SANDIA REPORT

SAND2016-5397
Unlimited Release
Printed March 2016

DARMA 0.3.0-alpha Specification

Jeremiah J. Wilke, David S. Hollman, Nicole L. Slattengren, Jonathan Lifflander,
Hemanth Kolla, Francesco Rizzi, Keita Teranishi, Janine C. Bennett

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

SAND2016-5397
Unlimited Release
Printed March 2016

DARMA 0.3.0-alpha Specification

Jeremiah J. Wilke, David S. Hollman, Nicole L. Slattengren, Jonathan Lifflander,
Hemanth Kolla, Francesco Rizzi, Keita Teranishi, Janine C. Bennett

Abstract

In this document, we provide the specification for DARMA (Distributed Asynchronous Resilient Models and Applications), a co-design research vehicle for asynchronous many-task (AMT) programming models that serves to: 1) insulate applications from runtime system and hardware idiosyncrasies, 2) improve AMT runtime programmability by co-designing an application programmer interface (API) directly with application developers, 3) synthesize application co-design activities into meaningful requirements for runtime systems, and 4) facilitate AMT design space characterization and definition, accelerating the development of AMT best practices.

DARMA is a translation layer between an application-facing front end API and a runtime system facing back end API. The application-facing user-level front end API is an embedded domain specific language (EDSL) in C++, inheriting the generic language constructs of C++ and adding semantics that facilitate expressing distributed asynchronous parallel programs. Though the implementation of the EDSL uses C++ constructs unfamiliar to many programmers to provide the front end semantics, it is nonetheless fully embedded in the C++ language and leverages a widely supported subset of C++14 functionality (gcc $>= 4.9$, clang $>= 3.5$, icc $>= 16$). The translation layer leverages C++ template metaprogramming to map the user's code onto the back end runtime API. The back end API is a set of abstract classes and function signatures that runtime system developers must implement in accordance with the specification requirements in order to interface with application code written to the DARMA front end. Executable DARMA applications must link to a runtime system that implements the abstract back end runtime API. It is intended that these implementations will be external, drawing upon existing AMT technologies. However, a reference implementation will be provided in the DARMA code distribution.

The front end API, translation layer, and back end API are detailed herein. We also include a list of application requirements driving the specification (along with a list of the applications contributing to the requirements to date), a brief history of changes between previous versions of the specification, and summary of the planned changes in upcoming versions of the specification. Appendices walk the user through a more detailed set of examples of applications written in the DARMA front end API and provide additional technical details for those the interested reader.

Acknowledgment

This work was supported by the U. S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) program and the DOE Office of Advanced Scientific Computing Research. Sandia National Laboratories (SNL) is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the DOE NNSA under contract DE-AC04-94AL85000.

Contents

Chapter 1

Introduction

As we look ahead to next generation platforms and exascale computing, hardware will be characterized by dynamic behavior, increased heterogeneity, decreased reliability, deep memory hierarchies, and a marked increase in system concurrency (both on-node and system-wide) [1, 2]. These architectural shifts are posing significant programming challenges for application developers as they seek solutions for: effective management of hybrid parallelism at an unprecedented scale, efficient load-balancing and work-stealing strategies that mitigate both application and system load imbalance, and effective management and staging of data across deep memory hierarchies. To further complicate matters, application codes must be made performance portable across a variety of planned system architectures and be made resilient to the increased number of anticipated faults.

AMT programming models and runtime systems show promise to mitigate the challenges associated with the changes in high-performance computing (HPC) system architectures. AMT models are a shift away from the current communicating sequential processes (CSP) programming model as they strive to exploit all available task parallelism and pipeline parallelism, rather than rely solely on data parallelism for concurrency. The term *asynchronous* encompasses the idea that 1) processes (threads) can diverge to different tasks, rather than execute the same tasks in the same order; and 2) concurrency is maximized (the minimal amount of synchronization is performed) by leveraging multiple forms of parallelism. The term *many-task* encompasses the idea that the application is decomposed into many migratable units of work, to enable the overlap of communication and computation as well as asynchronous load balancing strategies. A key design goal of AMT models is to enable performance-based optimizations of code dynamically at runtime. We note that performance-based code transformations are ubiquitous at the compiler-level. Compilers will add, delete, swap, or reorder instructions to avoid unnecessary operations, improve data locality, or improve pipelining. Furthermore, there are a number of compile-time optimization tools being developed [3, 4], that provide the ability to map a single code kernel onto high-performance execution across diverse compute platforms. However, many optimizations that benefit performance are unknowable until the program actually runs – as these decisions may be based on current system performance, or the data needs of the application itself. Such dynamic runtime optimizations are much more expensive than compile-time optimizations, thus the use of tasks as a basis for dynamic runtime transformations.

The AMT community is currently very active (e.g., [5–15]), representing a range of different design points within the design space of AMT models. While the technologies show significant potential to address challenges, the community has not yet identified best practices and existing systems still represent a variety of different programming model, execution model, memory model, and data model design choices.

Programming model: From a programming model perspective, AMT models all have some notion of decomposing applications into small, migratable units of work. Task parallelism can be expressed in a fork-join fashion, with users managing control-flow explicitly themselves. In other AMT programming models, the user expresses a algorithm step-by-step and, under some simplifying assumptions, the runtime derives the synchronizations required. This often takes the form of read/write data access annotations under the assumption of sequential semantics enabling runtime dependency analysis. Models leveraging runtime analysis are best suited for coarse-grained task parallelism, as runtime system overheads must be amortized.

Execution model: Execution models broadly cover how the algorithm and corresponding correctness and performance constraints specified in the programming model are translated to actual execution. For example, AMT runtimes implement a variety of execution models, including event-based, fork-join (either fully strict or terminally strict), actor model, or ubiquitous CSP model. More subtle details include whether a constant number of threads are always executing (e.g. pure MPI codes), new tasks are allocated to a thread pool, or if new threads are allocated (forked) for new tasks. These details will also affect the synchronizations required in an execution model. For example, fully strict fork-join models will generally not require barriers between sibling tasks. In contrast, CSP models will require barriers to synchronize parallel workers.

Memory model: HPC memory models will have several properties including distributed or shared and coherent or incoherent. Distributed memory models include message-passing models like MPI. Partitioned Global Address Space (PGAS) models have distinct address regions, but are "shared" in the sense that any memory location can be accessed across the system by specifying both a pointer address AND process ID. In DARMA, computational tasks by default can only operate on their local data. When remote data is required, it is communicated between the remote and local tasks. Across the spectrum of memory models, memory locations are usually accessed via address (put/get or send/recv), but key-value store (tuple space) models identify data regions by key identifiers (coordination). In coordination, parallel workers never directly communicate, instead 'coordinating" indirectly via a key-value store or tuple space.

Data model: In order for data-flow AMT models to make effective data management decisions (e.g., slicing the data and making copies to increase parallelism), they must have some knowledge of the structure of the data. One option for providing structural information regarding data is to impose a data model. Another option is to require application developers to define serialization, slicing, and interference tests for their data blocks.

1.1 Scope

Although the AMT model community is quite active, the lack of standards impedes adoption of these technologies by the application community. Although it is premature to standardize, there is sufficient breadth and depth in the AMT research community to begin developing community best practices. Towards this end, this document provides the specification for DARMA, a research vehicle for AMT programming model co-design. DARMA aims to serve four primary purposes:

Insulate applications from runtime system and hardware idiosyncrasies: As part of its design, DARMA separates its application-facing front end and runtime system-facing back end APIs. This separation of concerns enables an application team to explore the impact of runtime system design space decisions. For example, application developers can build their code using different DARMA-compliant back end implementations, without having to deal with the combinatorial complexity of implementing their application in many different front end APIs. It should be noted that DARMA's front end API is not fixed – it will evolve based on co-design feedback from both application and runtime system developers.

Improve AMT runtime programmability by co-designing a front end API directly with application developers: Recent work [16] highlighted gaps with respect to productivity in some existing AMT runtime systems, in particular noting requirements gaps and deficiencies in existing APIs. Co-designing DARMA's front end API directly with application developers provides a mechanism for capturing different application's runtime system requirements– giving them a voice in the design of an asynchronous tasking API. Experimenting with the API provides an agile method for application developers to reason about the API and better articulate their runtime system execution requirements.

Synthesize application co-design activities into meaningful requirements for runtimes: The specification provides a mechanism for tracking the provenance of design decisions and requirements as they evolve throughout the co-design process. Chapter 5 provides a list of the application requirements gathered, and Chapter 6 tracks the evolution of the specification, highlighting which requirements motivated changes to the specification. Runtime system software stack developers benefit from 1) DARMA's application-informed requirements, and 2) access to code kernels and proxy applications developed via the front end co-design process.

Facilitate AMT design space characterization, accelerating the development of AMT best practices: In the discussion above we summarize a range of high-level design decisions for AMT programming, execution, memory, and data models. DARMA's separation of front end and back end APIs seeks to facilitate this design space characterization and exploration. There is a notable tension between the design of 1) a front end API that is expressive, simple, and easy to incorporate within existing application code bases, and 2) a back end API that is simple enough to support multiple DARMA-compliant implementations that leverage existing runtime system technologies. Consequently, DARMA APIs (both front end and back end) are intended to evolve based on iterative feedback from application, programming model, and runtime system teams.

The rest of this chapter provides a high-level description of DARMA's structural design along with a brief summary of DARMA's programming, memory, data, and (compatible) execution models. We note that throughout the co-design process, decisions are first and foremost, made to best support application requirements. Furthermore, we target a back end API specification that is general enough to support AMT runtime system design space exploration, via build out of DARMA-compliant back ends using existing AMT runtime system technologies. Lastly, we note that the features detailed in Chapters 2 and 4 are not entirely comprehensive – meaning they do not yet capture all of the application requirements driving DARMA co-design. This is because we are formalizing the specification process from the inception of DARMA, layering-in features incrementally to provide the community opportunity for input, and active engagement in the co-design process. Suggested enhancements and changes to the DARMA specification are welcome and can be made via a DARMA Enhancement Plan (DEP).

1.2 High-level Design

DARMA is a translation layer between an application-facing front end API and a runtime system facing back end API. DARMA's front end API is an EDSL in C++, inheriting the generic language constructs of C++ and adding semantics that facilitate distributed, deferred, asynchronous, parallel programming. Though the EDSL uses C++ constructs unfamiliar to many programmers to implement these semantics, it is nonetheless fully embedded in the C++ language and requires a widely supported subset of C++14 functionality (gcc $>= 4.9$, clang $>= 3.5$, icc $>= 16$). The front end API is the center of programming model co-design activities, which seek to involve a wide variety of both application and runtime system developers.

DARMA's translation layer leverages C++ template metaprogramming to map the user's front end API calls onto the back end runtime API, bridging the programming model and actual program execution. We note however that the DARMA translation layer itself does not perform any performance optimizations – these are left entirely to the back end runtime system implementations. Rather, the translation layer converts the application code specified with DARMA's front end API into an "intermediate representation" that enables a runtime system to make intelligent, dynamic decisions (e.g., about task order and task locality or possibly even task deletion and task replication when appropriate).

The back end API is a set of abstract classes and function signatures that runtime system developers must implement in accordance with the specification requirements in order to interface with application code written to the DARMA front end. Strictly speaking, the back end API calls only generate a stream of deferred tasks (tasks with corresponding data inputs/outputs) that implicitly capture the program's data-flow. The information passed through the translation layer to the back end is sufficient to (and intended to) support a computational directed acyclic graph (CDAG) representation of the application. In a task-DAG representation, tasks are vertices V in a graph G with directed edges E. An edge from vertex v_1 to vertex v_2 indicates a precedence constraint. A CDAG representation describes task-data precedence constraints, rather than just task-task precedence constraints. In a CDAG there are two types of vertices - tasks T and data D that compose the complete set of vertices V. Edges never directly connect two tasks and instead edges are only ever described between a task vertex, t, and a data vertex, d indicating that (depending on direction of the edge) data is either consumed or produced by a task. The task-DAG indicating task-task precedence constraints can always be obtained from the CDAG, which captures the data-flow task graph. The CDAG is thus more general, capturing additional information to enable runtime code transformations.[1]

Finally, we highlight that a DARMA executable application must link to a runtime system that implements the abstract back end runtime API. It is intended that these implementations will be external, drawing upon existing AMT technologies. However, a reference implementation will be provided in the DARMA code distribution.

There ar a number of terms, such as rank, task, and process that are loaded with many definitions across the literature. Here we give special attention to define rigorous and limited definitions for such terms used throughout the document. We use process in the usual UNIX sense. Other terms are:

Task: The work unit instantiated directly by the application developer. Tasks are also the smallest granularity of *migratable* work unit. In the current specification, tasks cannot migrate after beginning execution. Tasks are guaranteed to make forward progress, but are interruptible.

[1] Although beyond the scope of this specification document, the interested reader will find numerous works discussing heuristics and order-preserving transformations of task graphs that demonstrate the utility of a coarse-grained CDAG for enabling dynamic runtime optimization of an algorithm [17–19]. We reiterate that the CDAG is only a concept guiding the design of the back end API and not strictly part of the DARMA specification.

Execution stream: An execution stream will consist of a sequence of many tasks, and, like tasks, is guaranteed to make forward progress. All execution streams are tasks, but execution streams specifically have no parent task and are the root of an independent task graph. Each execution stream is guaranteed to have a unique stack and, any point time, will have a local context of variables. A physical process (in the UNIX sense) can be running many parallel execution streams. Allowing multiple execution streams per physical process is the basis for overdecomposition. Since several execution streams can exist in the same process address space, this introduces a strict requirement of no global variables. An execution stream is the DARMA generalization of a thread, except that extra privatization of variables is necessary since no assumption of shared memory between independent execution streams can be made (even if execution streams happen to be executing in the same process). Just as processes must perform special operations to exchange data between them (message-passing, mmap), independent execution streams must perform special key-value store operations to exchange data between them. Execution streams are always assigned a unique identifier by the runtime system.

Operation: Used synonymously with work unit. This is a unit of execution that is guaranteed to be non-interruptible. An operation is not equivalent to a task since tasks are interruptible. Operations are the smallest, schedulable units of work. A task consists of a sequence of operations. While tasks are explicitly instantiated by the application developer, operations (individual portions of a task) can be implicitly instantiated by the runtime system. Tasks can yield at the beginning/end of its component operations, allowing the runtime system to schedule new work units for execution.

Rank: A unique integer ID for an execution stream. This matches the MPI notion of rank as an integer identifying a process within an MPI communicator. The term rank will often be used in the specification as a synonym for execution stream (more precisely, a metonymy for execution stream). Generally speaking, N parallel execution streams are created in an single-program multiple-data (SPMD) launch (more in Section 2.7.2). The runtime system then assigns unique identifiers (rank IDs) 0 through $N-1$ to each execution stream. Referring to "rank 0" will therefore function as shorthand for "the execution stream that has been assigned rank ID 0 by the runtime in an SPMD launch." Similarly, referring generically to a "rank" is shorthand for "an execution stream created by an SPMD launch with a particular rank ID."

Key-Value (KV) Store: A key-value store is an associative map from keys to values. In general, there are no restrictions on what keys or values are, although in many cases keys are strings. The only thing required is that keys be comparable. For an unordered map implementation of a key-value store, keys must usually be hashable.

Tuple Space : A generalization of a key-value store in which keys are tuples of individually comparable values. When this specification refers to a tuple space, we are only referring to a particular type the use of at tuple as a key within a key-value store. Unlike other tuple space languages (e.g. Linda) we do not require tuple spaces to implement wildcard (or any other operations), only the comparison on fully-specified tuples. Implementation of the key-value store as a tuple space is not required. Even though variables must be constructed with a unique tuple, a particular backend implementation may choose to convert tuple into string representations and implement a simple string-based key-value store.

distributed hash table (DHT) : A particular implementation of a key-value store for hashable key types. Intended to be scalable for large systems, the hash space is partitioned across distributed workers. This automatically (and predictably) scatters keys and corresponding values across the system. A DHT implementation of a key-value store is not required by the specification, but recommended for scalable execution.

1.3 Programming Model

Programming models provide application developers abstractions for expressing *correct* and *performant* algorithms. As described earlier, a key design goal of AMT models is to enable performance-based optimizations of code dynamically at runtime. Runtime-based optimizations come with an associated runtime cost, which is what motivates the use of tasks (rather than, e.g., instructions) as the basis for dynamic runtime transformations. Existing AMT models provide a variety of APIs for capturing and expressing data-flow dependencies and communicating these to the underlying runtime system. One of the adoption challenges many of these runtime systems face is that they require a significant shift away from what has become the defacto standard of distributed HPC programming: CSP.

DARMA's programming model seeks to facilitate the expression of deferred, asysnchronous work, enabling a back end runtime system to perform dynamic runtime optimizations, while making it as simple as possible for programmers to reason about the correctness of their code. This motivates DARMA's combined use of successful programming model concepts from a variety of existing runtime systems. One of DARMA's programming model key design decisions is rooted in the following observations: 1) all application developers can effectively reason about how to write correct sequential codes, 2) all MPI programmers can effectively reason about how to write correct CSP codes, and 3) most applications written in or ported to DARMA will likely have SPMD as their dominant parallelism. To simplify the implementation of SPMD-structured codes, the notion of a rank is maintained within the API. By maintaining the notion of a rank, DARMA provides application developers a convenience mechanism for creating the initial problem decomposition and distribution. Immediately after launch, any user-specified deferred work is free to be migrated by the runtime system, if it will result in better performance. Because DARMA maintains the notion of a rank, it is also possible for DARMA to maintain CSP-like semantics (in particular, within an initial implementation or port of a code prior to the introduction of deferred work). This grants developers the ability to express correctness constraints through a familiar and intuitive programming model. DARMA facilitates the expression of hybrid parallelism by supporting sequential semantics, within a rank. This means that application developers can reason about code as though it were being deployed sequentially within the rank, even in the presence of user-specified deferred work.

DARMA employs C++-embedded task annotations for the specification of deferred work. Each block of deferred work can be considered a task (coarse-grained blocks of procedural imperative code), which is not necessarily performed in program order. Instead, deferred work is performed asynchronously when all of its data-flow dependencies are satisfied. Task parallelism is primarily achieved through permissions/access qualifiers on data that enable a runtime to reason about which tasks can run in parallel and which tasks are strictly ordered. Task granularity is determined by the user and annotations are translated by DARMA's translation layer through standard C++ constructs (e.g., lambdas, reference counted pointers) and template metaprogramming to expose task parallelism inherent in the code. We note here that DARMA's runtime optimizations are complementary to compile-time optimizations performed by *performance portability* tools, e.g. [3, 4, 20]. Compile-time performance portability tools provide the ability to map a single code onto high-performance execution across diverse compute platforms.

One of the major differences between DARMA and a traditional CSP programming model is the manner in which communication is performed. Communication between DARMA ranks is not performed via direct messaging. Instead, coordination semantics are used. Processes *coordinate* by putting/getting data associated with a unique `key` in a key-value store (`key` are general tuples). Coordination semantics enables out-of-order message arrival, deferred execution, task migration, and resilience strategies since the application declares or describes the data it needs/produces rather than enforcing an explicit delivery mechanism. The coordination semantics in the specification are intended to support the use of zero-copy mechanisms and tuple caching, generally producing execution equivalent to an MPI `send/recv` code.

Together, these features make DARMA a mixed imperative/declarative programming model. As much as possible, sequential imperative semantics are used to produce intuitive, maintainable code. However, the "procedural imperative" function calls and code blocks do not necessarily execute immediately. Rather than explicitly perform all work in program order and block on data requests, they wait for all data-flow dependencies to be satisfied. Such deferred execution makes DARMA declarative, leaving the exact control-flow up to the runtime system. Furthermore, it is this ability to defer work and advance ahead that gives the back end runtime system the ability to make performance-improving transformations.

