatively small, bounded by the number of variables appearing in data clauses in a program. In contrast, deep-copy allows a single variable to trigger allocation and transfer of an arbitrary number of disjoint objects. This is a fundamental paradigm shift, moving from few large objects to many small objects, and it has direct scaling implications for OpenACC implementations. Since an implementation must track all host and device object mappings, a implementation with poor scaling will quickly become apparent for large deep-copy use cases. To reduce such scaling problems, programmers could employ aggregate allocation strategies, such as allocating many small objects with a single large allocation. This allows transferring a large group of objects with a single contiguous transfer. Implementations could automate similar strategies when applicable.

5. Future direction

Defining a directive-based mechanism for supporting deep-copy is a challenging problem, requiring trade-offs between expressiveness, usability, and implementation complexity. To strike a balance the OpenACC committee accepted several limitations on expressiveness, described in Section 4.3, which both simplify the syntax and reduce implementation effort. But since these trade-offs limit functionality for some use-cases, the OpenACC committee would like to relax some of these restrictions in future versions of the specification. The highest priority items are (in no particular order):

- parameters to policies (including default values)
  - Value parameters: allows providing one or more values as parameters when invoking a policy. Those parameters can then be referenced in a shape expression. This functionality allows a nested struct to shape a member based on a member in the enclosing class. It also allows member to be shaped based on the value of a local variable.
  - Policy parameters: allows specifying a named policy as a parameter when invoking a policy. This functionality allows a template type to accept a policy that will be applied to members of the template parameter type. For example, an `stl::vector` policy could accept a policy parameter that specifies the policy to use for elements in the container.
- policy parameter passing through language array shaping, such as

```
std::vector<float> a;
#pragma acc copy(a[0:n])
```

- parameters in nested syntax, allowing shape expressions in one nest to refer to member variables in an enclosing nest (analogous to value parameters for policies)
- nested syntax within policy definitions
- array parameters for policies, which allows for supporting true ragged arrays
- selective direction on direct members
- other forms of inherit (e.g., inherit-but-no-copyout)
- direction-based default policies, such as a `copyin` or `copyout` policy
- recursive data types and recursion in policies, which is required for supporting common implementations of `std::map` and `std::list`

6. Conclusion

OpenACC has received much attention in the high-performance computing community recently, and it is viewed by many people as a notable step forward in easing the difficulty of programming heterogeneous accelerator systems. The promises are lofty: a directive-based approach allows a single source-code base to target both homogeneous and heterogeneous systems; a descriptive rather than prescriptive set of directives facilitates performance portability by giving an implementation the flexibility to optimize and efficiently map a program onto different accelerator targets; and, the abstraction of disjoint host and device memory, coupled with compiler and runtime assistance in managing device memory, frees a programmer from the tedious details of manually managing device memory, a task that was historically synonymous with accelerator programming. However, as excitement around the initial success of OpenACC begins to fade, it becomes clear that lack of deep-copy support is the single largest impediment to porting a wider range of interesting data structures and applications to OpenACC. With no better option, users are forced into manually managing device memory and rewriting large portions of their application code – an outcome that runs counter to the entire purpose of a directive-based programming model.

The potential benefit of providing a directive-based solution for deep copy is great, as it would extend the usefulness of OpenACC to a much larger set of applications. But in general it is a very challenging problem, as illustrated in this report. Even so, we believe the problem is solvable, especially with some reasonable assumptions and limitations that we've outlined in the report. The shape directive provides a simple, intuitive mechanism for expressing full and selective member deep-copy, which is sufficient to support many real-world applications. Or, the slightly more complex policy directive can be employed to achieve selective direction deep-copy. These global directives facilitate encapsulation and reuse. Alternatively, the nested syntax provides a relatively concise way to express local, anonymous shapes and policies. Finally, for many simple cases the short-hand syntax allows users to bypass shapes and policies entirely, expressing deep-copy behavior in an intuitive, natural manner.