Although not yet supported in version 0.3.0-alpha of the specification, several important features will play a role in the DARMA programming model:

Expressive Underlying Abstract Machine Model: Notions of execution spaces and memory spaces will be introduced formally in later versions of the specification. These abstractions (or similar ones) appear in other runtime solutions, e.g. [3, 4]. Using such abstractions 1) facilitates performance portable application development across a variety of execution spaces, and 2) provides finer-grained control and additional flexibility in the communication of policies regarding data locality and data movement.

Runtime performance introspection In future versions of the specification DARMA will specify hooks for the application developer to express, guide, and leverage the use of runtime-level performance introspection. An important co-design activity will include determining whether performance introspection needs to factor into the application-level programming model on the front end or whether it belongs only in the back end runtime system API.

Expression of fine-grained deferred parallel patterns. In future versions of the specification, DARMA will specify deferred fine-grained parallel patterns, e.g., deferred `parallel-for`, `parallel-scan`, etc.

Instantiating tasks in class member functions Due to idiosyncracies in C++ lambda capture, inline `create_work` calls cannot operate on member variables within a class member function. Mechanisms for circumventing this C++ limitation will be introduced in later versions.

Subsetting/slicing handles Certain tasks may only require access to a subset of the data owned by a handle created with `initial_access`. Using the handle in such a task therefore overexpresses contraints, which is contrary to the philosophy of DARMA for avoiding unnecessary synchronizations and task preconditions. Expressing subsets of classes/slices of arrays will be an important part of future specifications.

Data Staging: The memory and execution space concepts introduced above enable 1) performance portable tasks that can run in multiple environments through a single code and 2) user-directed (or runtime-directed) asynchronous data movement to move data to compute devices.

Collectives: Some collectives will be supported by DARMA in version 0.3.1 of the specification, including `all-reduce`, `reduce-scatter`, and `barrier` collectives. Collectives will be data-centric rather rank-centric, as done in MPI.

Programmer-directed optimization While an abstract algorithm may make more information available to the compiler or runtime for performance-tuning transformations, compilers and runtime schedulers may not always understand the global nature of the problem. As such, they may not make performance-improving optimizations that are apparent to an application developer. A critically important part of future co-design activities will be the development of the interface by which developers can steer the runtime towards a desired set of optimizations that compilers or runtime schedulers might fail to perform.

1.4 Execution Models

The main focus of DARMA is the programming model and corresponding translation layer that maps a program expressed via a combination of CSP semantics, coordination semantics, and additional C++-embedded task annotations into a generic data-flow based description of an algorithm based on deferred tasks. DARMA therefore prescribes very little about execution. For example, DARMA prescribes nothing about the scheduling of tasks nor the implementation of the data structures (e.g., key-value store, tuple space) required to support coordination semantics. A back end runtime system scheduler is therefore free to use, for example, either depth-first or breadth-first priorities in deferred tasks (as captured in a CDAG). Similarly, a scheduler may use thread pools with work queues to manage tasks or it may use a fork-join model that creates new threads for each task. In this way, DARMA codes are execution model-agnostic, only requiring that a back end runtime system preserve the data-flow dependencies expressed in the application and derived by the translation layer.

DARMA furthermore prescribes nothing about the internals of each task. DARMA is fully compatible with parallel elastic tasks - task with flexible fine-grained parallelism, usually data parallelism. For example, depending on dynamic conditions, more or fewer threads may be requested for a GPU kernel. Although the DARMA front end API currently only allows expressing task granularity and task data-flow, we plan for the API to also express task elasticity in future versions.

DARMA-compliant back end runtime systems are required to enable an efficient SPMD launch of their program, similar to an MPI launch. This is based off application developer feedback, which has indicated that two of the most critical challenges for scientific applications with massive data parallelism in a task-based model include initial problem decomposition and distribution. DARMA's efficient SPMD runtime-based launch requirement will be modified if solutions are developed to support massive SPMD launches through compiler-based transformations.

1.5 Memory Model

The memory model for DARMA encompasses how variables are accessed and when updates become visible to parallel threads (concurrency). Within a DARMA execution stream, memory is local or private, and the standard C++ memory model applies. To share data between execution streams, DARMA uses a flat global memory space in which data is identified by unique tuple identifiers, i.e. a key-value store in which keys exist in a tuple space. Any object published into the key-value store can be read/written by any thread/process.

In DARMA a data handle is conceptually a reference counted pointer into the key-value store. Data handles are used to manage the complexities associated with task parallelism and inter-rank communication. When data needs to be made accessible off-`rank`, the application developer `publish`es the handle. Each handle has a globally unique handle ID (i.e, a `key` that is an arbitrary tuple of values into the key-value store). Before a `task` can begin, handle identifiers are resolved by the runtime system to a specific local address. Within the task, the standard C++ memory model applies.

When publishing, the user must specify an access group for that data. Declaring an access group informs the runtime system that other ranks currently need or will need the data, allowing the runtime to manage garbage collection and anti-dependency resolution. In most cases, the access group will be declared as the number of readers (1, in the case of simple point-to-point send). Once all read handles are released (go out of scope in C++ terms), garbage collection or anti-dependency resolution can occur.

In addition to facilitating coordination between ranks, handle data structures support sequential semantics (see Chapter 3 for details). Here concurrency is critical to the memory model and when/how updates data are made visible to parallel threads. Again, within tasks, the C++ memory model applies. At the task-level (coarse-grained), DARMA ensures atomicity of all tasks. The DARMA translation layer enforces the C++ sequential consistency model at the level of task in the same way that C++ ensures sequential consistency at the level of instructions. DARMA understands read/write usages of tasks and ensures that writes are always visible to subsequent reads - and reads always complete before subsequent writes. The use of handles enables this to happen automatically within an execution stream.

1.6 Data Model

DARMA only implements a data model through its serialization interface. The notion of data structure, data layout, and data type only exist in the application and translation layer (see Chapter 3). Thus, a runtime system implementing the DARMA specification is only aware of tuple or `key` identifiers for a coarse-grained data block of a given size. To actually migrate data, a back end runtime system invokes serialization hooks implemented by the application. In future versions, an API similar to the serialization interface will support the definition of data subsets and data slices. Again, the back end runtime system will only understand data and task dependencies, requiring the type-aware application and translation layers to define the details of subsetting and slicing operations. This leaves the application developer free to use arbitrary data structures, but puts more responsibility on the application developer to articulate the structure of the data.

1.7 Document organization

This document is organized as follows. In Chapter 2 we introduce the front end API. In Chapter 3 we provide a description of the translation layer, and in Chapter 4 we provide the specifics regarding what must be supported by each of the back end abstract classes in order to implement the DARMA specification. In Chapter 5 we include a list of application requirements driving the specification (along with a list of the applications contributing to the requirements to date). We conclude this document with Chapter 6, which includes a brief history of changes between previous versions of the specification, along with a list of the planned changes in upcoming versions. Appendix A provides a suite of examples that illustrate the front end API features.

Chapter 2

DARMA Front End API

2.1 Deferred Work Creation

In DARMA, like other AMT runtime systems, the application developer creates blocks of work (a task)and defines the preconditions for the task to begin executing. Rather than require application developers to explicitly define vertices and edges in a task-DAG or use explicit fork-join constructs, in DARMA, task preconditions are implicit in either the sequential order of tasks or the data-flow inherent in the key-value store coordination (more below). Deferred work is instantiated (but not necessarily executed) via the `create_work` function. For inline tasks (as compared to functor-based tasks, more below), this utilizes the C++ lambda capture mechanism to yield the following syntax:

```
//outer task
create_work([=]{
  // <-- deferred work in captured context
});
//continuing context in outer task
```

In DARMA, deferred work and task are generic terms we use for work performed by code inside the capturing lambda. This does not necessarily imply that the continuing context (after the `create_work`) will be executed before the captured work (note that "captured work" and "captured context" are two other generic terms we use interchangeably with deferred work). We highlight here that deferred work does not *need* to be deferred. A more precise term may therefore be *deferrable* work, but we use deferred to match previous literature. If a task's preconditions are all satisfied (data is available with correct permissions), the runtime system may execute it immediately. In fact, the runtime may execute the outer continuing context, the inner deferred context, or another context entirely if there are pending tasks.

While this syntax leverages C++ 11 lambdas, the user does not need to understand C++ 11 standard features to use `create_work` (this complexity is managed by DARMA's translation layer, as summarized in Chapter 3). All the work specified within a `create_work` is queued for deferred execution. The task does not need to execute immediately and may be executed by the back end runtime system any time after all of its preconditions are satisfied. Preconditions are either dependency (waiting for data to be produced) or anti-dependency (waiting for data to be released so it can be overwritten). Preconditions for a task are never given explicitly, but are instead derived implicitly based on sequential usage of `AccessHandle` objects, discussed in detail below. For example, to satisfy sequential semantics the following code should print "first: 42, second: 84":

```
auto my_handle = initial_access("some_data_key");
create_work([=]{
  my_handle.set_value(42);
});
create_work([=]{
  cout << "first: " << my_handle.get_value();
});
create_work([=]{
  my_handle.set_value(my_handle.get_value()*2);
});
create_work([=]{
  cout << ", second: " << my_handle.get_value();
});
```

The code produces results equivalent to a C++ code in which `create_work` is removed and `AccessHandle` is just replaced with the underlying type. These sequential semantics are pivotal to the DARMA programming model.

Sequential semantics provide a simple and intuitive way of coding asynchronous work, by limiting programmer burden, avoiding deadlock, and enabling runtime optimizations. However, in cases with massive SPMD parallelism across a distributed memory machine, it may be more scalable and natural to code in a CSP-like framework involving parallel execution streams. Rather than coding as if only operating within a single execution stream, the programmer must be aware of multiple parallel execution streams.

DARMA uses coordination data between parallel execution streams, rather than exchanging data through `send/recv` pairs. Two execution streams never explicitly exchange data. Instead they `publish` and fetch from a key-value store. Coordinating (rather than communicating) abstracts physical data locations to better support task migration. Additionally, it removes message-ordering requirements to better support asynchronous data transfers. While the key-value store appears to be a centralized, global data store that copies data in/out, the key-value store can be implemented as a DHT that supports zero-copy transfers. Thus both sequential semantics and coordination semantics follow the same principle in DARMA: intuitive programming model concepts that simplify reasoning about algorithms are transformed to a parallel, scalable execution by the translation layer and back end runtime system.

In the example here, variables are not passed down from a parent task to child tasks. Instead, one execution stream produces a value and publishes it to a key-value store. Another execution stream reads the value by fetching it from a key-value store. The processes coordinate with `publish/fetch` pairs similar to `send/recv` pairs in the CSP model of MPI.

Execution Stream 0:

```
auto sender =
   initial_access<int>("counter");
sender.set_value(42);
sender.publish(n_readers=1);
```

Execution Stream 1:

```
auto recver =
   read_access<int>("counter");
```

Instead of defining task preconditions implicitly via sequential order, task preconditions are specified more explicitly by requiring that a particular block of data be fetched from the key-value store. More on SPMD programs and parallel execution streams are given in Section 2.6.

2.2 Data Access Handles

`AccessHandle<T>` objects are lightweight wrappers around the actual data structure of interest having type `T`. The handles add a control block (metadata) that tracks uses of the handle and enforces sequential semantics, analogous to smart pointers that wrap pointer types and provide a reference counting control block. Critically, this interface is non-intrusive, wrapping any type `T` without requiring that type to be modified.

Most critically, an `AccessHandle` enables deferred access since `AccessHandle` can exist in ready or pending states. Ready and pending are not rigorously defined in the state table for `AccessHandle` in Section 2.2.3, but rather guiding concepts. Ready handles can be dereferenced (have their underlying values fetched) and be used immediately to perform work. Pending handles cannot be dereferenced, but can still be used to *schedule* or *instantiate* work. Thus even if a handle is carrying pending or unresolved data, execution can advance thereby unrolling more of the task graph.

This *lookahead* is the key element that enables runtime optimizations. Lookahead gives the runtime system more complete knowledge of the task graph instead of locally executing step-by-step. By looking ahead, the runtime system can reorder or migrate tasks to maximize data locality and improve load balance. The most critical conceptual change from standard C++ to DARMA are pending variables that enable lookahead, unlike conventional C++ variables that must always be “ready”.

Handles can be created three different ways.

1. a handle to data that does not yet exist in the system but needs to be created, or
2. a handle to data produced by another process that needs to be read, or
3. a handle to data produced by another process that needs to be overwritten or modified. Note that this type of handle does not exist in the current version of the specification.

Type 1 is denoted as `initial_access` in DARMA, which informs the runtime system that the data with the specified `key` does not yet exist, and the user intends to create this data.Hence, an `initial_access` data handle is usually followed by a memory allocation, and a value assignment. Remark: Although we could explicitly write out `AccessHandle<T>` in the code below, we *strongly encourage* programmers to use the C++ **`auto`** keyword. It will greatly increase code portability for future (potentially backwards-incompatible) versions of DARMA, with the additional benefit of decreasing code verbosity.

```
auto float_handle = initial_access<float>("float_key");
create_work([=]{
  float_handle.set_value(3.14);
});
```

As stated above, DARMA provides two methods for expressing task preconditions: sequential semantics and coordination semantics. `initial_access` is necessary in both methods. Once created, a handle can be passed along to subtasks within the same, sequential execution stream. A handle created by `initial_access` can also be `publish`ed, making it available to other execution streams via key-value store coordination.

Handle of Type 2 above request read-only access to data produced via external execution streams through `read_access` (which causes a fetch to be performed). As such, `read_acccess` is only relevant for applications that use coordination to express data flow.

```
auto float_handle = read_access<float>("another_float_key");
create_work([=]{
  float val = float_handle.get_value();
  std::cout << "Value read with key another_float_key is " << val;
});
```

Immediately following the `read_access` function, the `AccessHandle` will be *pending* instead of *ready*. To enable `get_value` to be called (put the handle in a ready state), the handle must be used inside of a `create_work`. `create_work` defers execution of the code block until the key-value store resolves the value of `float_handle` and converts it to a ready state. This might involve moving data if the **`float`** is on a remote node. Remark: future versions of DARMA will enable tasks to begin optimistically with some handles still in a pending state, but this is not supported in the current version of the specification.

In general, any calls to `get_value` should occur within a scoped code block to avoid dangling references to stale physical memory locations. Calls to `get_value` should go inside a `create_work` block when possible to guarantee availability of the data.

2.2.1 Publish

By default, unless explicitly published, data handles are visible only to tasks within the same scope (tasks that have a copy of the actual `AccessHandle<T>` object, created as discussed in Section 2.2). For data to be globally visible in the global memory space (key-value store), the application developer must explicitly `publish` data. Unpublished data will be reclaimed once the last handle referencing it goes out of scope (i.e refcount goes to zero), freeing the memory and resolving any anti-dependencies analogous to the destructor invocation in C++ when a class goes out of scope.

Published data, however, is globally visible to all execution streams and requires more "permanence." In order to resolve anti-dependencies associated with the publish or garbage collect the memory, published data must know its access group. When all read handles within an access group have been deleted or released *globally*, the memory holding the published data can be reclaimed. The easiest way to declare an access group (and currently the only supported method) is to simply give the total number of additional read AccessHandle<T> objects that will be created referring to it (recall that read AccessHandle<T> objects cause a fetch to be performed). In future versions, hints will be supported about which specific tasks will need to read data. This publish/fetch mechanism replaces an analogous MPI_Send/Recv or, for a publish with many readers, replaces an MPI_Bcast. In MPI, these function calls force an MPI_Send or MPI_Wait to block until the runtime system guarantees that the data has been delivered. An access group in DARMA provides a similar guarantee. Until all readers in an access group have received or released their data, DARMA cannot garbage collect or clear anti-dependencies.

```
auto float_handle = initial_access<float>("float_key");
create_work([=]{
  float_handle.set_value(3.14);
});
float_handle.publish(n_readers=1);
```

The n_readers specification in the publish call is a keyword argument (see Section 2.4) that informs the runtime system that the data (associated with float_key) will only ever be read once, and hence can be safely garbage collected soon after. This code provides similar functionality to an MPI send/receive.

As discussed above, handles can either be ready or pending. In reality, the distinction is more subtle. The "readiness" and "pendingness" can be different for read usages and write usages. Thus a handle can be read-ready, but modify-pending. This will be the case after publish operations. publish operations are treated as asynchronous read operations — that is, h.publish(...) is equivalent to

```
create_work(reads(h), [=]{...});
```

This means that the same precautions should be taken as with asynchronous reads. In particular, even if the handle was ready for modifying before publish, it is no longer valid to call h.set_value() after the publish. The asynchronous read done by the publish may or may not have occurred yet. In this scenario, one should use instead

```
create_work([=]{ h.set_value(...); });
```

to force the handle from a pending state to a ready state.

Publication Versions

If a handle is going to be published multiple times (or, more specifically, if the key with which the handle was created is going to be published multiple times), it needs to be published with a different version each time. A version is just like a key — an arbitrary tuple of values (see Section 2.2.2). For instance:

```
/** Execution stream 0 */
auto float_h =
  initial_access<float>("float_key");
auto int_h =
  initial_access<int>("int_key");
/* Execution stream 1 */
int_h.publish(n_readers=3, version=77);
//Use version() for multiple parts
float_ha.publish(n_readers=1,
    version("alpha",42));
```

```
/* Execution stream 1*/
auto my_int =  read_access<int>(
  "int_key", version=77);
auto my_float = read_access<float>(
  "float_key", version("alpha",42));
```

A version has similarities with an MPI tag, as they both ensure the uniqueness of data. However, unlike MPI which uses a combination of message order and tag to uniquely identify messages and match send/recv pairs, the DARMA asynchronous model does not allow implicit publication order to be used in matching publish/fetch pairs. Instead, all publications must uniquely identify each publication with a specific version.

2.2.2 Keys

In the examples in this section, the `key` to the `AccessHandle<T>` has always been a single string. A `key` in DARMA can be an arbitrary tuple of values. This makes it very easy for the application developer to create an expressive and descriptive `key` for each piece of data. Tuples can comprise different bit-wise copiable data types. The example at the end of Section 2.6 illustrates the use of the rank within the handle `key`. The following example shows the use of an aribitrary tuple as a `key`:

```
  int neighbor_id
  double other_identifier;

  // some code that sets neighborID and other_identifier

  auto float_handle = initial_access<float>("float_key",
                                            neighbor_id,
                                            other_identifier);
```

2.2.3 Handle Usage Rules

As alluded to above, handles are assigned states, and these states change based on the operations applied to them. The state of a handle encompasses both its read/write permissions and its "readiness." Pending handles can only be used for scheduling tasks while ready handles can be immediately used to do work. Here we more rigorously divide permissions into two main categories:

a *Scheduling*: Permissions a handle may use when instantiating tasks with `create_work`. These permissions apply independent of handle readiness (immediate permissions). Generally, this will be Read (handle may only used in read-only tasks) or Modify (handle may be used in read-only or read-write tasks).

b *Immediate*: Permissions that apply immediately, indicating the "readiness" of the handle. Immediate permissions can never be greater than scheduling permissions. A handle within a task can never have greater permissions doing immediate work than it can for instantiating deferred work.

For the two methods of creating handles, we have the following initializations.

- `initial_access<T>`: Initialized with scheduling modify, immediate none. The handle can be used in any mode when instantiating deferred work. However, the handle is not necessarily initialized and as such cannot be used immediately for reads or writes.
- `read_access<T>`: Initialized with scheduling read, immediate none. The handle can only be used for reads when instantiating deferred work. However, the handle is not necessarily initialized and as such cannot be used immediately.

To clarify, consider the following code:

```
//Predecessor outer state
create_work([=]{
  //Capture (inner) state
})
//Continuing outer state
```

In the outer task, a handle will have an initial pair of scheduling/immediate permissions (predecessor state). After the call to `create_work`, the handle's state will have changed, potentially losing some immediate permissions within the continuing outer state block. As specified currently, execution does not begin inside the captured work block until the handles it uses becomes ready. Inside the `create_work` (capture state), the handle's immediate and scheduling permisssions will therefore remain the same as they were in the predecessor outer state block.

Predecessor State		get_value		emplace_value set_value get_reference	
Scheduling permissions	Immediate permissions	Allowed?	Continuing as	Allowed?	Continuing as
None	None	No	-	No	-
Read	None	No	-	No	-
Read	Read	Yes	*Read/Read*	No	-
Modify	None	No	-	No	-
Modify	Read	Yes	*Modify/Read*	No	-
Modify	Modify	Yes	*Modify/Modify*	Yes	*Modify/Modify*

Table 2.1 Operations on the various states

Predecessor State		*read-only capture and publish*			*modify capture*		
Scheduling permissions	Immediate permissions	Allowed?	Capture Handle	Continuing Handle	Allowed?	Capture Handle	Continuing Handle
None	None	No	-	-	No	-	-
Read	None	Yes	*Read/Read*	*Read/None*	No	-	-
Read	Read	Yes	*Read/Read*	*Read/Read*	No	-	-
Modify	None	Yes	*Read/Read*	*Modify/None*	Yes	*Modify/Modify*	*Modify/None*
Modify	Read	Yes	*Read/Read*	*Modify/Read*	Yes	*Modify/Modify*	*Modify/None*
Modify	Modify	Yes	*Read/Read*	*Modify/Read*	Yes	*Modify/Modify*	*Modify/None*

Table 2.2 Deferred (capturing) operations on the various states.

Table 2.1 summarizes the state transitions involving these three handles following `create_work`.

To illustrate the importance of requesting the minimum permissions a task requires, consider the following:

```
auto float_handle = initial_access<float>("yet_another_float_key");
....
create_work(reads(float_handle), [=] {
  std::cout << "Value read with key yet_another_float_key is "
          << float_handle.get_value() << std::endl;
})
create_work(reads(float_handle), [=] {
  float val = float_handle.get_value();
  if (val > 0) std::out << "Value is positive" << std::end;
})
//read-write work down here
```

In this case, subtasks are created that only need read access. Without the `reads` qualifier, these tasks could not run in parallel (or out-of-order) since they would by default request read-write permissions. Sequential semantics would then require them to write in-order sequentially. This example highlights the importance that tasks only ever request the permissionsthey need. Over-requesting permissions will limit the amount of available parallelism in the code.

The distinction between immediate permissions and scheduling permissions is generally not explicit in the application. When a handle is created with `initial_acces` or `read_acces`, it is implicitly given immediate permissions of None. When a handle is used inside a task instantiated in the application with `create_work`, the back end runtime system (and translation layer) implicitly guarantees immediate permissions of Read or Write equal to the scheduling permissions. For simple cases, the application developer only needs to think of a single permission (not distinguishing scheduling/immediate). More more advances uses of DARMA (and features in future versions of the specification), an application developer will need to understand both scheduling permissions and immediate permissions.

To further illustrate, below is an *incorrect* usage of modify permissions

WRONG

```
initial_access<int> a
//a is in Modify/None
a.set_value(1) ✗
a.get_value()  ✗
```

CORRECT

```
initial_access<int> a
//a is in Modify/None
create_work([=]{ //modify capture
    a.emplace_value(1)   ✓
    a.set_value(1)       ✓
    a.get_reference()=1 ✓
});
```

Additionally, we demonstrate an *incorrect* usage of read permissions

WRONG

```
read_access<int> b
//b is in Read/None
b.get_value()   ✗
b.set_value(1)  ✗
create_work([=]{ //capture
  b.set_value(1) ✗
});
```

CORRECT

```
read_access<int> b
//b is in Read/None
create_work([=]{ // capture
  b.get_value()   ✓
});
```

2.2.4 Access Handles with Compile-time Checking

The AccessHandle<T> class actually has a second template argument, traits, that the translation layer uses to propagate static information about permissions, so that it can do as many compile-time checks as possible. The user should never directly specify traits for an AccessHandle<T> . Rather, DARMA returns a type with the correct compile-time traits from initial_access and read_access. DARMA also uses traits to implement the ReadAccessHandle<T> type alias used as a formal parameter to functors that need read-only permissions on a handle[1] (see § 2.3 for details of the DARMA functor interface). DARMA can take advantage of this to, for instance, raise a compile-time error if the user attempts to call set_value on a handle returned by read_access. Note that this will only work if the user gives the **auto** type specifier for the left-hand side of the assignment. The type AccessHandle<T> itself (i.e., with default traits template parameter) has completely unrestricted compile-time permissions, and thus implies no compile-time checking. However, unlike its more restricted analogs, it can hold handles with any permissions.

2.3 Creating Deferred Work using Functors

Thus far, the only method we've introduced for creating deferred work is using C++ lambdas. For instance,

```
auto h = initial_access<int>("my_key");
create_work([=]{ h.set_value(42); });
create_work(reads(h), [=]{
  cout << h.get_value() << endl; // prints "42"
});
```

While this is a useful shorthand that makes it easy to get simple programs up and running quickly, DARMA also provides a far more powerful and flexible mechanism for describing and creating deferred work: functors. While functors are significantly more verbose than the in-line lambda syntax, they are also much more feature rich and allow

[1]Note that ReadAccessHandle<T> is *not* the same as the return value of read_access. The former specifies *requirements* for a functor parameter, while the latter specifies bounds on the *available permissions* for a handle. The traits template parameter is different for these two, though the latter can be cast to the former.

DARMA to perform some additional optimizations that aren't available to lambdas because of the limitations inherent to the C++ language itself. The same piece of code from above can be written with functors:

```
struct SetTo42 {
  void operator()(AccessHandle<int> h) const {
    h.set_value(42);
  }
};

struct PrintIntValue {
  void operator()(int v) const {
    cout << v << endl;
  }
};

int darma_main(...) {
  /* ... */
  auto my_handle = initial_access<int>("my_key");
  create_work<SetTo42>(my_handle);
  create_work<PrintIntValue>(my_handle);
  /* ... */
}
```

Even though this code snippet is substantially more verbose than the lambda version, it provides some useful advantages. Most noticeably, the functors `SetTo42` and `PrintIntValue` are reusable, just like normal functions. They can be implemented in different files or even different translation units for code cleanliness and modularization.

There are some more subtle differences too, though. Notice that in `PrintIntValue`, `AccessHandle::get_value()` never needs to be called. As long as the type to which the `AccessHandle` refers is convertible to the formal parameter given in the functor call operator, DARMA will call `get_value` automatically. Also, since the formal parameter is a value (as opposed to a reference), DARMA can deduce at compile time that this `PrintIntValue` makes a read-only usage of its argument. (This would work the same way if the formal parameter had been **`int const`**`&`, a const lvalue reference). Even more subtly, the fact that the formal parameter for `PrintIntValue` isn't an `AccessHandle` communicates to DARMA at compile time that `PrintIntValue` won't schedule any tasks that depend on `my_handle` inside of `PrintIntValue` (we call this a leaf task with respect to `my_handle`), which is useful information that the back end runtime system can utilize to make informed scheduling decisions. To accomplish the same effect for a modify usage, we can give a formal parameter that is a non-**`const`** lvalue (e.g., **`int`**`&`). The `SetTo42` functor could then be rewritten:

```
struct SetTo42 {
  void operator()(int& val) const {
    val = 42;
  }
};
```

As you can see, the functor code starts to look very much like regular C++ code.

The lambda interface can still be mixed with the functor interface. For instance,

```
struct Compute42 {
  void operator()(AccessHandle<int> h) const {
    create_work([=]{
      h.set_value(21);
    });
    create_work([=]{
      h.set_value( h.get_value() * 2 );
    });
  }
};
```

As you can see, if we want to be able to schedule more deferred uses of a handle, we have to take an `AccessHandle` as a formal parameter.[2] If we want to pass on a read-only `AccessHandle`, we can do so by giving `ReadAccessHandle` [3] and the formal parameter type:

```
struct PrintIntReadHandle {
  void operator()(ReadAccessHandle<int> h) const {
    create_work([=]{
      std::ofstream f("42.txt");
      f << h.get_value() << endl;
    });
    create_work([=]{
      cout << h.get_value() << endl;
    });
  }
};
```

The nested tasks will request read permissions on `h`, just as if they had been created with `create_work(reads(h), ...)`. Moreover, any attempts to call `h.set_value()` inside of `PrintIntReadHandle` will result in a *compile-time* error (unlike in the pure lambda case), since DARMA knows at compile time that the `ReadAccessHandle<T>` is read-only.

2.3.1 Mixing Deferred and Immediate Arguments

In DARMA, deferred functor invocations can also take normal, value arguments. These can, of course, be mixed (i.e., an invocation can take some `AccessHandle` arguments and some value arguments in the same call). However, since deferred execution can be tricky, there are some special rules involved with value arguments to `create_work`.

- When the formal parameter to the functor is a non-const lvalue reference, deferred invocation of the functor can only be made with an `AccessHandle` as that argument. For instance,

```
int i;
auto j = initial_access<int>("mykey");;
create_work<SetTo42S>(i); // ✗ compile-error!
create_work<SetTo42>(j); // ✓
```

- When the formal parameter is a **`const`** lvalue reference, the deferred invocation must also take `AccessHandle` as argument. References cannot be made to regular C++ variables for deferred execution since, by deferring, the referred to variable may no longer exist. An implicit copy would be required that isn't apparent from the syntax to make the variable permanent. To do this, you need to either use a value formal parameter or explicitly use `darma_runtime::darma_copy`:

```
struct PrintIntRef {
  void operator()(const int& val) const { cout << val << endl; }
};
int darma_main(...) {
  /* ... */
```

[2]Equivalently, an lvalue reference or a **`const`** lvalue reference to an `AccessHandle` can be given. `AccessHandle` objects ignore **`const`**, and the copy overhead is negligible (though giving a reference parameter will be *slightly* more efficient)

[3]Note that is identical to `AccessHandle` in every way except that it is known to be read-only at compile time, whereas `AccessHandle` has unknown compile-time permissions. All compile-time qualified `AccessHandle` variants are castable to `AccessHandle` (and to any other `AccessHandle` variant with greater compile-time permissions)

```
6    int i = 42;
7    auto j = read_access<int>("mykey");
8    create_work([=]{ j.set_value(42); });
9    create_work<PrintIntRef>(i); // ✗ compile error, implicit copy of i
10   create_work<PrintIntRef>(42); // ✓ 42 is an rvalue (prvalue)
11   create_work<PrintIntRef>(j); // ✓ j is an AccessHandle<int>
12   create_work<PrintIntRef>(darma_copy(i)); // ✓ explicit darma_copy used
13   create_work<PrintIntRef>(std::move(i)); // ✓ i is an rvalue (xvalue)
14   /* ... */
15 }
```

This isn't a matter of constness, but one of reference-ness; even if `i` had been declared as **`const int`** `i = 42;`, line 9 would still be a compile-time error, since the reference may have expired by the time the deferred work actually runs and a copy needs to be made.

- Normal arguments by default can not be implicitly converted to `AccessHandle` formal parameters (whereas the reverse is allowed, and even encouraged when appropriate).

2.3.2 Functor Interface Pitfalls

- Rvalue references (`T&&`) can't be given as formal parameters to DARMA functors, and doing so may lead to unexpected and/or undefined compile-time and/or runtime behavior. This is related to how DARMA detects attributes of the formal parameters for functors, but it's also redundant — you can just use a regular value parameter. Because of the way deferred execution works, rvalues must be moved into storage until the actual invocation occurs anyway, so there is no savings from using an rvalue reference over a value parameter.

- The DARMA functor interface doesn't currently support deferred invocation of functors with templated call operators (the detection idiom we use woudn't work here, and besides, we wouldn't even know if the deduced type is supposed to be an `AccessHandle` or not!). Limited support for this may be available in the future. Note that the functor class itself can still be templated, as long as the functor's template parameters are given explicitly at the invocation site:

```
struct CantBeUsed {
  template <typename T>
  operator()(T val) const;
};
template <typename T>
struct ThisIsFine {
  operator()(T val) const;
};
int darma_main(...) {
  /* ... */
  auto h = initial_access<int>("hello");
  create_work<ThisIsFine<int>>(h);
  /* ... */
}
```

- Functors don't have state in DARMA. (Even if they did, there is no access to the functor instance at the call site, so that state wouldn't be useful).

2.4 Keyword arguments

Similar to higher-level languages like Python, the DARMA C++ interface allows the user to specify arguments to many of the API functions and constructs using either positional arguments or keyword arguments. In addition, many optional arguments may *only* be specified using keyword arguments. The syntax for specifying a keyword argument is identical to that of Python: `keyword=value`. For instance, if there is a function `some_function` in the DARMA API that accepts positional or keyword arguments `arg_a`, `count`, and `flag`, that function can be invoked equivalently in any of the following ways:

```
/* some_function signature:
 *  void some_function(std::string arg_a, int count, bool flag);
 */
// All of the following are equivalent:
some_function("hello", 42, true);
some_function(arg_a="hello", count=42, flag=true);
some_function(count=42, flag=true, arg_a="hello");
some_function("hello", flag=true, count=42);
```

Note that positional arguments may *not* be specified after the first keyword argument, and an argument cannot be specified more than once, even as a positional and keyword argument. Both of these lead to compile-time errors. Omitting a required argument is also a compile-time error, as is giving an argument of the incorrect type:

```
// Error: arg_a specified more than once
some_function("hello", 42, true, arg_a="whoops!");

// Error: missing required argument flag
some_function("hello", count=42);

// Error: cannot convert bool to std::string
some_function(arg_a=false, flag=true, count=42);
some_function(false, 42, true);

// Error: positional argument given after first keyword argument
some_function(arg_a="hello", 42, flag=true);
```

The enabling of Python-like keyword arguments introduces no runtime overhead. For those interested in C++ details, keyword arguments are accomplished using **`constexpr`** class instances with overloaded assignment operator, with arguments passed to the callable using perfect forwarding. More implementation details are given in Section 3.2.2.

2.5 Serialization and Layout Description

Any data that is migrated or moved across the network bewteen memory spaces must be first be serialized. The DARMA front end programming model provides an extremely flexible and extensible interface for describing serialization and/or layout of C++ types. In spite of this flexibility, the vast majority of use cases only require the understanding and use of one or two very basic abstractions. However, the DARMA serialization interface provides a wide variety of features to handle complex and corner cases, as well as features to tune and optimize performance-critical cases. The following section describes the DARMA serialization interface, beginning with abstractions that handle the vast majority of use cases and expanding to progressively more niche features later in the section.

Note that this section is entitled “Serialization and Layout Description” rather than just “Serialization” because the interface provides ways to specify movement of data in ways that aren’t traditionally considered serialization, such as describing a type as a series of remote direct memory access (RDMA) pointers with associated sizes. More details to follow.

2.5.1 Basic Intrusive Interface

The most basic and straightforward way to specify serialization of a user type in DARMA, and the method that should be used in the vast majority of cases (with, perhaps, one simple extension discussed below), is providing a publicly accessible[4] `serialize` method in the user class. The `serialize` method provided for this purpose should be non-**`const`** and should take a single argument, which in the simplest case will be an lvalue reference to a `darma_runtime::serialization::SimplePackUnpackArchive` object. For instance, consider the following (somewhat contrived) user-defined class:

```
class MyClass {
  private:
    double a_, b_;
    std::string label_;
    double prod_sqrt_;
  public:
    static constexpr const char unlabeled_string[] = "<unlabeled>";
    MyClass(int a, int b)
      : MyClass(unlabeled_string, a, b)
    { }
    MyClass(std::string const& label, int a, int b)
      : a_(a), b_(b), label_(label),
        prod_sqrt_(a_ == b_ ? a_ : std::sqrt(a_*b_))
    { }
};
```

The simplest way to allow DARMA to interact with `MyClass` is to provide a `serialize` method in the class definition:

```
using Archive = darma_runtime::serialization::SimplePackUnpackArchive;
class MyClass {
  public:
    /* ... */
    void serialize(Archive& ar) {
      ar | a_ | b_ | label_ | prod_sqrt_;
    }
};
```

As you can see, the type `SimplePackUnpackArchive` has an overload for **`operator`**`|()`, takes a serializable type, and returns itself (more on what constitues a "serializable type" later).

2.5.2 `SimplePackUnpackArchive`

DARMA encapsulates advanced serialization behaviors in the archive concept. The only archive type fully implemented in the current specification is `SimplePackUnpackArchive`, which performs serialization in the most basic and traditional way. On the sender side, DARMA performs two serialization passes: one in sizing mode and one in packing mode. The receiver only requires one pass: unpacking. These modes can be queried using the archive object methods `is_sizing()`, `is_packing()`, and `is_unpacking()`, only one of which will return true at any given time. All archive types implement these methods.

[4]later versions of the specification may allow private implementations with a **`friend`** specification

2.5.3 Generic Archive Serialization

As DARMA evolves and as more performance considerations are addressed, the DARMA team and our collaborators plan to provide other archive types which take more advanced serialization strategies, such as enabling RDMA access to pieces of a type. In order to write code that can take advantage of these features when they become available, the vast majority of user types can simply provide a templated `serialize` method:

```cpp
class MyClass {
  public:
    /* ... */
    template <typename Archive>
    void serialize(Archive& ar) {
      ar | a_ | b_ | label_ | prod_sqrt_;
    }
};
```

2.5.4 Different Behaviors in Different Modes

Consider again the `MyClass` example above. Since `prod_sqrt_` can be recomputed on the fly, it may be desirable to avoid including it in the data to be moved and instead just recompute it on the receiving side. To do this, however, we need the `serialize` method to perform different actions in unpacking mode than in the other modes. The `is_unpacking()` method makes this easy:

```cpp
class MyClass {
  public:
    /* ... */
    template <typename Archive>
    void serialize(Archive& ar) {
      ar | a_ | b_ | label_;
      if(ar.is_unpacking())
        prod_sqrt_ = a_ == b_ ? a_ : std::sqrt(a_*b_);
    }
};
```

Notice also that the `label_` field of `MyClass` has the same static value if a label is ungiven every time. If `MyClass` often does not have a `label_`, it may be advantagous to pack a boolean indicating whether the label exists, followed by the label itself only if the label is given. We can do this using the same approach:

```cpp
class MyClass {
  public:
    /* ... */
    template <typename Archive>
    void serialize(Archive& ar) {
      ar | a_ | b_;
      bool has_label;
      if(!ar.is_unpacking()) {
        has_label = label_ != unlabeled_string;
        ar | has_label;
        if(has_label) ar | label_;
      }
      else { // ar.is_unpacking()
        ar | has_label;
        if(has_label) ar | label_;
        else label_ = unlabeled_string;
        // From before:
        prod_sqrt_ = a_ == b_ ? a_ : std::sqrt(a_*b_);
      }
    }
};
```

2.5.5 Seperate Methods for Seperate Modes

If the logic for packing is significantly different from the logic for unpacking, the serialization of a `MyClass` object may involve a significant number of if statements. For this and other reasons, DARMA allows the user to specify seperate `pack`, `unpack`, and `compute_size` methods as needed. Each takes an archive object as an argument, and the `pack` and `compute_size` methods must be **`const`**. The first example that recomputes `prod_sqrt_` could then be rewritten as:

```
class MyClass {
  public:
    /* ... */
    template <typename Archive>
    void serialize(Archive& ar) {
      ar | a_ | b_ | label_;
    }
    template <typename Archive>
    void unpack(Archive& ar) {
      ar | a_ | b_ | label_;
      prod_sqrt_ = a_ == b_ ? a_ : std::sqrt(a_*b_);
    }
};
```

The more specialized `pack`, `unpack`, and `compute_size` methods always have higher priority than `serialize`, so in this case DARMA will invoke `unpack` during the unpacking pass while still calling `serialize` in the sizing and packing passes.