With this new syntax the deep-copy requirements for many applications can be expressed solely through compiler directives, without any additional code modification, allowing a compiler and runtime to automatically handle the tedious details of allocating and transferring complex data structures between host and device memory. More generally, these directives provide an elegant mechanism for programmers to describe shapes and relationships of objects in their data structures. This capability will likely remain important even as the emergence of unified memory diminishes the need for traditional deep-copy functionality, since achieving high performance will likely require careful attention to data locality. A directive-based approach can assist programmers in placing and relocating complex data structures within the memory hierarchy.

Bibliography

[1] Heterogeneous System Architecture (HSA) Foundation. http://www.hsafoundation.com.

[2] ICON GCM: ICOsahedral Non-hydrostatic General Circulation Model. http://icon.enes.org.

[3] International Standard ISO/IEC 14882:1998. *Programming Language C++*.

[4] Message Passing Interface Forum, Knoxville, TN, USA. *MPI: A Message-Passing Interface Standard*, September 2012.

[5] Nvidia. *CUDA C Programming Guide*.

[6] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, July 2013.

A. Examples

```
struct deep_type {
  int n;
  float* a;
  float* b;
  float* c;
  // This default shape includes deep copy of members a, b, and c, and
  // it ensures member n is always initialized; C pointers must be
  // shaped to get deep copy, since the default shape is a bitcopy of
  // the pointer value
  #pragma acc shape init_needed(n) include(a[0:n],b[0:n],c[0:n])

  // This named shape specializes the default shape by excluding
  // members b and c; thus, it only includes members n and a
  #pragma acc shape(part_a) exclude(b,c)

  // This named shape excludes everything except member b, which is
  // useful for updates
  #pragma acc shape(only_b) default(exclude) include(b)
};

deep_type X;

// Performs deep copy of X
#pragma acc data copy(X)

// Performs deep create of X; creates but doesn't initialize members
// a, b, and c; creates and initializes member n
#pragma acc data create(X)

// Performs selective member deep copy; member n is initialized;
// member a is created but not initialized; members a and n are copied
// out at the end of the data region
#pragma acc data copyout<part_a>(X)

// Performs a deep update of X
#pragma acc update self(X)

// Only updates member b
#pragma acc update self<only_b>(X)
// Equivalent to above pragma
#pragma acc update self(X.b[0:X.n])

deep_type* Y;
int size;

// Performs a deep copy of Y; note that member n can be different for
// each element of Y
#pragma acc data copy(Y[0:size])

// Performs an update of member b for all elements in Y
#pragma acc update self<only_b>(Y[0:size])
// Equivalent to above pragma
for (int i=0;i<size;++i) {
  #pragma acc update self(Y[i].b[0:Y[i].n)
}
```

Figure A.1.: C++ example showing how to use the `shape` directive (see Section 3.1)

```
template<Type T>
class vector {
  T* base;
  T* end;
  T* reserved;
  #pragma acc shape include(base[0:size()], end[@base], reserved[@base])
};

vector<float> v1;

// The following directives imply the expected full deep-copy behavior
#pragma acc data copy(v1)
#pragma acc data create(v1)
#pragma acc data update self(v1)


class Data {
  vector<float> d1;
  vector<float> d2;
  // This named shape excludes member d2; member d1 is included from
  // its implicit default shape
  #pragma acc shape(only_d1) exclude(d2)
};

Data d;
// This directive performs full deep-copy, since the implicit default
// shape is default(include) and each member has a default shape
#pragma data copy(d)

class Stuff {
  Data data1;
  Data data2;
  vector<float> raw;
  int direct[100];
  // This shape indicates that member raw is not needed on the device
  #pragma acc shape default(include) exclude(raw)
  // Shapes can also exclude members of user defined type
  #pragma acc shape(data1) exclude(data2)
  #pragma acc shape(data2) exclude(data1)
  // An include clause can specify a named shape for some members; a
  // copy using the named shape "small" would only copy members
  // direct[100], data1.d1, and data2.d1; members raw, data.d2, and
  // data2.d2 are not copied
  #pragma acc shape(small) include<only_d1>(data1,data2)
};
```