The performance-concious reader may have further noticed that since the **`operator`**`|()` implementation must function in all three modes, there will be branches or switches based on the mode. Thus, in a performance-critical context, the user may want operators that are specific to the phase in question. This could also be accomplished (and may be in the future) by passing in different types for the sizing, packing, and unpacking phases, which is yet another reason to use the templated versions of these methods instead. This can also be done using **`operator`**`<<()` for packing, **`operator`**`>>()` for unpacking, and **`operator`**`%` for sizing. The final `serialize` with both the `prod_sqrt_` and `label_` optimizations could then be rewritten as:

```
class MyClass {
  public:
    /* ... */
    template <typename Archive>
    void compute_size(Archive& ar) const {
      ar % a_ % b_;
      if(label_ == unlabeled_string) ar % false;
      else ar % true % label_;
    }
    template <typename Archive>
    void pack(Archive& ar) const {
      ar << a_ << b_;
      if(label_ == unlabeled_string) ar << false;
      else ar << true << label_;
    }
    template <typename Archive>
    void unpack(Archive& ar) {
      ar >> a_ >> b_;
      bool has_label;
      ar >> has_label;
      if(has_label) ar >> label_;
      else label_ = unlabeled_string;
      prod_sqrt_ = a_ == b_ ? a : std::sqrt(a_*b_);
    }
};
```

Unless absolutely performance critical, these optimizations should be avoided. Besides being significantly more verbose, these optimizations affect code maintainability. If another member variable `c_` were added to `MyClass`, the serialization implementation in the final example would have to be modified in *three* places, whereas the earlier examples, while potentially less performant, only have to be updated in one place. Also, failure to ensure that the order of member variable serialization is identical in multiple places can lead to hard-to-detect bugs. Thus, we recommend using the single `serialize` method except in performance-critical, inner-loop-like code.

2.5.6 Serializing Pointers and Ranges

DARMA also provides a convenient way to serialize iterables of serializable objects using `darma_runtime::serialization::range`. As a simple example:

```
template <typename T>
class MyData {
  private:
    T* data_;
    size_t n_items;
  public:
    MyData(T const* copy_from, size_t n)
      : n_items(n) {
      data_ = new T[n];
      std::copy(copy_from, copy_from+n, data_);
    }
    ~MyData() { delete[] data_; }
};
```

If we restrict ourselves to only making `MyData<T>` instances that hold serializable types `T`, we can write the `serialize` method for this class as

```
using darma_runtime::serialization::range;
template <typename T>
class MyData {
  public:
    /* ... */
    template <typename Archive>
    void serialize(Archive& ar) {
      ar | range(data_, data_ + n_items);
    }
};
```

2.5.7 Non-intrusive Interface

Classes for which the user cannot define an intrusive `serialize` method (or any of the other intrusive methods), for one reason or another, can still be made serializable by defining a specialization (partial or full) of the class `darma_runtime::serialization::Serializer<T>` for the type in question.[5] Like the intrusive interface, these classes can define a `serialize` method; individual `compute_size`, `pack`, and `unpack` methods; or some combination of these, with the specific versions having higher priority. All of these methods must be **`const`** (the Serializer object itself isn't allowed to have state anyway; it's just a convenient mechanism for grouping functions for a class non-intrusively). Their signatures are a bit different from the intrusive analogues. Consider a slightly different version of `MyClass` from above, the public interface of which is specified as:

```
class YourClass {
  public:
    double get_a() const;
    void set_a(double val);
    double get_b() const;
    void set_b(double val);
    std::string const& get_label() const;
    void set_label(std::string const& val);
    double get_product_sqrt() const;
};
```

Assuming `YourClass` is default constructible, a way to specify a serialization for `YourClass` non-intrusively is:

```
namespace darma_runtime { namespace serialization {
template <>
struct Serializer<YourClass> {
  template <typename Archive>
  void serialize(YourClass& yc, Archive& ar) const {
    if(!ar.is_unpacking()) {
      double a = yc.get_a();
      double b = yc.get_b();
      std::string label = yc.get_label();
      ar | a | b | label;
    }
    else {
      double a, b;
      std::string label;
      ar | a | b | label;
      yc.set_a(a);
      yc.set_b(b);
      yc.set_label(label);
    }
  }
};
}} // end namespace darma_runtime::serialization
```

As before, we can split this into `serialize` and `unpack` methods. However, the `unpack` method requires a slightly different signature:

```
template <typename Archive>
darma_runtime::serialization::Serializer<YourClass>::unpack(
  void* allocated, Archive& ar
```

[5]Generic implementations requiring partial specialization with an `enable_if` clause should use `darma_runtime::serialization::detail::Serializer_enabled_if<T, Enable>`. Consult source code for more details.

```
) const;
```

Rather than being a reference to an instance of the class itself, the first argument to the non-intrusive `unpack` method is a pointer to the beginning of a chunk of memory of size **`sizeof`**`(YourClass)` allocated by the backend, but not constructed. This allows for the unpacking of non-default-constructible classes. The `unpack` method must construct the object at that memory location using the C++ placement new. The syntax of placement new might be a little strange if you've never seen it before, but once you see it, its use is pretty straightforward. The non-intrusive `Serializer` for `YourClass` can then be written as:

```
namespace darma_runtime { namespace serialization {
template <>
struct Serializer<YourClass> {
  template <typename Archive>
  void serialize(YourClass& yc, Archive& ar) const {
    assert(!ar.is_unpacking()) // just in case
    double a = yc.get_a();
    double b = yc.get_b();
    std::string label = yc.get_label();
    ar | a | b | label;
  }
  template <typename Archive>
  void unpack(void* allocated, Archive& ar) const {
    double a, b;
    std::string label;
    ar | a | b | label;
    // Since YourClass is default-constructible, the placement new
    // that we want looks like this:
    YourClass* yc = new (allocated) YourClass();
    yc->set_a(a);
    yc->set_b(b);
    yc->set_label(label);
  }
};
}} // end namespace darma_runtime::serialization
```

The non-intrusive interface versions of `pack` and `compute_size` have similar signatures to that of `serialize`, except they take a **`const`** lvalue reference as their first argument:

```
template <typename Archive>
darma_runtime::serialization::Serializer<YourClass>::compute_size(
  YourClass const& val, Archive& ar
) const;
template <typename Archive>
darma_runtime::serialization::Serializer<YourClass>::pack(
  YourClass const& val, Archive& ar
) const;
```

The non-intrusive serialization interface has higher priority than the intrusive one, but in general the user should not define both or mixed intrusive and non-intrusive serializations.

2.5.8 Definition of "Serializable"

Having introduced the archive concept, the intrusive interface, and the non-intrusive interface, we're finally ready to formally define "serializable" as DARMA sees it. DARMA views the serializability of a given type as a property associated with that type and a given archive type — a type T can be described as “serializable with archive type A”. We’ve been sloppy about this up to this point because it’s usually clear from context which archive type we’re referring to (or if we’re referring to a generic archive type given as a template parameter). This allows for the development of archive types that, for instance, only handle performance sensative types, but do so very efficiently.

2.5.9 Implementations for Builtin and Standard Library Types

The DARMA translation layer has default `Serializer` implementations for many builtin and standard library types. Currently this includes anything that meets the standard container concept (for which the value types are also serializable), plain old data (POD) types, `std::pair` of serializable types, and compile-time sized arrays of serializable types. Many of the implementations in the current backend are with respect to a generic archive type, but since `SimplePackUnpackArchive` is the only archive type currently implemented, the code is only tested with this archive thus far.

You can define a serialization for any bitwise copyable, POD type simply by specializing `darma_runtime::serialization::serialize_as_pod<T>` to inherit from `std::true_type`:

```
class MyPlainOldData {
  int i, j, k;
  double x, y, z;
};
namespace darma_runtime { namespace serialization {
template <>
struct serialize_as_pod<MyPlainOldData> : std::true_type { };
}} // end namespace darma_runtime::serialization
```

2.5.10 Polymorphism

Deserialization into polymorphic base class pointers is currently not supported by the DARMA serialization interface. If a type is to be used in a context that requires DARMA to deserialize it (most importantly, as wrapped by an `AccessHandle<T>` or the type of an argument passed to a functor-style `create_work` that could be migrated), the concrete type must be known at compile time. Support for this will be forthcoming, but will likely require an intrusive interface. There are a number of other patterns in the programming literature that can be used to mimic run-time polymorphism, and we suggest the user consider these if necessary.

2.5.11 Serialization Pitfalls

- Because DARMA detects the various intrusive and non-intrusive `serialize`, `pack`, `unpack`, etc. methods, **`const`**-incorrectness or an otherwise incorrect signature can cause these methods to go undetected and lead to unexpected behavior. For instance,

```
class MyClass {
  public:
    /* ... */
    template <typename Archive>
    void serialize(Archive& ar) {
      ar | a_ | b_ | label_;
    }
    template <typename Archive>
    void unpack(Archive ar) { // ✗ missing lvalue reference in parameter!
      ar | a_ | b_ | label_;
      prod_sqrt_ = a_ == b_ ? a_ : std::sqrt(a_*b_);
    }
};
```

would fail to ever define prod_sqrt_ because the unpack method would not be detected and DARMA would fall back on serialize for the unpacking process. Most of the common mistakes we anticipate are checked with static_asserts, but it is impossible to check all possible mistakes. Care should be taken when this issue could arise. Note that if only serialize had been defined incorrectly, for instance:

```cpp
class MyClass {
  public:
    /* ... */
    template <typename Archive>
    void serialize(Archive& ar) const { // ✗ should not be const!
      ar | a_ | b_ | label_ | prod_sqrt_;
    }
};
```

then the code that uses MyClass would simply fail to compile, since MyClass isn't serializable with any archive types.

2.6 SPMD support

Most applications written in or ported to DARMA will likely have SPMD as the dominant form of parallelism. To simplify the implementation of SPMD-structured codes, the notion of a rank is maintained within the API. Again, rather than rely entirely on sequential semantics in cases of massive data parallelism, many independent parallel execution streams can begin simultaneously and coordinate via the key-value store. Each execution stream is assigned a unique rank ID, analogous to the MPI rank assigned to processes in a MPI communicator. The initialization and termination of the runtime in each execution stream is via the calls darma_init and darma_finalize. The total number of SPMD execution streams are queried with the call darma_spmd_size, and the rank ID of a particular execution stream is queried with darma_spmd_rank. A typical user written main program will look as follows:

```cpp
int darma_main(int argc, char**argv){
  darma_init(argc, argv);
  size_t n_ranks = darma_spmd_size();
  size_t me = darma_spmd_rank();
  ...
  darma_finalize();
  return 0;
}
```

The rank is a very useful concept to orchestrate dependencies in a SPMD model since data pertaining to a rank can be associated with keys that utilize the rank for uniqueness. The example below illustrates this concept, where the rank is integral to the key associated with data originating on that rank.

```cpp
size_t me = darma_spmd_rank();
auto data_handle = initial_access<double>("data_key", me);
```

Note that in DARMA, SPMD ranks are actually just a special kind of task that happens to have a name containing the rank, and can be treated as such. In most cases, these named tasks (ranks) will be execution streams, independent tasks with no parent. However, the similarity to traditional, MPI-style SPMD upon launch should improve the ease of porting and scalability significantly.

We emphasize again that within DARMA's supprot for SPMD, coordinating (rather than communicating) abstracts physical data locations to better support task migration. Additionally, it removes message-ordering requirements to better support asynchronous data transfers. We further reiterate that even though the key-value store appears to the application developer to be a traditional data store, it can be implemented in a scalable distributed fashion.

2.7 API: Creating and Managing Work

In this section we provide details regarding the DARMA-0.3.0-alpha API.

2.7.1 darma_main

Summary
darma_main is the entry point DARMA uses to launch the user's code.

Syntax

```
int darma_main(int argc, char** argv)
```

Details
The signature of darma_main mimics the signature of **int** main(), which C++ uses as the entry point to user code.

Code Snippet

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  return 0;
}
```

Figure 2.1 Basic usage of darma_main.

2.7.2 `darma_init`

Summary
`darma_init` initializes the DARMA execution environment for a rank.

Syntax

```
void darma_runtime::darma_init(int& argc, char**& argv);
```

Positional Arguments

- argc: command line arguments count.
- argv: array arguments.

The input parameters are the command line argument count and array arguments provided to main. Note that the back end will process and remove any DARMA back end-specific arguments from these, leaving any application-specific arguments untouched.

Details
Must be called exactly once per rank ("exactly once" may change in later spec versions) before any other DARMA function is called. Together with `darma_finalize` (see § 2.7.3), this creates an execution stream that defines a DARMA rank.

Code Snippet
See code for `darma_init` in Figure 2.2.

2.7.3 `darma_finalize`

Summary
`darma_finalize` the DARMA execution environment for a rank.

Syntax

```
void darma_runtime::darma_finalize();
```

Positional Arguments
None.

Details Called to signify the end of the execution stream that defines a DARMA rank. At least by the time this function *returns*, the back end guarantees that all work (tasks) created between the corresponding `darma_init` call and this invocation, as well as all of the decendents of that work, must be completed. No user-level DARMA operations are allowed after this call, though the implicit invocation of the destructors of `AccessHandle` objects (at, e.g., the final closing brace of `darma_main`) is allowed. Must be called exactly once for each call of `darma_init` (which, in turn must be called exactly once per rank in the current version of the specification).

Code Snippet

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;

  darma_init(argc, argv);
  std::cout << "DARMA initialized" << std::endl;

  // code goes here

  std::cout << "Finalizing DARMA..." << std::endl;
  darma_finalize();
  return 0;
}
```

Figure 2.2 Basic usage of `darma_init` and `darma_finalize` to initialize and finalize environment.

Restrictions and Pitfalls

- `darma_finalize` should be called at the outermost task depth on a rank. In other words, it should *never* be called from within a `create_work` or other asynchronous context.

2.7.4 darma_spmd_size

Summary
darma_spmd_size the number of ranks (or execution streams) in the DARMA environment.

Syntax

```
/* unspecified */ darma_runtime::darma_spmd_size();
```

Positional Arguments
None.

Return
An object of unspecified type that may be treated as a std::size_t giving the number of ranks in the DARMA environment.

Details
This function gives the number of ranks or execution streams DARMA is executing the program with. Specifically, it is the number of times the back end has invoked darma_main anywhere in the system for this particular run of the program (and thus, it is also the number of times the back end expects the user to invoke darma_init).

Code Snippet

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;
  darma_init(argc, argv);

  const size_t size = darma_spmd_size();
  // ...

  darma_finalize();
  return 0;
}
```

Figure 2.3 Basic usage of darma_spmd_size.

Restrictions and Pitfalls

- The value returned by this function will always return **true** for greater-than comparison with 0, and will always be convertible to a std::size_t with a value greater than 0.
- The return type is unspecified to allow future expansion to generalized ranks. For instance, future versions of the specification may allow the user to request the rank as an {x, y, z} tuple of indices in a structured lattice.

2.7.5 `darma_spmd_rank`

Summary
`darma_spmd_rank` returns the rank index associated with the execution stream from which this function was invoked.

Syntax

```
/* unspecified */ darma_runtime::darma_spmd_rank();
```

Positional Arguments
None.

Output
An object of unspecified type that may be treated as a `std::size_t` which is less than the value returned by `darma_spmd_size`.

Details
This function returns the rank index of the calling DARMA execution stream. If the value returned by `darma_spmd_size` is convertible to a `std::size_t` with the value N, then the value returned by this function will be convertible to a `std::size_t` with the value r, which will always satisfy $0 <= r < N$. Furthermore, the type of the value returned by this function will always be directly comparable to the type returned by `darma_spmd_size` and to 0 such that this previous condition is met. The value returned is also equality comparable with 0, the value returned will be true for equality comparison with 0 on exactly one rank. The value returned by this function will be unique on every rank (in the equality sense), and will be the same across multiple invocations of the function within a given rank. The value returned will also be the same at any asynchronous work invocation depth within a rank's execution stream, regardless of whether that work gets stolen or migrated.

Code Snippet

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;
  darma_init(argc, argv);

  // get my rank
  const size_t myRank = darma_spmd_rank();
  // get size
  const size_t size = darma_spmd_size();

  std::cout << "Rank " << myRank << "/" << size << std::endl;

  darma_finalize();
  return 0;
}
```

Figure 2.4 Basic usage of `darma_spmd_rank`.

2.7.6 `create_work`

Summary
`create_work` instantiates deferred work to be executed by the runtime system.

Syntax

```
// Functionally:
create_work([=]{
  // Code expressing deferred work goes here
});
// or:
create_work(
  ConstraintExpressions...,
  [=]{
    // Code expressing deferred work goes here
  }
);
// or:
create_work<FunctorType>(ArgumentsToFunctor...);

// Formally:
/* unspecified */ create_work(Arguments..., LambdaExpression);
/* unspecified */ create_work<Functor>(Arguments...);
```

Positional Arguments

- `LambdaExpression` A C++11 lambda expression with a copy default-capture (i.e., `[=]`) and taking no arguments. More details below.
- `ConstraintExpressions...` (optional) If given, these arguments can be used to express modifications in the default capture behavior of `AccessHandle<T>` objects captured by the `LambdaExpression` given as the final argument. In the 0.3.0-alpha-specification, the only valid permission modification expression is the return value of the `reads()` modifier (see § 2.10.1), which indicates that only read operations are performed on a given handle or handles within the `LambdaExpression` that follows.
- `ArgumentsToFunctor...` In the deferred functor invocation version, these arguments are pattern-matched with the formal parameters of the functor, causing the deferred invocation to invoke the call operator of `FunctorType` with arguments derived from these as described in § 2.3. Constraint expressions may also be used in the corresponding positional argument spots for a given `AccessHandle<T>` argument.

Return
Currently **`void`** in the 0.3.0-alpha-specification, but may be an object of unspecified type in future implementations.

Details
This function expresses work to be executed by the runtime system. Any `AccessHandle` variables used in the `LambdaExpression` or given in `ArgumentsToFunctor...` will be captured and made available inside the capturing context or `FunctorType` call operator as if they were used in sequence with previous capture operations or deferred functor invocations with the same handle. Depending on the scheduling permissions available to the `AccessHandle<T>` at the time of `create_work` invocation and on the `ConstraintExpressions...` given as arguments, this function call expresses either a *read-only capture* or a *modify capture* operation on a given handle (see § 2.2.3). If a handle `h` has *Read* scheduling permissions when it is captured or if the explicit constraint

expression `reads(h)` is given in the `ConstraintExpressions...` arguments, `create_work` functions as a *read-only capture* operation on that handle. Otherwise, it functions as a *modify capture*. Formal parameters to the `FunctorType` call operator can also affect the type of capture operation that is performed, as discussed in § 2.3.

Additional general discussion on use of `create_work` can be found in § 2.1.

Code Snippet

```
create_work([=]{
  std::cout << " Hello world! " << std::endl;
});
```

Figure 2.5 Basic usage of `create_work`.

Restrictions and Pitfalls

Most of the general restrictions and pitfalls related to `create_work` are discussed in § 2.1. Some more technical restrictions are given here.

- Because of the way in which `create_work` is implemented, placement of multiple `create_work` operations on the same line of code will not compile. For instance:

  ```
  // ✗ does not compile, gives cryptic error message
  create_work([=]{}); create_work([=]{});
  ```

  This is particularly easy to accidentally do when defining preprocessor macros:

  ```
  // ✗ does not compile, gives even more cryptic error message
  #define foo(...) __VA_ARGS__
  foo(
    create_work([=]{});
    create_work([=]{});
  )
  ```

  Note that this is not a problem when using nested `create_work` calls:

  ```
  // ✓ not a problem
  create_work([=]{ create_work([=]{}); }); // works fine
  ```

  Other than the obvious solution of putting the `create_work` invocations on multiple lines, this issue can be worked around by putting any of the later `create_work` calls within their own scopes:

  ```
  // ✓ works fine
  create_work([=]{}); { create_work([=]{}); }
  // ✓ also fine
  foo(
    create_work([=]{});
    { create_work([=]{}); }
    { create_work([=]{}); }
  )
  ```

2.8 API: Data Access Handles

In this section, we discuss the functions that create handles in the DARMA-0.3.0-alpha API.

2.8.1 `initial_access`

Summary

`initial_access` creates a handle to data that does not yet exist in the key-value store but needs to be created.

Syntax

```
AccessHandle<T> darma_runtime::initial_access<T>(arg1, arg2, ...);
```

Positional Arguments

arg1, arg2, ...: arbitrary tuple of values defining the `key` of the data.

Return

An object of unspecified type that may be treated as an `AccessHandle<T>` with the `key` given by the arguments.

Details

This construct creates a handle to data that does not yet exist but needs to be created. The handle is created with *Modify* scheduling permissions and *None* immediate permissions. The function takes as input an arbitrary tuple of values. Note that this `key` has to be unique (see Section 2.2.2). One cannot define two handles with the same `key`, even if they are created by different ranks. One basic way to ensure this is the case is to always use the rank ID as one component of the `key`.

Code Snippet

```
auto my_handle1 = initial_access<double>("data_key_1", myRank);
auto my_handle2 = initial_access<int>("data_key_2", myRank, "_online");
```

Figure 2.6 Basic usage of `initial_access`.

Restrictions and Pitfalls

- Because the actual type returned by `initial_access<T>` is unspecified, you should generally use **`auto`** instead of naming the type on the left hand side of the assignment (this is generally a good idea in modern C++). In other words,

```
// ✓ good, preferred
auto my_handle1 = initial_access<double>("good");

// ✗ still compiles, but not preferred (may miss out
//   on some future optimizations and compile-time checks)
AccessHandle<double> my_handle1 = initial_access<double>("bad");
```

For more, see § 2.2.4

2.8.2 read_access

Summary
read_access<T> creates a handle with read-only access to data that has been or will be published elsewhere in the system.

Syntax

```
/* unspecified, convertible to AccessHandle<T> */
darma_runtime::read_access<T>(KeyParts..., version=KeyExpression);
```

Positional Arguments

- KeyParts...: tuple of values identifying the key of the data to be read.

Keyword Arguments

- version=KeyExpression (or version(KeyExpression...), see § 2.4 for multiple-right-hand-side keyword argument usage): the version used to publish the data to be accessed. The value can be an arbitrary KeyExpression.

Return
An object of unspecified type that may be treated as an AccessHandle<T> with the key given by the arguments.

Details
This function creates a handle to data that already exists and needs to be accessed with read-only privileges. It takes as input the tuple of values uniquely identifying the data that needs to be read. Immediately following this function, the handle will have Read scheduling permissions and None immediate permissions. The key-version requested must eventually match that of a key-version that was published.

In general, read_access data is migratable and potentially stored off-node.

Code Snippet

```
/* on one rank: */
auto my_handle1 = initial_access<double>("key_1");
create_work([=]{
  my_handle1.emplace_value(5.3);
});
my_handle1.publish(n_readers=1, version="final");

//...

/* potentially on another rank: */
auto readHandle = read_access<double>("key_1", version="final");
create_work([=]{
  std::cout << readHandle.get_value() << std::endl;
});
```

Restrictions and Pitfalls

- Because the actual type returned by `read_access<T>` is unspecified, you should generally use **`auto`** instead of naming the type on the left hand side of the assignment (this is generally a good idea in modern C++). In other words,

  ```
  // ✓ good, preferred
  auto my_handle1 = read_access<double>("good", version=17);

  // ✗ still compiles, but not preferred (may miss out
  //   on some future optimizations and compile-time checks)
  AccessHandle<double> my_handle1 = read_access<double>("bad", version="oops");
  ```

  For more, see § 2.2.4.

2.9 API: `AccessHandle` methods

In this section, we describe the methods that can be called on `AccessHandle<T>` objects (i.e., the objects returned by the `initial_access` and `read_access` functions).

2.9.1 `emplace_value`

Summary
`emplace_value` constructs an object of the type pointed to by an `AccessHandle<T>` object (that is, `T`) in place by forwarding the arguments to the constructor for `T`.

Syntax

```
// functional:
some_handle.emplace_value(arg1, arg2, ...);

// Formal:
void AccessHandle<T>::emplace_value(Args&&... args);
```

Positional Arguments

- `args...` (deduced types): Arguments to forward to the constructor of `T`.

Details

`AccessHandle<T>::emplace_value(...)` mimics the syntax for in-place construction in standard library containers. See, for instance, `std::vector<T>::emplace_back(...)`. If in-place construction is unnecessary or undesired, `set_value` can be used instead. Note that calling `emplace_value` on a handle requires *Modify* immediate permissions (see § 2.2.3). If a previously constructed value exists (or a default constructed value, if possible) for the value held by the `AccessHandle<T>`, it will be destroyed via `T::~T()`.

Code Snippet

```
struct LoudMouth {
  LoudMouth(int i, double j) { cout << "Ctor: " << i << ", " << j << endl; }
};
auto h = initial_access<LoudMouth>("key");
create_work([=]{
  h.emplace_value(42, 3.14); // prints "Ctor: 42, 3.14"
});
```

Figure 2.7 Basic usage of `emplace_value`.

Restrictions and Pitfalls

- In the current version of the specification, types that are default constructible will always be default constructed before first use. For non-default-constructible types, however, memory of the correct size (i.e., `sizeof(T)`) will only be allocated, but no constructor will be called. The user must call `emplace_value(...)` before performing any operations on the data (or risk undefined behavior).

2.9.2 `publish`

Summary
`publish` the data pointed to by a given handle so that it can be retrieved on other DARMA ranks.

Syntax

```
void
AccessHandle<T>::publish(n_readers=..., version=...)
```

Positional Arguments
None.

Keyword Arguments

- `n_readers=size_t` (optional): informs the runtime system how many times `read_access` will be called in order to access this data. If omitted, it defaults to 1.
- `version=KeyExpression` (or `version(KeyExpression...)`, see § 2.4 for multiple-right-hand-side keyword argument usage) (optional): informs the runtime system what version to associate with the data being published. The value can be an arbitrary `KeyExpression`. If omitted, the version defaults to an empty `key` (i.e., a `key` tuple with zero components). Omitting this keyword implicitly indicates to the runtime system that the handle (or any handle with the same name `key`) will not be published again in the remaining lifetime of the program.

Details
Publish the data associated with a given handle `h` such that it can be retrieved `n_readers` times anywhere via a `read_access` invocation that gives the same name key as `h` and the same `version` `key` as the one given to the keyword argument to `publish`. A `publish` is a *read-only capture* operation (see § 2.2.3).

Code Snippet

```
auto me = darma_spmd_rank();
assert(darma_spmd_size() >= 2);
if(me == 0) {
  auto my_handle = initial_access<double>("key_1");
  create_work([=]{
    my_handle.emplace_value(5.3);
  });
  my_handle.publish(n_readers=1, version="only");
}
else if(me == 1) {
  auto my_handle = read_access<double>("key_1", version="only");
  create_work([=]{
    cout << my_handle.get_value() << endl; // prints "5.3"
  });
}
```

Figure 2.8 Basic usage of `publish`.

Restrictions and Pitfalls

- publish puts more burden on the programmer to avoid race conditions and deadlock, which are automatically avoided when relying entirely on sequential semantics. This is similar to deadlock situations with sends and receives in MPI in which communicating processes block on a receive before sending to each other. For instance, the following snippet deadlocks:

  ```
  // This code deadlocks!
  auto me = darma_spmd_rank();
  assert(darma_spmd_size() >= 2);
  if(me == 0) {
    auto h1 = initial_access<int>("key", 0);
    auto h2 = read_access<int>("key", 1);
    create_work([=]{
      h1.set_value(42);
      h1.publish();
      cout << h2.get_value() << endl;
    });
  }
  else if (me == 1) {
    auto h3 = initial_access<int>("key", 1);
    auto h4 = read_access<int>("key", 0);
    create_work([=]{
      h3.set_value(73);
      h3.publish();
      cout << h4.get_value() << endl;
    });
  }
  // Deadlock! (eventually, at the latest when darma_finalize() is
  // called): neither of the above create_work()s can ever run
  ```

  This snippet deadlocks because a dependency loop has been created between two publish/read_access pairs. While the deadlock is relatively obvious here, it can be much more difficult to decipher in a more complex code, especially if, for instance, h1 and h2 are arguments to a function, or if the parts of the keys used to construct the handles are variables with values dependent on some previous computation.
- It is particularly easy to create deadlock scenarios by publishing a handle and fetching it within the same rank. For this reason, we recommend extreme caution when this scenario could arise, and in general we suggest that the user should avoid doing so if at all possible.
- Since publish is a *read-only capture* operation, it must have scheduling permissions of *Read* or *Modify*; calling publish on a handle with other scheduling permissions is a runtime error. Also, as with all *read-only capture* operations, calling publish on a handle with *Modify* immediate permissions results in a handle with *Read* immediate permissions in the continuing context. See § 2.2.3 for more details. For example, the following code results in a runtime error at the marked line:

  ```
    auto h = initial_access<int>("key");
    create_work([=]{
      h.set_value(5);
      h.publish();
      h.set_value(10); // ✗ h does not have Modify immediate permissions
    });
  ```

- It is an error to call publish on a handle with the same codlinkkey and version more than once.
- If publish is called on a given handle without the version keyword argument, it is an error to call publish again on that handle or any other handle with the same name key for the remaining lifetime of the program. Note that because of the default behavior of the version keyword argument, giving an explicit version that

is the empty `key` (e.g., `h.publish(version())` or `h.publish(version=make_key())`) will lead to this same behavior.

2.9.3 `get_value`

Summary `get_value` accesses the data pointed to by a handle in a read-only manner.

Syntax

```
const T& AccessHandle<T>::get_value();
```

Positional Arguments
None.

Return
A const reference to the data associated with the handle.

Details
Calling `get_value` on a handle requires *Read* or *Modify* immediate permissions (see § 2.2.3).

Code Snippet

```
AccessHandle<double> my_handle = initial_access<double>("key_1", myRank);
create_work([=]{
  my_handle.set_value(3.14);
});
create_work(reads(my_handle), [=]{
  cout << my_handle.get_value() << endl; // prints "3.14"
});
```

Figure 2.9 Basic usage of `get_value`.

Restrictions and Pitfalls

- Do not hold the reference returned by this method across an asyncronous operation on the source handle. For example, the following results in undefined behavior:

  ```
  auto h = initial_access<int>("my_key");
  create_work([=]{ h.set_value(5); });

  create_work([=]{
    auto const& v = h.get_value();
    create_work([=]{ h.set_value(10); });
    cout << v << endl; // ✗ undefined behavior!!
  });
  ```

  Instead, to be safe, we recommend that when mixing synchronous and asynchronous code, enclose assignments and their corresponding uses in their own scope:

```
auto h = initial_access<int>("my_key");
create_work([=]{ h.set_value(5); });
create_work([=]{
  { // begin scope for v
  auto const& v = h.get_value();
  cout << v << endl;
  } // ✓ prevent accidental usage of v after the create_work using h
  create_work([=]{ h.set_value(10); });
  // uses of v here are now a compile-time error rather
  // than undefined behavior
});
```

2.9.4 `set_value`

Summary
`set_value` sets the value of the data pointed to by a handle.

Syntax

```
template <typename U>
void AccessHandle<T>::set_value(U&& value)
```

Positional Arguments

- value (type convertible to `T`): The new value for the data.

Details
This invokes `T::`**`operator`**`=(U&&)` (`T`'s assignment operator to a universal reference to `U`) with the argument `value`. If the type `T` has no assignment operator for this type, calling `set_value` will be a compile-time error. If you need to invoke an in-place constructor instead, use `emplace_value`.

Code Snippet

```
auto h = initial_access<double>("key_1");
create_work([=]{
  h.set_value(55.343);
});
```

Figure 2.10 Basic usage of `set_value`.

Restrictions and Pitfalls

- The specification of the method is likely to change in the future to be analogous to the behavior of, e.g., `std::vector<T>::push_back()` (as it relates to `std::vector<T>::emplace_back()`). If this could be a problem for `T`, you should probably use `emplace_value` for now.

2.9.5 get_reference

Summary
`get_reference` gets a non-constant reference to the data pointed to by the handle.

Syntax

```
T& AccessHandle<T>::get_reference()
```

Positional Arguments
None.

Return
A non-constant reference to the data.

Details
This method requires *Modify* immediate permissions. See § 2.2.3 for more information on immediate permissions.

Code Snippet

```
AccessHandle<double> my_handle1 = read_access<double>("key_1", myRank);
create_work([=]{
  my_handle1.get_reference() = 242.343;
});
```

Figure 2.11 Basic usage of `get_reference`.

Restrictions and Pitfalls

- Do not hold the reference returned by this method across an asyncronous operation on the source handle. For example, the following results in undefined behavior:

  ```
  auto h = initial_access<int>("my_key");
  create_work([=]{ h.set_value(5); });

  create_work([=]{
    auto& v = h.get_reference();
    create_work([=]{ h.set_value(10); });
    cout << v << endl; // ✗ undefined behavior!!
  });
  ```

  See recommendations in § 2.9.3 for more.

2.9.6 operator->

Summary

operator-> is a dereference operator to directly access the object pointed to by the handle.

Syntax

```
T* AccessHandle<T>::operator->();
```

Input Parameters

None.

Return

Returns a reference to the data pointed to by the handle.

Details

Just like `set_value` and `get_reference`, this operator requires *Modify* immediate permissions to invoke safely. Unlike `set_value` and `get_reference`, however, the **operator->** can also be invoked on handles that only have *Read* immediate permissions. In that case, it is up to the user to ensure that only **const** methods are called on the resulting object. In other words, `AccessHandle<T>::`**operator**`->()` lets you "shoot yourself in the foot." If more safety is desired, use the more verbose forms with `set_value` and `get_reference`.

Code Snippet

```
//...
typedef std::vector<double> vec;
AccessHandle<vec> my_handle2 = initial_access<vec>("key_2", myRank);

create_work([=]{
  my_handle2.emplace_value(0.0);
  my_handle2->resize(4);
  double * vecPtr = my_handle2->data();
});
```

Figure 2.12 Basic usage of :operator->.

2.9.7 `get_key`

Summary
`get_key` gets the `key` identifying the data pointed to by the handle.

Syntax

```
darma_runtime::types::key_t const& AccessHandle<T>::get_key();
```

Positional Arguments
None.

Return
The `key` identifying the data.

Details
This method can be called at any time after the handle is created. It does not require any scheduling permissions nor immediate permissions.

Code Snippet

```
//...
auto myRank = darma_spmd_rank();
AccessHandle<double> my_handle1 = read_access<double>("key_1", myRank);
auto myK = my_handle1.get_key();

create_work([=]{
  my_handle1.get_reference() = 242.343;
  auto myK = my_handle1.get_key();
  assert(myRank == myK.get_key().component<1>().as<int>());
});
```

Figure 2.13 Basic usage of `get_key`.

2.9.8 =0 or release

Summary
release or =0 releases the reference to the data held by the handle.

Syntax
These two are equivalent.

```
// Functional:
some_handle = 0;
some_handle.release()

// Formal
void AccessHandle<T>::operator=(std::nullptr_t);
void AccessHandle<T>::release();
```

Positional Arguments
None.

Return
None.

Details
Release the reference to the underlying data held by a given handle. Note that this effectively only decrements the reference count; the data itself will not be deleted unless there are no other existing handles referring to it. Releasing at the earliest possible time can help avoid some deadlock situations, particularly with published data, and potentially increase concurrency.

Code Snippet

```
// ...
AccessHandle<double> my_handle1 = initial_access<double>("key_1", myRank);
create_work([=]{
  my_handle1.get_reference() = 242.343;
});
my_handle1.release();
// ...
```

Figure 2.14 Basic usage of =0 or release.

2.10 API: Keywords

In this section, we describe the keywords of the DARMA-0.3.0-alpha API.

2.10.1 `reads`

Summary
`reads` is a keyword argument for `create_work` that constrains permissions of a set of handles to be read-only within that task.

Syntax

```
create_work(reads(handles...), [=]{
  // code
});
```

Positional Arguments

- `handles...`: list of `AccessHandle<T>` objects to constrain to read-only privileges.

Details
Used as a keyword argument to a `create_work` to constrain permissions for a list of handles to be read-only within that task. It can contain a single handle or a list of handles.

Code Snippet

```
// ...
auto my_handle = initial_access<double>("data", myRank);
create_work([=]{
  my_handle.emplace_value(0.55);
});
create_work(reads(my_handle), [=]{
  std::cout << " " << my_handle.get_value() << std::endl;
  my_handle.set_value(3.14); // ✗ runtime error
});
// ...
```

Figure 2.15 Basic usage of `reads`.

Restrictions and Pitfalls

- This can only be called as keyword argument to `create_work`. Use in other contexts will lead to compile-time errors, run-time errors, or undefined behavior.

2.10.2 `n_readers`

Summary
`n_readers` is a keyword argument to `publish`.

In namespace `darma_runtime::keyword_arguments_for_publication`.

2.10.3 `version`

Summary
`version` is a keyword argument to `publish` and `read_access<T>`.

In namespace `darma_runtime::keyword_arguments_for_publication`.

Chapter 3

Translation Layer

A key design principle of DARMA is the ability to explore the design space of back end AMT runtime system implementations without requiring changes in the application code. Since the front end API is essentially an EDSL and most back end runtime systems with which we want to interface use traditional C or C++ constructs, a layer is needed that translates EDSL-based application code into C++ constructs that the back end runtime systems can easily implement and interact with. Given that DARMA is strictly embedded in C++, this layer makes heavy use of newer C++ motifs and features from C++11 and C++14, such as template metaprogramming, perfect forwarding, constant expressions (**`constexpr`**), and lambda capture. Many of the additions to C++ in recent years have centered around making it easier for the user to express compile-time optimizations and transformations that the compiler can make to reduce runtime overhead. As such, much of the translation DARMA does between the front end EDSL and the back end runtime system API happens at compile time, and should result in minimal runtime overhead with most modern compilers.

3.1 Separation of Responsibilities Across Layers

DARMA separates responsibilities across the three different layers: application, translation, and back end runtime system. The list below describes the most important quantities and concepts that are required for writing and running DARMA applications. Each layer will either read, write, or never use each quantity. Some of these quantities are parts of the specification while certain other quantities are introduced for illustrating concepts, and are not strictly part of the specification. Some quantities are repeated from previous sections.

- AccessHandle: a variable (templated on data type) that is used in the application as arguments to tasks and for reading/writing values in a data block. Each task has its own unique copy of AccessHandle. AccessHandles are never shared across tasks.
- Data Type: the type of a variable, e.g., **`int`**, `vector<`**`double`**`>`.
- Data Layout: the layout or internal structure of a data type, usually telling whether a type is contiguous in memory and whether a type holds only data (e.g., double) or has lookup pointers.
- Data Size: the total size a data block occupies in memory (number of bytes).
- Task Dependencies: a relationship between a task and data indicating the task depends on the data being available before the task can run.
- Task Precedence Constraints: a relationship between tasks indicating that an ordering constraint exists between tasks.
- Access Permissions: access permissions (read, read-write, etc.) for an AccessHandle within a task.
- Address: a pointer through which data is accessed. The pointer provides no information on the size or type of memory being accessed. It merely provides a means of accessing data at a particular memory location.

The following items are not strictly part of the specification, but are useful for having a rigorous vocabulary to explain and understand the translation layer. These are quantities that are *likely* to be used in a back end runtime system implementation, but are *not required*. As will be discussed in the back end section, all of these quantities (if used by a back end runtime system), exist in an abstract class `Instance` that the translation layer interacts with.