Figure A.2.: C++ example showing how to use the `shape` directive (see Section 3.1) with a template type

```
struct deep_type {
  int n;
  float* a;
  float* b;
  float* c;
  #pragma acc shape init_needed(n) include(a[0:n],b[0:n],c[0:n])

  // Policy to copyin members b and c and copyout member a (which
  // might be used for a computation like a = b + c)
  #pragma acc policy(calc_a) default(copyin) copyout(a)

  // Policy to copyin member a, copyout member c, and exclude member b
  // (which might be used for a computation like c = a)
  #pragma acc policy(move_a_to_c) default(copyin) copyout(c) exclude(b)

  // Policy to update member b only
  #pragma acc policy(update_b) default(exclude) update(b)
};

deep_type X;

// This directive performs a full deep-copy based on the default shape
#pragma acc data copy(X)

// These directives use policies to perform selective direction deep-copy:
// copyin members B and C, copyout member A
#pragma acc data invoke<calc_a>(X)
// copyin member A, copyout member C, exclude member B
#pragma acc data invoke<move_a_to_c>(X)

// This directive updates member b on the device
#pragma acc update invoke<update_b>(device: X)

deep_type* array_X;
int size;
// A policy can be used to update member b for each element in an
// array of objects
#pragma acc update invoke<update_b>(device: array_X[0:size])

class Compound_type {
  vector<float> v1;
  vector<float> v2;
  vector<float> v3;
  vector<float> raw;
  Deep_type data;

  #pragma acc shape exclude(raw)
  #pragma acc shape(only_vs) exclude(data)

  // create(v1,v2,v3) copyin(data) with default shape raw is excluded from default
      shape
  #pragma acc policy(name1) default(create) copyin(data)

  // Using a policy for a member
  #pragma acc policy(name2) default(copyin) invoke<calc_a>(data)

  // Use a named shape raw and data are excluded
  #pragma acc policy(name3) shape(only_vs) default(none) copyin(v2,v3) copy(v1)

  // Policy to update some arbitrary members: v1,v2 and data (using default shape)
  #pragma acc policy(update_stuff) default(exclude) update(v1,v2,data)

  // Policy to update some members, including only specific parts of members (using a
       policy)
  #pragma acc policy(update_other) default(exclude) update(v2) invoke<update_b>(data)
};
```

Figure A.3.: C++ example showing how to use the `policy` directive (see Section 3.2)

```
type DeepType
  real,allocatable :: A(:)
  real,allocatable :: B(:)
  real,allocatable :: C(:)
  ! use exclude to not do deep copy for a member (selective member)
  !$acc shape(not_c) exclude(C)
end type DeepType

type(DeepType) :: X
! implicit default shape will do a full deep copy
!$acc data copy(X)

! use named shape to only copyin A and B
!$acc data copyin<not_c>(X)

type CompoundType
  type(DeepType), alloctable :: d1(:)
  real, allocatable :: raw
  ! Can exclude members by default and specify default shape for member
  !$acc shape exclude(raw) include<not_c>(d1)
end type CompoundType
```

Figure A.4.: Fortran example showing how to use the `shape` directive (see Section 3.1)

```
type DeepType
  real,allocatable :: A(:)
  real,allocatable :: B(:)
  real,allocatable :: C(:)
  ! Policy to copyin a,b and copy c maybe for something like a = b + c
  !$acc policy(calc_a) default(copyin) copyout(A)
end type DeepType

! invoke a policy
type(DeepType) :: X
!$acc data invoke<calc_a>(X)

! policies can be used on arrays of type
type(DeepType), allocatable :: array_X(:,:)
!$acc data invoke<calc_a>(array_X)
```

Figure A.5.: Fortran example showing how to use the `policy` directive (see Section 3.2)

```
struct deep_type {
  int n;
  float* a;
  float* b;
  float* c;
};

deep_type X;

// Performs deep copy of X
#pragma acc data copy(X)::{ init_needed(n) include(a[0:n],b[0:n],c[0:n]) }

// Performs deep create of X; creates but doesn't initialize members
// a, b, and c; creates and initializes member n
#pragma acc data create(X)::{ init_needed(n) include(a[0:n],b[0:n],c[0:n]) }

// Performs selective member deep copy; member n is initialized;
// member a is created but not initialized; members a and n are copied
// out at the end of the data region
#pragma acc data copyout(X)::{ init_needed(n) include(a[0:n]) exclude(b,c) }

// Performs a deep update of X
#pragma acc update self(X)::{ include(n, a[0:n],b[0:n],c[0:n]) }

// Only updates member b
#pragma acc update self(X)::{ include(b[0:n]) exclude(n,a,c) }
// Equivalent to above pragma
#pragma acc update self(X.b[0:X.n])

deep_type* Y;
int size;

// Performs a deep copy of Y; note that member n can be different for
// each element of Y
#pragma acc data copy(Y[0:size])::{ init_needed(n) include(a[0:n],b[0:n],c[0:n]) }

// Performs an update of member b for all elements in Y
#pragma acc update self(Y[0:size])::{ include(b[0:n]) exclude(n,a,c) }
// Equivalent to above pragma
for (int i=0;i<size;++i) {
  #pragma acc update self(Y[i].b[0:Y[i].n)
}
```

Figure A.6.: C++ example showing how to define an anonymous shape using the nested shape syntax (see Section 3.3); this example duplicates the behavior in Figure A.1 using local instead of global syntax

```
struct deep_type {
  int n;
  float* a;
  float* b;
  float* c;
  #pragma acc shape init_needed(n) include(a[0:n],b[0:n],c[0:n])
};

deep_type X;

// This directive performs a full deep-copy based on the default shape
#pragma acc data copy(X)

// This directives uses a local policy to extend the default shape
// with selective direction: copyin members b and c and copyout member
// a (which might be used for a computation like a = b + c)
#pragma acc data invoke<>(X)::{ default(copyin) copyout(a) }

// This directives uses a local policy to extend the default shape
// with selective direction: copyin member a, copyout member c, and
// exclude member b (which might be used for a computation like c = a)
#pragma acc data invoke<>(X)::{ default(copyin) copyout(c) exclude(b) }

// This directive uses a local policy to update member b only
#pragma acc update invoke<>(device: X)::{ default(exclude) update(b) }

deep_type* array_X;
int size;
// A local policy can be used to update member b for each element in
// an array of objects
#pragma acc update invoke<>(device: array_X[0:size])::{ default(exclude) update(b) }
```

Figure A.7.: C++ example showing how to define an anonymous policy using the nested policy syntax (see Section 3.3); this example duplicates the behavior in Figure A.3 using local instead of global syntax

```
class deep_type {
  int n;
  float* a;
  float* b;
  float* c;
  void foo();
};

deep_type X;

// Performs a deep-copy of member a only, but not members b and c
#pragma acc data copy(X.a[0:X.n])
// Equivalent to this anonymous policy
#pragma acc data invoke(X)::{ default(exclude) copy(a[0:n]) }

// Performs a copyin of members a and b and a copyout of member c
#pragma acc data copyin(X.a[0:X.n], X.b[0:X.n]) copyout(X.c[0:X.n])
// Equivalent to this anonymous policy
#pragma acc data invoke(X)::{ default(exclude) copyin(a[0:n], b[0:n]) copyout(c[0:n])
     }

// Performs full deep-copy of X
#pragma acc data copy(X, X.a[0:X.n], X.b[0:X.n], X.c[0:X.n])
// Equivalent to this anonymous policy
#pragma acc data invoke(X)::{ default(copy) copy(a[0:n], b[0:n], c[0:n]) }


void deep_type::foo() {
  // Performs a copyin of members a and b
  #pragma acc data copyin(a[0:n], b[0:n])
  // Equivalent to this anonymous policy
  #pragma acc data invoke(this[0:1])::{ default(exclude) copyin(a[0:n], b[0:n]) }
}
```

Figure A.8.: C++ example showing how to use convenience syntax (see Section 3.4), also showing the equivalent local policy syntax (see Section 3.3)