- Handle ID: the generalization of a variable in C++. A globally unique ID identifying a block of data that represents the "same" quantity across time. This corresponds to, e.g., values mesh that are updated iteratively. This is NOT synonymous with an actual physical location.
- Generation: an ID that distinguishes logically distinct generations of the same Handle ID. Updating the values in

Quantity	App	Translation Layer	Backend
Data Type	Creates	Reads	DNE
Data Layout	Modifies	Reads	DNE
Data Size	Modifies	Reads	Reads
Task Dependencies	Modifies	Reads	Reads
Task Order Constraints	DNE	Creates	Modifies
Address	Reads	DNE	Modifies
Access Permissions	Modifies	Reads	Reads
Handle ID	DNE	Opaque Create	Creates
Generation	DNE	Opaque Modify	Creates
Logical ID	DNE	Opaque Pass	Creates
Physical ID	DNE	Opaque Pass	Modifies

Table 3.1 Which concepts are modified by a given layer, which exist opaquely, and finally which concepts do not exist (DNE).

a Data Handle changes the data and therefore progresses the generation. Two data blocks with the same Handle ID that contain different logical times (usually different iterations) will be different generations.

- Data Blocks: the actual physical memory allocations where data lives. Data blocks comprise not just an address, but potentially size and location information such as whether memory is DRAM, HBM, GPU, or remote.
- Logical ID: a tuple of Generation and Handle ID. Two Data Handles with the same logical ID must access exactly the same values, but potentially different physical locations, and thus are logically the same. All objects with the same Logical ID are required to have the same Handle ID. Thus, Logical ID equivalence is a stronger condition than Handle ID equivalence.
- Physical ID: a tuple of Address, Generation, and Handle ID (although not required to be implemented as a tuple). Two Data Handles with the same Physical ID not only access exactly the same values, but must also access exactly the same memory location. Two logically distinct blocks that happen to access the same memory location at different times do NOT share a Physical ID. All data handles with the same Physical ID must have the same Logical ID and therefore the same Handle ID. Physical ID equivalence is then a stronger condition than Logical ID or Handle ID.

The way in which quantities are used in each layer of the software stack can have several possibilities:

- Modifies: The layer both reads and manipulates the given quantity.
- Creates: A subset of Modifies. The layer creates the initial version of something, but is not allowed to modify the quantity thereafter.
- Reads: The layer reads and understand a quantity, but is not allowed to manipulate it.
- Opaque Modify: The caller layer understands operations that need to be performed that will modify a struct, but the implementation details are hidden by an interface. For example, a caller can pass a forward-declared struct by pointer to a function. The function (callee) can modify integer members within the struct. Even though the caller initiates the modification, the internal details of the struct are opaque to the caller and are only known to the callee (function).
- Opaque Pass: A caller provides values in a struct to be read by a callee function, but the values are opaque to the caller function. Similar to Opaque Modify, but the callee function cannot modify the struct.
- DNE: The concept does not exist - it is neither manipulated nor read by a layer.

We now summarize where quantities are created, modified, and read in Table 3.1.

A critical part of the DARMA design avoids potential interference between layers that can read/modify the same quantity. Operations occurring in the back end must have guarantees that the application and translation layer are not creating conflicts. We must therefore define a life cycle for each task:

- Precursor: Dependencies are modified during the execution of a precursor task.
- Initialization: The task is created and initialized before being passed to a scheduler or task queue.
- Waiting: The task has been created is waiting in a queue to be scheduled or run.
- Running: The task has been popped from the queue and is actively running.
- Deletion: The task has finished running and is releasing its resources.

Quantity	Precursor	Initializing	Waiting	Running	Deletion
Data Type	App Creates	TL Reads	TL Reads	App Reads	TL Reads
Data Size	App Creates	TL,BE Reads	BE Reads	App Modifies	TL,BE Reads
Access Permissions	App Modifies	TL,BE Reads	BE Reads	n/a	TL,BE Reads
Task Dependencies	App Creates	TL,BE Reads	BE Reads	n/a	TL,BE Reads
Task Order Constraints	n/a	TL Creates	BE Modifies	n/a	BE Reads
Address	n/a	BE Modifies	BE Modifies	App Reads	BE Reads

Table 3.2 Usage of different quantities throughout a single task's life cycle

Critically, Table 3.2 ensures that each layer has no conflicts. No two layers simultaneously have modify permissions during the life cycle of a task, nor can one layer read simultaneously as another layer modifies. Two layers can simultaneously read, although even that rarely happens.

3.2 Important C++ Concepts

Even though neither the front end API application programmers nor the back end implementation developers need to understand the implementation details of the translation layer, it is useful to document several of the idioms and "tricks" used in the translation layer for those wishing to have a thorough understanding of all DARMA layers, and particularly for those who wish to contribute to the expansion and adaptation of the EDSL that is the front end API. Thus, a few of the basic techniques and concepts used by the translation layer are documented below.

3.2.1 Lambda Capture for Automatic Dependency Detection and Versioning

The most pivotal trick to document here is (semi-)automatic dependency detection through the C++ 11 lambda mechanism and the copy *capture-default* (that is, `[=]`). The C++ standard specifies that if the copy *capture-default* is given, any variables that are ODR-used[1] inside of the lambda's scope but defined outside of it[2] are copied by value into that lambda's scope. Furthermore, if a lambda with a copy *capture-default* is moved[3], the move constructors of the inner scope copies will be invoked (or, if no user-defined move constructor is given but a user-defined copy constructor is, the copy constructor is invoked). We can leverage this fact along with a thread-safe (and thread-specific) global context object to associate captured `AccessHandle<T>` objects with the capturing `create_work(...)` invocation as dependencies. Furthermore, if we make the relevant members of `AccessHandle<T>` **`mutable`**, we can modify the handle that has been *copied from* to increment the version, so that later tasks will depend on the completion of earlier `create_work` calls that capture the same handle. Thus, the capture mechanism can be used for both dependency detection and sequential semantics.

3.2.2 Keyword Arguments

The tricks used to emulate keyword arguments in C++ are well-known and have been exploited elsewhere[4] to similar effect. The addition of perfect forwarding and constant expression semantics to C++ 11 and C++ 14 allow this to be done with rigorously zero runtime overhead — all transformations used to interpret keyword arguments as traditional, positional arguments can occur at compile time.

[1]"one definition rule"-used. See `http://en.cppreference.com/w/cpp/language/definition#ODR-use` for details. As this source states, "Informally, an object is odr-used if its address is taken, or a reference is bound to it, and a function is odr-used if a function call to it is made or its address is taken."

[2]technically, defined in the lambda's "reaching scope," which is also formally defined

[3]e.g., with `std::move()`; lambdas have a deleted copy constructor

[4]e.g., the Boost::Parameter library

Chapter 4

Backend

The back end API is organized into two namespaces:

1. `darma_runtime::abstract::frontend`
2. `darma_runtime::abstract::backend`

The first contains abstraction base classes of entities that are implemented in the translation layer and are the only constructs in that layer that the back end runtime system is allowed to interact with. The second contains abstract base classes that must be concretely implemented in the back end runtime system and are the only back end abstractions the translation layer is allowed to interact with. Below is a summary of the requirements to implement these abstractions, the documentation for which is taken from the Doxygen-style comments in the source code itself. As such, the source code may be a better resource for those interested in this part of the document, but we have included it here for completeness.

4.1 Important Backend Concepts

Although some of this terminology was given in the introduction, we repeat definitions here. Some of the terms here have C++ classes that directly represent them. Other terms are only concepts, useful in illustrating the use of other C++ classes.

Task: The work unit instantiated directly by the application developer. Tasks are guaranteed to make forward progress, but are interruptible.

Execution stream: An execution stream will consist of a sequence of many tasks, and, like tasks, is guaranteed to make forward progress. All execution streams are tasks, but execution streams specifically have no parent task and are the root of an independent task-DAG. There is no class corresponding uniquely to an execution stream since all streams are tasks.

Operation: This is a unit of execution that is guaranteed to be non-interruptible. An operation is not equivalent to a task since tasks are interruptible. Operations are the smallest, schedulable units of work. A task consists of a sequence of operations. While tasks are explicitly instantiated by the application developer, operations (individual portions of task) can be implicitly instantiated by the runtime system. There is no class provided corresponding directly to an operation. Since only one component operation of a task may be active at any given time, a task always corresponds uniquely to an operation.

Handle: The DARMA generalization of a variable. Handle encapsulates both a unique immutable name (key) and an immutable type.

Logical Time: An abstract notion of time progressing as operations are performed on the values encapsulated by a Handle. There is no class corresponding to logical time. The progression of logical time for Use objects is encapsulated in the input and output Flows (see below).

Use: A Use corresponds to a Handle at a particular moment in logical time. Uses are always unique to an operation. Operations cannot add or remove uses from its context. Tasks, being interruptible, can add and remove Use instances. Uses carry particular permissions and therefore have some intent of Read, Write, or Modify.

Flow: A Flow encapsulates a data-task relationship. An input Flow indicates that a Use requires a particular value before its corresponding operation begins. An output Flow indicates that a Use produces a particular value after being released at the end of its corresponding operation. All Use objects have exactly two Flow objects — one input and one output — and each Flow is associated with exactly one Use. A Modify Use will have an input Flow indicating the value consumed and an output Flow indicating the value produced. A Read Use is will also have an output Flow even though it produces no data since the "output" indicates the release of data and clearing of an anti-dependence.

Dependency: Although Dependency is not a class in DARMA, a task will always have an initial set of Uses that must become available for the task to begin. This initial set of uses are the "dependencies" of a task.

4.2 Class Index

4.2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

4.3 Class Documentation

4.3.1 darma_runtime::abstract::backend::Flow Class Reference

A backend-allocated object representing the input/output state of a Handle at the beginning/end of a task.

`#include <flow.h>`

Detailed Description

A backend-allocated object representing the input/output state of a Handle at the beginning/end of a task.

When executing tasks, data "flows" from one task to the next. A precursor task produces data that will be consumed by a successor task. Each task carries a unique Use variable for each Handle it uses. Each Use has an input flow and output flow. This is true even of a read-only Use, with the output indicating the release of anti-dependence. An equivalence relationship between two Flows `a` and `b` is indicated by allocating the Flow `a` with a call to `Runtime↩ ::make_same_flow(b)` or vice versa. Equivalence must be defined within the backend. The translation layer will never make an equivalence test itself.

The life-cycle of a Flow consists of 4 strictly ordered phases. For some Flow instance flw,

- Creation – `&flw` is a pointer returned by any of `make_initial_flow()`, `make_fetching_flow()`, `make_null_flow()`, `make_same_flow()`, `make_forwarding_flow()`, or `make_next_flow()`

- Registration – Each flow is owned by a Use as either input or output. Each Use will be registered through `Runtime::register_use()` before being used in a task or publication. All flows have exactly one `Use` association in their lifetime; that is, `&flw` is either a return value of `Use::get_in_flow()` or `Use::get↩ _out_flow()` for some `Use` object that is an argument to `register_use()` at some time after `flw` was created but before it is released. To ensure this strict ordering of the `Flow` life-cycle, the runtime must enforce atomicity among `register_use(&u)`, `make_next_flow(&flw, ...)`/`make_forwarding_↩ flow(&flw, ...)`/`make_same_flow(&flw, ...)`, `release_use(&u)` for any `Flow` `flw` that could be returned by `u.get_in_flow()` or `u.get_out_flow()` for some `Use` `u`.
- Release – Each `Flow` is owned by a `Use` as either input or output. `Flow`s are released through to `Runtime↩ ::release_use()` on the owning `Use`. The `Flow` will never be used directly (or indirectly) by the translation after calling `release_use()`.
- At most one call to `runtime.make_next_flow(&flw, ...)` can happen anytime after creation, but before release. Any number of calls to `runtime.make_same_flow(&flw, ...)` can happen anytime after creation, but before release. At most one call to `runtime.make_forwarding_flow(&flw, ...)` can be made in the lifetime of a `Flow` (and this call does not preclude a `make_next_flow(&flw, ...)` call also being made)

Two Flow objects, `a` and `b`, are considered to consume or produce the same version of the same data if `a` was constructed using `make_same(b)` or if `b` was constructed using `make_same(a)`. The flow returned by `make_↩ same(a)`, however, is a different object and is therefore has an independent life cycle and is independently modifiable by the backend.

The documentation for this class was generated from the following file:

- flow.h

4.3.2 darma_runtime::abstract::frontend::Handle Class Reference

Encapsulates a named, mutable chunk of data which may be accessed by one or more tasks that use that data (or the privilege to schedule permissions on that data).

`#include <handle.h>`

Public Member Functions

- virtual types::key_t const & get_key () const =0

  get_key Returns a unique key. Multiple calls to this function on the same key object must always return the same value
- virtual SerializationManager const ∗ get_serialization_manager () const =0

  get_serialization_manager Returns a type-specific serialization manager. The object returned will be persistent as long as the Handle exists

Detailed Description

Encapsulates a named, mutable chunk of data which may be accessed by one or more tasks that use that data (or the privilege to schedule permissions on that data).

A Handle represents an entity conceptually similar to a variable in a serial program.

Member Function Documentation

virtual types::key_t const& darma_runtime::abstract::frontend::Handle::get_key ( ) const `[pure virtual]` get_key Returns a unique key. Multiple calls to this function on the same key object must always return the same value

Returns

A unique key identifying the tuple.

virtual SerializationManager const∗ darma_runtime::abstract::frontend::Handle::get_serialization_manager ( ) const `[pure virtual]` get_serialization_manager Returns a type-specific serialization manager. The object returned will be persistent as long as the Handle exists

Returns

A type-specific serialization manager

The documentation for this class was generated from the following file:

- handle.h

4.3.3 darma_runtime::abstract::frontend::PublicationDetails Class Reference

A class encapsulating the attributes of a particular publish operation.

`#include <publication_details.h>`

Public Member Functions

- virtual types::key_t const & get_version_name () const =0

  Get the unique version (as a key) of the item being published. The combination of h.get_key() and get_version_name() must be globally unique for a Handle `h` *returned by Use::get_handle() for the use given as the first argument to Runtime↩ ::publish_use() for which this object is the second object.*
- virtual size_t get_n_fetchers () const =0

  Get the number of unique fetches that will be performed. All N fetches must be complete before the backend can declare a publication to be finished.

Detailed Description

A class encapsulating the attributes of a particular publish operation.

Member Function Documentation

virtual types::key_t const& darma_runtime::abstract::frontend::PublicationDetails::get_version_name ( ) const `[pure virtual]` Get the unique version (as a key) of the item being published. The combination of h.get_key() and get_version_name() must be globally unique for a Handle `h` returned by Use::get_handle() for the use given as the first argument to Runtime::publish_use() for which this object is the second object.

Returns

A unique version name for the current publication of a given Handle

virtual size_t darma_runtime::abstract::frontend::PublicationDetails::get_n_fetchers ( ) const `[pure virtual]` Get the number of unique fetches that will be performed. All N fetches must be complete before the backend can declare a publication to be finished.

Returns

The number of Runtime::make_fetching_flow() calls that will fetch the combination of key and version given in the publish_use() call associated with this object

The documentation for this class was generated from the following file:

- publication_details.h

4.3.4 darma_runtime::abstract::backend::Runtime Class Reference

Abstract class implemented by the backend containing much of the runtime.

```
#include <runtime.h>
```

Public Types

- enum FlowPropagationPurpose { Input, Output, ForwardingChanges, OutputFlowOfReadOperation }

  A set of enums identifying the relationship between two flows.
- typedef frontend::Task **task_t**
- typedef enum darma_runtime::abstract::backend::Runtime::FlowPropagationPurpose flow_propagation_purpose↩_t

  A set of enums identifying the relationship between two flows.

Public Member Functions

- virtual void register_task (types::unique_ptr_template< frontend::Task > &&task)=0

  Register a task to be run at some future time by the runtime system.
- virtual frontend::Task ∗ get_running_task () const =0

  Get a pointer to the frontend::Task object currently running on the thread from which get_running_task() was invoked.
- virtual void register_use (frontend::Use ∗u)=0

  Register a frontend::Use object.
- virtual Flow ∗ make_initial_flow (frontend::Handle ∗handle)=0

  Make an initial Flow to be associated with the handle given as an argument.
- virtual Flow ∗ make_fetching_flow (frontend::Handle ∗handle, types::key_t const &version_key)=0

  Make an fetching Flow to be associated with the handle given as an argument.
- virtual Flow ∗ make_null_flow (frontend::Handle ∗handle)=0

  Make a null Flow to be associated with the handle given as an argument.
- virtual Flow ∗ make_same_flow (Flow ∗from, flow_propagation_purpose_t purpose)=0

  Make a flow that is logically identical to the input parameter.
- virtual Flow ∗ make_forwarding_flow (Flow ∗from, flow_propagation_purpose_t purpose)=0

  Make a new input Flow that receives forwarded changes from another input Flow, the latter of which is associated with a Use on which Modify immediate permissions were requested.
- virtual Flow ∗ make_next_flow (Flow ∗from, flow_propagation_purpose_t purpose)=0

  Make a flow that will be logically (not necessarily immediately) subsequent to another Flow.
- virtual void release_use (frontend::Use ∗u)=0

  Release a Use object previously registered with register_use.
- virtual void publish_use (frontend::Use ∗f, frontend::PublicationDetails ∗details)=0

  Indicate that the state of a Handle corresponding to a given Use should be accessible via a corresponding fetching usage with the same version_key.

Detailed Description

Abstract class implemented by the backend containing much of the runtime.

Note

> Thread safety of all methods in this class should be handled by the backend implementaton; two threads must be allowed to call any method in this class simultaneously.

Member Enumeration Documentation

enum darma_runtime::abstract::backend::Runtime::FlowPropagationPurpose A set of enums identifying the relationship between two flows.

Enumerator

- ***Input*** The new flow will be used as the input to another logical Use of the data
- ***Output*** The new flow will be used as the output for another logical Use of the data
- ***ForwardingChanges*** The new flow will be used as an input to another logical Use of the data that incorporates changes made to data associated with an input Flow for which Modify immediate permissions were requested, thus ”forwarding” the modifications to a new logical Use. Only ever used with make_forwarding_flow()
- ***OutputFlowOfReadOperation*** The new flow will be used as the corresponding return of get_out_flow() for a read-only Use that returns a given flow for get_in_flow().

Member Function Documentation

virtual void darma_runtime::abstract::backend::Runtime::register_task (types::unique_ptr_template< frontend::Task > && *task*) `[pure virtual]` Register a task to be run at some future time by the runtime system.

See frontend::Task for details

Parameters

task	A unique_ptr to a task object. Task is moved as rvalue reference, indicating transfer of ownership to the backend.

See also

> frontend::Task

virtual frontend::Task∗ darma_runtime::abstract::backend::Runtime::get_running_task ( ) const `[pure virtual]` Get a pointer to the frontend::Task object currently running on the thread from which get_running_task() was invoked.

Returns

> A non-owning pointer to the frontend::Task object running on the invoking thread. The returned pointer must be castable to the same concrete type as was passed to Runtime::register_task() when the task was registered.

Remarks

If the runtime implements context switching, it must ensure that the behavior of Runtime::get_running_task() is consistent and correct for a given running thread as though the switching never occurred.
The pointer returned here is guaranteed to be valid until Task::run() returns for the returned task. However, to allow context switching, it is not guaranteed to be valid in the context of any other task's run() invocation, including child tasks, and thus it should not be dereferenced in any other context.

See also

frontend::Task

virtual void darma_runtime::abstract::backend::Runtime::register_use (frontend::Use ∗ *u*) `[pure virtual]` Register a frontend::Use object.

This method registers a Use object that can be accesses through the the iterator returned by t.get_dependencies() for some task t. register_use will always be invoked before register_task for any task holding a Use `u`. Accessing a frontend::Use `u` through a frontend::Task `t` is only valid between the time `register_use(&u)` is called and `release_use(&u)` returns. No `make_*` functions may be invoked on either the input or output flows of a Use `u` returned by Use::get_input_flow() and Use::get_output_flow() before calling register_use(). Additionally, no `make_*` functions may be invoked on the input or output flows of a Use `u` after calling release_use().

virtual Flow∗ darma_runtime::abstract::backend::Runtime::make_initial_flow (frontend::Handle ∗ *handle*) `[pure virtual]` Make an initial Flow to be associated with the handle given as an argument.

The initial Flow will be used as the return value of u->get_in_flow() for the first Use∗ u registered with write privileges that returns handle for u->get_handle() (or any other handle with an equivalent return for get_key() to the one passed in here). In most cases, this will derive from calls to initial_access in the application code.

Parameters

handle	A handle encapsulating a type and unique name (variable) for which the Flow represents the initial state

virtual Flow∗ darma_runtime::abstract::backend::Runtime::make_fetching_flow (frontend::Handle ∗ *handle*, types::key_t const & *version_key*) `[pure virtual]` Make an fetching Flow to be associated with the handle given as an argument.

The fetching usage will be used as a return value of u->get_in_flow() for a Use∗ u intended to fetch the data published with a particular handle key and version_key.

Parameters

handle	A handle object carrying the key identifer returned by get_key()
version_key	A unique version for the key returned by handle->get_key()

virtual Flow∗ darma_runtime::abstract::backend::Runtime::make_null_flow (frontend::Handle ∗ *handle*) `[pure virtual]` Make a null Flow to be associated with the handle given as an argument.

A null usage as a return value of u->get_out_flow() for some Use∗ u is intended to indicate that the data associated with that Use has no subsequent consumers and can safely be deleted. See release_use().

Parameters

handle	The handle variable associate with the flow

virtual Flow∗ darma_runtime::abstract::backend::Runtime::make_same_flow (Flow ∗*from,* flow_propagation↩ _purpose_t *purpose*) `[pure virtual]` Make a flow that is logically identical to the input parameter.

Calls to make_same_flow() indicate a logical identity between Flows in different Use instances. make_same_flow() may not return the original pointer passed in. Flow objects must be unique to a Use. Flows are registered and released indirectly through calls to register_use()/release_use(). The input Flow to make_same_flow() must have been registered through a register_use() call, but not yet released through a release_use() call. There is no restriction on the number of times make_same_flow() can be called with a given input.

Parameters

from	An already initialized flow returned from `make_*_flow`
purpose	An enum indicating the relationship between logically identical flows (purpose of the function). For example, this indicates whether the two flows are both inputs to different tasks or whether the new flow is the sequential continuation of a previous write (forwarding changes)
A	new Flow object that is equivalent to the input flow

virtual Flow∗ darma_runtime::abstract::backend::Runtime::make_forwarding_flow (Flow ∗ *from,* flow_↩ propagation_purpose_t *purpose*) `[pure virtual]` Make a new input Flow that receives forwarded changes from another input Flow, the latter of which is associated with a Use on which Modify immediate permissions were requested.

Parameters

from	An already initialized flow returned from make_∗_flow
purpose	An enum indicating the relationship between logically identical flows (purpose of the function). In the current specification, this enum will always be ForwardingChanges

virtual Flow∗ darma_runtime::abstract::backend::Runtime::make_next_flow (Flow ∗*from,* flow_propagation↩ _purpose_t *purpose*) `[pure virtual]` Make a flow that will be logically (not necessarily immediately) subsequent to another Flow.

Calls to make_next_flow() indicate a producer-consumer relationship between Flows. make_next_flow() indicates that an operation consumes Flow∗ from and produces the returned Flow∗. Flows are registered and released indirectly through calls to register_use()/release_use(). Flow instances cannot be shared across Use instances. The input to make_next_flow() must have been registered with register_use(), but not yet released through release_use().

Parameters

from	The flow consumed by an operation to produce the Flow returned by make_next_flow()
purpose	An enum indicating the purpose of the next flow

Returns

A new Flow object indicating that new data will be produced by the data incoming from the Flow given as a parameter

virtual void darma_runtime::abstract::backend::Runtime::release_use (frontend::Use ∗ *u*) `[pure virtual]` Release a Use object previously registered with register_use.

Upon release, if the Use∗ u has immediate_permissions() of at least Write, the release allows the runtime to match the producer flow to pending Use instances where u->get_out_flow() is equivalent to the consumer pending->get_in_←flow() (with equivalence for Flow defined in flow.h). The location provided by u->get_data_pointer_reference() holds the data that satisfies the pending->get_in_flow()

If the return value of u->get_out_flow() is the same as or aliases a Flow created with make_null_flow() at the time release_use() is invoked, the data at this location may be safely deleted.

If the Use∗ u has scheduling_permissions() of at least Write, but has no immediate permissions the Use∗ is an ”alias” use. As such, u->get_out_flow() only provides an alias for u->get_in_flow(). u->get_in_flow() is the actual producer flow that satisfies all tasks/uses dependeing on u->get_out_flow(). There will be some other task t2 with Use∗ u2 such that u2->get_out_flow() and u->get_in_flow() are equivalent. release_use(u2) may have already been called, may be in process, or may not have been called when release_use(u) is invoked. The backend runtime is responsible for ensuring correct satisfaction of pending flows and thread safety (atomicity) of release_use(...) with aliases. An alias use can correspond to another alias use, creating a chain of aliases that the backend runtime must resolve.

Alias resolution should be implemented in constant time. That is, if Flow a aliases b and Flow b aliases c, the fact that a aliases c should be discernible without linear cost in the size of the set {a, b, c}.

If the Use∗ u has immediate_permissions() of Read, the release allows the runtime to clear anti-dependencies. For a task t2 with Write privileges on Use∗ u2 such that u2->get_in_flow() is equivalent to u->get_in_flow() (or u->get_←out_flow(), depending on backend implementation) If u is the last use (there are no other Use∗ objects registered with u->get_in_flow() equivalent to u2->get_in_flow()) then task t2 has its preconditions on u2 satisfied.

Parameters

u	The Use being released, which consequently releases an in and out flow with particular permissions.

virtual void darma_runtime::abstract::backend::Runtime::publish_use (frontend::Use ∗ *f*, frontend::Publication←Details ∗ *details*) `[pure virtual]` Indicate that the state of a Handle corresponding to a given Use should be accessible via a corresponding fetching usage with the same version_key.

See PublicationDetails for more information

Parameters

u	The particular use being published
details	This encapsulates at least a version_key and an n_readers

See also

PublicationDetails

The documentation for this class was generated from the following file:

- runtime.h

4.3.5 darma_runtime::abstract::frontend::SerializationManager Class Reference

An immutable object allowing the backend to query various serialization sizes, offsets, behaviors, and data, for a given handle and its associated data block.

`#include <serialization_manager.h>`

Public Member Functions

- virtual size_t get_metadata_size () const =0

  returns the size of the data as a contiguous C++ object in memory (i.e., sizeof(T))
- virtual size_t get_packed_data_size (const void ∗const object_data) const =0

  Get the size of the buffer that the pack_data() function needs for serialization.
- virtual void pack_data (const void ∗const object_data, void ∗const serialization_buffer) const =0

  Packs the object data into the serialization buffer.
- virtual void unpack_data (void ∗const object_dest, const void ∗const serialized_data) const =0

  Unpacks the object data from the serialization buffer into object_dest.

Detailed Description

An immutable object allowing the backend to query various serialization sizes, offsets, behaviors, and data, for a given handle and its associated data block.

Remarks

The only method that is valid to invoke for the 0.2.0 spec implementation is get_metadata_size()

Member Function Documentation

virtual size_t darma_runtime::abstract::frontend::SerializationManager::get_packed_data_size (const void ∗const *object_data*) const `[pure virtual]` Get the size of the buffer that the pack_data() function needs for serialization.

Parameters

object_data	pointer to the start of the C++ object to be serialized. The object must be fully constructed and valid for use in any context where it could be used when unpacked (”could be used” is a user-defined concept here, but basically means that operations performed on the object must yield results and side-effects ”as-if” the serialization had never happened).

virtual void darma_runtime::abstract::frontend::SerializationManager::pack_data (const void ∗const *object_-data,* void ∗const *serialization_buffer*) const `[pure virtual]` Packs the object data into the serialization buffer.

Parameters

object_data	pointer to the start of the C++ object to be serialized. Must be in the exact same state as when get_packed_data_size() was invoked with the same object.
serialization_buffer	the buffer into which the data should be packed. The backend must preallocate this buffer to be the size returned by get_packed_data_size() when invoked *immediately* prior to pack_data() with the same object_data pointer

Remarks

> The backend must ensure that no running task has write access to the object_data between the time get_packed_-data_size() is called and pack_data() returns, such that the state of object_data does not change in this time frame (under, of course, the allowed assumptions that the user has correctly specified aliasing characteristics of the handle or handles pointing to object_data).

virtual void darma_runtime::abstract::frontend::SerializationManager::unpack_data (void ∗const *object_dest,* const void ∗const *serialized_data*) const `[pure virtual]` Unpacks the object data from the serialization buffer into object_dest.

Upon invocation, object_dest must be allocated (by the backend) to have size get_metadata_size(), but the unpack_↩data() method is responsible for construction of the object itself into this buffer. Upon return, object_dest should point to the beginning of a C++ object that is fully constructed and valid for use in any context where it could have been used before it was packed (see get_packed_data_size() for clarification of "could have been used")

Parameters

object_dest	backend-allocated buffer of size get_metadata_size() into which the object should be constructed and deserialized
serialized_data	a pointer to the beginning of a buffer of the same size and state as the second argument to pack_data() upon return of pack_data() for the corresponding object to be unpacked.

The documentation for this class was generated from the following file:

- serialization_manager.h

4.3.6 darma_runtime::abstract::frontend::Task Class Reference

A piece of work that acts on (accesses) zero or more Handle objects at a particular point in the apparently sequential uses of these Handle objects.

```
#include <task.h>
```

Public Member Functions

- virtual types::handle_container_template< Use const ∗ > const & get_dependencies () const =0

  Return an Iterable of Use objects whose permission requests must be satisfied before the task can run.
- virtual void run ()=0

  Invoked by the backend to start the execution phase of the task's life cycle.
- virtual const types::key_t & get_name () const =0

  returns the name of the task if one has been assigned with set_name(), or a reference to a default-constructed Key if not.
- virtual void set_name (const types::key_t &name_key)=0

  sets the unique name of the task
- virtual bool is_migratable () const =0

  returns true iff the task can be migrated
- virtual size_t get_packed_size () const =0

  Returns the number of bytes required to store the task object. Not relevant for current specification which does not support task migration.
- virtual void pack (void ∗allocated) const =0

  Pack a migratable serialization of the task object into the passed-in buffer.

Detailed Description

A piece of work that acts on (accesses) zero or more Handle objects at a particular point in the apparently sequential uses of these Handle objects.

Life-cycle of a Task, for some Task instance t:

- registration – register_task() is called by moving a unique_ptr to t into the first argument. At registration time, all of the Use objects returned by the dereference of the iterator to the iterable returned by t.get_dependencies() must be registered and must not be released at least until the backend invokes t.run() method.
- execution – the backend calls t.run() once all of the dependent Uses have their required permissions to their data. By this point (and not necessarily sooner), the backend must have assigned the return of get_data_pointer_-reference() to the beginning of the actual data for any Use dependencies requiring immediate permissions.
- release – when Task.run() returns, the task is ready to be released. The backend may do this by deleting or resetting the unique_ptr passed to it during registration, which will in turn trigger the ∼Task() virtual method invocation. At this point (in the task destructor), the frontend is responsible for calling release_handle_access() on any Use instances requested by the task and not explicitly released in the task body by the user.

Member Function Documentation

virtual types::handle_container_template<Use const∗> const& darma_runtime::abstract::frontend::Task::get↩ _dependencies ( ) const `[pure virtual]` Return an Iterable of Use objects whose permission requests must be satisfied before the task can run.

See description in Task and Use life cycle discussions.

Returns

An iterable container of Use objects whose availability are preconditions for task execution

virtual const types::key_t& darma_runtime::abstract::frontend::Task::get_name ( ) const `[pure virtual]` returns the name of the task if one has been assigned with set_name(), or a reference to a default-constructed Key if not.

In the current spec this is only used with the outermost task, which is named with a key of two size_t values: the SPMD rank and the SPMD size. See darma_backend_initialize() for more information

Returns

A key object giving a unique name to the task

virtual void darma_runtime::abstract::frontend::Task::set_name (const types::key_t & *name_key*) `[pure virtual]` sets the unique name of the task

In the current spec this is only used with the outermost task, which is named with a key of two size_t values: the SPMD rank and the SPMD size. See darma_backend_initialize() for more information

Parameters

name_key	A key object containing a unique name for the task

virtual bool darma_runtime::abstract::frontend::Task::is_migratable ( ) const `[pure virtual]` returns true iff the task can be migrated

Remarks

always return false in the current spec implementation. Later specs will need additional hooks for migration

Returns

Whether the task is migratable.

virtual size_t darma_runtime::abstract::frontend::Task::get_packed_size ( ) const `[pure virtual]` Returns the number of bytes required to store the task object. Not relevant for current specification which does not support task migration.

Returns

The size in bytes need to pack the task into a serialization buffer

virtual void darma_runtime::abstract::frontend::Task::pack (void ∗ *allocated*) const `[pure virtual]` Pack a migratable serialization of the task object into the passed-in buffer.

Parameters

allocated	The pointer to region of memory guaranteed to be large enough to hold the serialization of the class

The documentation for this class was generated from the following file:

- task.h

4.3.7 darma_runtime::abstract::frontend::Use Class Reference

Encapsulates the state, permissions, and data reference for a given use of a Handle at a given time.

```
#include <use.h>
```

Public Types

- enum Permissions {
  None =0, Read =1, Write =2, **Modify** =3,
  Reduce =4 }

  An enumeration of the allowed values that immediate_permissions() and scheduling_permissions() can return.
- typedef enum darma_runtime::abstract::frontend::Use::Permissions permissions_t

  An enumeration of the allowed values that immediate_permissions() and scheduling_permissions() can return.

Public Member Functions

- virtual Handle const ∗ get_handle () const =0

  Return a pointer to the handle that this object encapsulates a use of.
- virtual backend::Flow ∗ get_in_flow ()=0

  Get the Flow that must be ready for use as a precondition for the Task t that depends on this Use.
- virtual backend::Flow ∗ get_out_flow ()=0

  Get the Flow that is produced or made available when this Use is released.

- virtual permissions_t immediate_permissions () const =0
- virtual permissions_t scheduling_permissions () const =0
- virtual void ∗& get_data_pointer_reference ()=0

Detailed Description

Encapsulates the state, permissions, and data reference for a given use of a Handle at a given time.

Use objects have a life cycle with 3 strictly ordered phases. For some Use instance u,

- Creation/registration – `&u` is passed as the argument to register_use(). At this time, u.get_in_flow() and u.get_↩ out_flow() must return unique, valid Flow objects.
- Task or Publish use (up to once in lifetime):
  - Task use: For tasks, `&u` can be accessed through the iterable returned by t.get_dependencies() for some Task object tpassed to register_task() after `u` is created and before `u` is released. At this time, u.immediate↩ _permissions(), u.scheduling_permissions(), and u.get_data_pointer_reference() must return valid values, and these values must remain valid until Runtime::release_use(u) is called (note that migration may change this time frame in future versions of the spec).
  - Publish use: A single call to Runtime::publish_use() may be made for any Use. The frontend may immediately call release_use() after publish_use(). If the publish is deferred and has not completed by the time release_use() is called, the backend runtime must extract the necessary Flow and key fields from the Use.
- Release – Following a task use or a publish use, the translation layer will make a single call to Runtime::release_-use. The Use instance may no longer be valid on return. The destructor of Use will NOT delete its input and output flow. The backend runtime is responsible for deleting Flow allocations, which may occur during release.

Member Enumeration Documentation

enum darma_runtime::abstract::frontend::Use::Permissions An enumeration of the allowed values that immediate↩ _permissions() and scheduling_permissions() can return.

Enumerator

None A Use may not perform any operations (read or write). Usually only immediate_permissions will be None

Read An immediate (scheduling) Use may only perform read operations (create read-only tasks)

Write An immediate (scheduling) Use may perform write operations (create write tasks)

Reduce An immediate (scheduling) Use may perform reduce operations (create reduce tasks). This is not a strict subset of Read/Write privileges

Member Function Documentation

virtual permissions_t darma_runtime::abstract::frontend::Use::immediate_permissions ( ) const `[pure virtual]` Get the immediate permissions needed for the Flow returned by get_in_flow() to be ready as a precondition for this Use

virtual permissions_t darma_runtime::abstract::frontend::Use::scheduling_permissions ( ) const `[pure virtual]` Get the scheduling permissions needed for the Flow returned by get_in_flow() to be ready as a precondition for this Use

virtual void∗& darma_runtime::abstract::frontend::Use::get_data_pointer_reference ( ) `[pure virtual]`
Get a reference to the data pointer on which the requested immediate permissions have been granted.

For a Use requesting immediate permissions, the runtime will set the value of the reference returned by this function to the beginning of the data requested at least by the time the backend calls Task::run() on the task requesting this Use

The documentation for this class was generated from the following file:

- use.h

Chapter 5

Requirements

5.1 High-level Philosophy

The front end API requirements are informed by a few high-level design principles:

- Keep simple things simple
- Keep tractable things tractable
- Make difficult things tractable
- New programming models should not complicate reasoning about code correctness
- New programming models should simplify application-specific performance optimizations
- Pareto rule: 80% of the compute benefit from modest human effort preferred over 100% of compute benefit from massive human effort

Essentially, code written in the DARMA programming model should be not be more difficult than existing programming models. Additionally, the DARMA programming model should not pass off 100% of the responsibility for high-performance to the runtime/compilers. Rather, DARMA should enable application developers to express performance improvements in ways not previously possible.

Our approach is informed by what we see as the "axiomatic" challenges facing high-performance computing:

- SPMD (data parallelism) will remain the dominant parallelism and primary structure of application codes
- New architectures will have too much compute capacity for basic data parallelism to fill
- Task parallelism and pipeline parallelism will help "fill" the compute capacity on machines
- The traditional abstract machine model (flat memory spaces, uniform compute elements) will get further from actual system architecture as accelerators and deep memory hierarchies become more commonplace
- Applications with dynamic load balance or dynamic sparsity will require composable, migratable chunks of work

5.2 Application Requirements for the front end API

Based on co-design efforts with application and runtime system development teams, the following DARMA front end API requirments have been identified:

- The DARMA front end API must enable the development and deployment of SPMD algorithms in an intuitive and simple way.
- The DARMA front end API must not limit the ability of hte application developer to use their own data structures.
- The DARMA front end API must support collective communication operations.
- The DARMA front end API must not limit the ability of the application developer to express and control the initial problem decomposition.
- The DARMA front end API must not limit the application developer's ability to mix and express all forms of parallelism.

5.3 Back end runtime system requirements

Althouth a primary purpose of the DARMA specification is to provide a back end runtime system specification that is relatively execution model agnostic, we will synthesize our application and runtime-system co-design activities into a list of back end runtime system requirements. To date, the following requirements have been identified:

- A DARMA-compliant runtime system must support an efficient SPMD launch of an application code.
- A DARMA-compliant runtime system must not limit the ability of the application developer to use their own data structures.
- A DARMA-compliant runtime system must efficiently implement distributed, key-value-style coordination between multiple streams of execution.

5.4 Co-design contributors

In addition to the authors listed on this document, the API is being co-designed and vetted with application developers and computer scientists whose knowledge spans the entire runtime software stack.

Applications affecting the design and requirements:

- Sandia ASC Advanced Technology Development and Mitigation (ATDM) electromagnetic plasma code (POC: Matt Bettencourt)
- Sandia ASC ATDM reentry code (POCs: Micah Howard, Steve Bova)
- Trilinos Phalanx package for finite element matrix assembly (POC: Roger Pawlowski)
- Uncertainty quantification driver (POCs: Eric Phipps, Francesco Rizzi)
- Domain decomposition preconditioners for linear solvers (POCs: Ray Tuminaro, Clark Dohrman)

Computer Science Research Efforts

- Kokkos (POCs: Carter Edwards, Christian Trott)
- Data Management (POCs: Craig Ulmer, Gary Templet)
- Low-level operating systems requirements (POCs: Stephen Olivier, Ron Brightwell)

Chapter 6

Evolution of the Specification

6.1 Specification History

Version 0.1 of the specification existed in API form only, and the documention of that version of the specification differs substantially enough from the current one that it is not included in this work. In version 0.1 of the specification:

1. all input and output dependencies had to be explicitly enumerated by the application developer,
2. data was passed to all tasks (even inline tasks) via function parameters,
3. all inputs and outputs to each task were declared using coordination semantics,
4. explicit versioning of inputs/outputs was required to keep data logically distinct, and
5. sequential ordering of statements within DARMA had no significance for task ordering.

Application developer concerns regarding version 0.1 of the specification centered around the 1) verbosity of the approach, 2) the difficulty of reasoning about correct program order of tasks, and 3) the fact that `create_work` functioned poorly in the contexts of hierarchical data structures and dependencies, like classes with members that were also classes. The first two of these issues are addressed in version 0.3.0-alpha of the specification, and the third concern will be addressed in later releases of the specification.

6.2 New Features in 0.3.0-alpha

In version 0.3.0-alpha of the specification we:

1. leverage the C++ capture mechanism to minimize verbosity of the front end API,
2. introduce a functor interface that is more feature rich than the lambda interface,
3. provide sequential semantics within an execution stream to facilitate reasoning about program order,
4. introduce the use of handle variables to access data in the key-value store to limit number of key-value operations for often-used data,
5. require explicit publication of all data to the key-value store for data shared between execution streams.

6.3 Planned Features in Future Releases

As part of the co-design process, this specification will evolve quickly. Based on feedback thus far, there are already many additional features planned for future incarnations of the specification that will be released this calendar year (2016). These are summarized below:

0.3.1:
- Hierarchical dependencies (e.g., classes that have dependencies as member variables) and containment and aliasing management
- Task creation within class member functions
- Support for collectives

0.4:
- Schedule-only handles for ”branch” tasks that create many other tasks, but do not read data
- Include support for expression of `execution space` and `memory space` and assignment of work among these abstract machine model concepts
- Custom data models supporting arbitrary data slicing/interference tests

- Data staging hooks to accompany custom slicing
- MPI interoperability, allowing DARMA to run within MPI programs

0.5:
- Leaf task optimizations for tasks that create no subtasks
- Load balancing hooks and hints to expose existing backend load balancing algorithms and hints to the user
- Serialization of polymorphic classes
- MPI interoperability, allowing MPI to run within DARMA programs

0.6:
- Distributed containers (vectors and maps distributed across execution streams)
- Serialization of polymorphic classes
- `read_write_access` fetching of published data

Appendix A

Examples

A.1 Basic functionalities for lambda interface

A.1.1 DARMA environment

Example showing how to initialize and finalize the DARMA environment.

```cpp
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;

  std::cout << "Initializing darma" << std::endl;
  darma_init(argc, argv);

  // empty, don't do anything

  std::cout << "Finalizing darma" << std::endl;
  darma_finalize();
  return 0;
}
```

A.1.2 DARMA rank and size

Example showing DARMA rank and size.

```cpp
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;
  darma_init(argc, argv);

  // get my rank
  const size_t myRank = darma_spmd_rank();
  // get size
  const size_t size = darma_spmd_size();

  std::cout << "Rank " << myRank << "/" << size << std::endl;

  darma_finalize();
  return 0;
}
```

A.1.3 Deferred work creation

Example showing a very simple `create_work` with no dependencies.

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;
  darma_init(argc, argv);
  const size_t myRank = darma_spmd_rank();
  const size_t size = darma_spmd_size();

  create_work([=]
  {
    std::cout << "CW: Rank " << myRank << "/" << size << std::endl;
  });

  darma_finalize();
  return 0;
}
```

A.1.4 Creating handles 1

Example showing the use of `initial_access`.

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;
  darma_init(argc, argv);
  const size_t myRank = darma_spmd_rank();
  const size_t size = darma_spmd_size();

  // this just creates different handles for different types
  // NOTE: data does not exist yet, only handles!
  auto my_handle1 = initial_access<double>("data_key_1", myRank);
  auto my_handle2 = initial_access<int>("data_key_2", myRank);
  auto my_handle3 = initial_access<std::string>("data_key_3", myRank);
  // etc...

  darma_finalize();
  return 0;
}
```

A.1.5 Creating handles 2

Another example on `initial_access` handle and its use.

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;

  darma_init(argc, argv);
  const size_t myRank = darma_spmd_rank();
  const size_t size = darma_spmd_size();

  // this just creates different handles for different types
  // NOTE: data does not exist yet, only handles!
  auto handle1 = initial_access<double>("data_key_1", myRank);
  auto handle2 = initial_access<std::string>("data_key_3", myRank);

  create_work([=]
  {
    // first, constructs data with default constructor
    handle1.emplace_value(3.3);
    handle2.emplace_value("Sky is blue");

    // get current values pointed to by the handles
    auto h1Val = handle1.get_value();
    auto h2Val = handle2.get_value();
    std::cout << "After construction: h1Value=" << h1Val << std::endl;
    std::cout << "After construction: h2Value=" << h2Val << std::endl;

    // reset values using set value function
    handle1.set_value(6.6);
    handle2.set_value("Sky is green");
    std::cout << "After reset: h1Value=" << handle1.get_value() << std::endl;
    std::cout << "After reset: h2Value=" << handle2.get_value() << std::endl;

    // reset values using reference
    auto & h1r = handle1.get_reference();
    auto & h2r = handle2.get_reference();
    h1r = 9.9;
    h2r = "Sky is yellow";
    std::cout << "After reset: h1Value=" << handle1.get_value() << std::endl;
    std::cout << "After reset: h2Value=" << handle2.get_value() << std::endl;
  });

  darma_finalize();
  return 0;
}
```

A.1.6 Arrow operator for handles

Example showing the arrow operator on an handle.

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;
  darma_init(argc, argv);
  const size_t myRank = darma_spmd_rank();
  const size_t size = darma_spmd_size();

  // create handle to data
  auto my_handle1 = initial_access<std::vector<double>>("data", myRank);
  create_work([=]
  {
    // first, constructs data with default constructor
    my_handle1.emplace_value(0.0); // set to zero
    // operator-> : get access to methods of object pointed to by handle
    my_handle1->resize(4);

    // get the data and set values
    double * vecPtr = my_handle1->data();
    for (int i = 0; i < 4; ++i){
      vecPtr[i] = (double) i + 0.4;
    }

    // get the last element and check its value
    std::cout << my_handle1->back() << std::endl;
    if (my_handle1->back() != 3.4){
      std::cerr << "Error: handle value != 3.4!" << std::endl;
      std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
      exit( EXIT_FAILURE );
    }
  });

  darma_finalize();
  return 0;
}
```

A.1.7 Deferred work and constraining privileges

Example showing how to issue a `create_work` and constraining privileges on a handle to be read-only.

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;
  darma_init(argc, argv);
  const size_t myRank = darma_spmd_rank();
  const size_t size = darma_spmd_size();

  // handle to data
  auto my_handle = initial_access<double>("data", myRank);
  create_work([=]{
    my_handle.emplace_value(0.55);
  });

  // downgrade my_handle to read_only inside following create_work
  create_work(reads(my_handle),[=]{
    std::cout << " " << my_handle.get_value() << std::endl;
  });

  darma_finalize();
  return 0;
}
```

A.2 Hello World

Example for one possible implementation of “hello world”. There are three main parts involved:

1. the DARMA environment is initialized,
2. each rank issues a task to store a greeting message into a string, and
3. each rank then creates a task to printing to standard output the message and its rank.

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;

  darma_init(argc, argv);
  size_t me = darma_spmd_rank();
  size_t n_ranks = darma_spmd_size();

  // create handle to string variable
  auto greeting = initial_access<std::string>("myName", me);
  // set the value
  create_work([=]{
    greeting.set_value("hello world!");
  });

  // print the value
  create_work([=]{
    std::cout << "DARMA rank " << me
              << " says: " << greeting.get_value() << std::endl;
  });

  darma_finalize();
  return 0;
}
```

A.3 Key-Value Example

This example is to illustrate simple transactions with the key-value store, but in a distributed setting. We will ask each rank to publish a float to be read by two readers, a rank on the left and one on the right. Then we will ask each rank to get two floats, those published by the left and right neighbors and print to screen. We use periodic logic for neighbors.

```
#include <darma.h>
using namespace darma_runtime;
using namespace darma_runtime::keyword_arguments_for_publication;
int darma_main(int argc, char** argv)
{
  darma_init(argc, argv);
  size_t me = darma_spmd_rank();
  size_t n_ranks = darma_spmd_size();

  // define neighbors with periodic arrangement
  size_t left_nbr = (me == 0) ? n_ranks-1 : me-1 ;
  size_t right_nbr = (me == n_ranks-1) ? 0 : me+1 ;

  auto float_to_pub = initial_access<float>("floatKey", me);
  create_work([=]
  {
    //set_value could be replaced by the more verbose
    //->allocate, followed by, ->get() = value
    float_to_pub.set_value(2692.0 + me); //a float I like
  });

  float_to_pub.publish(n_readers=2);
  //n_readers=2: two read_access handles will be defined for this

  // fetch the data
  auto float_from_left = read_access<float>("floatKey", left_nbr);
  auto float_from_right= read_access<float>("floatKey", right_nbr);
  create_work([=]
  {
    std::cout << "My rank is " << me
              << " values from my left/right are "
              << float_from_left.get_value() << " "
              << float_from_right.get_value() << std::endl;
  });

  darma_finalize();
  return 0;
}
```

A.3.1 Publishing and read access

This example explains in more detail the use of `publish` and `read_access`. The example involves two DARMA ranks, each creating data, publishing it, and then fetching the other rank's data.

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;
  using namespace darma_runtime::keyword_arguments_for_publication;

  darma_init(argc, argv);

  const size_t myRank = darma_spmd_rank();
  const size_t size = darma_spmd_size();

  // only run with 2 ranks
  if (size!=2){
    std::cerr << "# of ranks != 2, not supported!" << std::endl;
    std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
    exit( EXIT_FAILURE );
  }

  // rank0 reads from source = rank1
  // rank1 reads from source = rank0
  size_t source = myRank==0 ? 1 : 0;

  auto my_handle = initial_access<double>("data", myRank);

  create_work([=]
  {
    my_handle.emplace_value(0.5 + (double) myRank);

    // n_readers == 1 because:
    //  rank0 reads data of rank1
    //  rank1 reads data of rank0
    my_handle.publish(n_readers=1);
  });

  AccessHandle<double> readHandle = read_access<double>("data", source);
  create_work([=]
  {
    std::cout << myRank << " " << readHandle.get_value() << std::endl;
    if (myRank==0){
      if (readHandle.get_value() != 1.5){
        std::cerr << "readHandle.get_value() != 1.5" << std::endl;
        std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
        exit( EXIT_FAILURE );
      }
    }
    else
    {
      if (readHandle.get_value() != 0.5){
        std::cerr << "readHandle.get_value() != 1.5" << std::endl;
        std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
        exit( EXIT_FAILURE );
      }
    }
  });

  darma_finalize();

  return 0;
}
```

A.4 Publishing, versioning and lifetime of handles

Lifetime of handles is tricky, particularly for `read_access` type handles. In the following example, we initialize data, publish it, fetch it from another rank, modify the data, publish it again under a new version, and then fetch the new version from another rank. The `create_work` on lines $64 - 68$ can't execute until the back end knows the first fetched version is no longer in use. We put an extra set of $\{\ \}$ around the code in lines $40 - 61$ to tell the back end that the `readHandle` is no longer needed and can go out-of-scope and the fetching is done.

Without the scoping $\{\}$, the code would deadlock. `darma_finalize()` cannot return until after *all* the `create_works` have completed. However, without the additional scoping, the backend would not know that the first fetched version is no longer needed until `darma_main()` returns, which requires `darma_finalize()` to have already returned.

While scoping is necessary in this case, there will be other cases where it only helps to improve efficiency and concurrency in the scheduling and execution of tasks. Scoping is a good programming practice.

```
#include <darma.h>
int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;
  using namespace darma_runtime::keyword_arguments_for_publication;

  darma_init(argc, argv);
  const size_t myRank = darma_spmd_rank();
  const size_t size = darma_spmd_size();

  // only run with 2 ranks
  if (size!=2)
  {
    std::cerr << "# of ranks != 2, not supported!" << std::endl;
    std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
    exit( EXIT_FAILURE );
  }

  // rank0 reads from source = rank1
  // rank1 reads from source = rank0
  size_t source = myRank==0 ? 1 : 0;

  // create data
  auto my_handle = initial_access<double>("data", myRank);
  create_work([=]
  {
    my_handle.emplace_value(0.5 + (double) myRank);

    // n_readers == 1 because:
    //  rank0 reads data of rank1
    //  rank1 reads data of rank0
    my_handle.publish(n_readers=1,version=0);
  });

  // first time reading
  /* scopinh below {} is needed because it tells the backend that readHandle
     will go outofscope and so backend has more detailed info.
     Scoping is a good practice and in this case is needed to avoid deadlock.
  */
  {
    auto readHandle = read_access<double>("data", source,version=0);
    create_work([=]
    {
      std::cout << myRank << " " << readHandle.get_value() << std::endl;
      if (myRank==0){
        if (readHandle.get_value() != 1.5){
          std::cerr << "readHandle.get_value() != 1.5" << std::endl;
          std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
          exit( EXIT_FAILURE );
        }
      }
      else
      {
        if (readHandle.get_value() != 0.5){
          std::cerr << "readHandle.get_value() != 0.5" << std::endl;
          std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
          exit( EXIT_FAILURE );
        }
      }
    });
  }

  // reset value and update version
  create_work([=]
  {
    my_handle.set_value(2.5 + (double) myRank);
    my_handle.publish(n_readers=1,version=1);
  });
  // second time reading
  auto readHandle2 = read_access<double>("data", source,version=1);
  create_work([=]
  {
    std::cout << myRank << " " << readHandle2.get_value() << std::endl;
    if (myRank==0){
      if (readHandle2.get_value() != 3.5){
        std::cerr << "readHandle2.get_value() != 3.5" << std::endl;
        std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
        exit( EXIT_FAILURE );
      }
    }
    else
    {
      if (readHandle2.get_value() != 2.5){
        std::cerr << "readHandle2.get_value() != 2.5" << std::endl;
        std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
        exit( EXIT_FAILURE );
      }
    }
  });

  darma_finalize();
  return 0;
}
```

A.5 1D Poisson Equation

Boundary value problem:

$$\frac{\partial^2 u(x)}{\partial x^2} = f(x) \quad \text{in } \Omega = (0,1), \text{ with } u(0) = 0, \; u(1) = \exp(1)\sin(1) \tag{A.1}$$

where $f(x) = 2\exp(x)\sin(x)$. This problem is chosen because it has an exact solution, namely $u_{exact} = \exp(1)\sin(1)$. The exact solution will be used for checking the correctness of the code.

Discretize the domain with N equally spaced points such that

$$u_i \approx u(x_i), \quad f_i = f(x_i) \quad x_i = i\Delta x, \; \Delta x = \frac{1}{N-1}, \; i = 0, 1, ..., N-1 \tag{A.2}$$

Use central difference approximation for the second derivative for all interior points:

$$\begin{cases} \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} = f_i, & \text{for } i = 1, ..., N-2 \\ u_0 = 0, \; u_{N-1} = \exp{(1)} * \sin{(1)} & \text{Dirichlet BC} \end{cases} \tag{A.3}$$

This translates to a linear system of equations $Au = f$ where A is an $N-2 \times N-2$ tridiagonal matrix

$$A = \begin{vmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -2 \end{vmatrix}. \tag{A.4}$$

and u is the unknown, and f is the right-hand-side. Both u and f have size $N-2$. Solving this linear system yields the solution at all the inner points of the domain. For demonstration purposes, we solve this system using Thomas algorithm, a method well-suited for tridiagonal systems. The solver needs these vectors:

1. a: contains all the sub-diagonal entries.
2. b: contains all the diagonal entries.
3. c: contains all the upper-diagonal entries.
4. d: contains all the right-hand-side entries. Also, our current version of the solver is such that on exit, the vector d contains the solution.

How do we implement this in DARMA? For demonstration purposes, we limit our attention to the case of a single rank. More complex examples involving multiple ranks will be shown later.

There are three main steps involved, namely initialization, solution of the linear system, and error checking. The DARMA main file is as follows:

```
#include "../common_poisson1d.h"
#include "../constants.h"
#include <darma.h>
using namespace darma_runtime; //here because headers below need this too
#include "initialize.h"
#include "solveTridiag.h"
#include "checkError.h"

int darma_main(int argc, char** argv)
{
  darma_init(argc, argv);
  size_t me = darma_spmd_rank();
  size_t n_spmd = darma_spmd_size();

  // supposed to be run with 1 rank
  if (n_spmd>1){
    std::cerr << "# of ranks != 1, not supported!" << std::endl;
    std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
    exit( EXIT_FAILURE );
  }

  typedef std::vector<double> vecDbl;
  // handles for data needed for matrix
  auto subD = initial_access<vecDbl>("a",me); // subdiagonal
  auto diag = initial_access<vecDbl>("b",me); // diagonal
  auto supD = initial_access<vecDbl>("c",me); // superdiagonal
  auto rhs  = initial_access<vecDbl>("d",me); // rhs and solution

  // initialize the handles
  initialize(subD, diag, supD, rhs);

  // solve tridiagonal system
  solveTridiagonalSystem(subD, diag, supD, rhs);

  // check solution L1 error
  checkFinalL1Error(rhs);

  darma_finalize();
  return 0;
}
```

The header file constants.h contains:

```
#ifndef EXAMPLES_POISSON1D_CONSTS_H_
#define EXAMPLES_POISSON1D_CONSTS_H_

constexpr int nx = 51;       // grid points
constexpr int nInn = nx-2;   // inner grid points
constexpr double xL = 0.0;   // left boundary
constexpr double xR = 1.0;   // right boundary
constexpr double dx = 1.0/(double)(nx-1);    // grid spacing

#endif /* EXAMPLES_POISSON1D_CONSTS_H_ */
```

The initialization function has the form:

```
void initialize(AccessHandle<std::vector<double>> & subD,
                AccessHandle<std::vector<double>> & diag,
                AccessHandle<std::vector<double>> & supD,
                AccessHandle<std::vector<double>> & rhs)
{
  create_work([=]
  {
    // first call default constructors
    subD.emplace_value();   diag.emplace_value();
    supD.emplace_value();   rhs.emplace_value();
    // resize and reset all to zeros
    subD->resize(nInn,0.0); diag->resize(nInn,0.0);
    supD->resize(nInn,0.0); rhs->resize(nInn,0.0);
    // get data pointers
    double * ptrD1 = subD->data();  double * ptrD2 = diag->data();
    double * ptrD3 = supD->data();  double * ptrD4 = rhs->data();

    // loop and set the values based on finite-difference stencil
    double x = dx;
    for (int i = 0; i < nInn; ++i)
    {
      ptrD2[i] = -2; // diagonal elements

      // sub and super diagonals
      if (i>0)
        ptrD1[i] = 1.0;
      if (i<nInn-1)
        ptrD3[i] = 1.0;

      // right hand side
      ptrD4[i] = rhsEval(x) * dx*dx;

      // correction to RHS due to known BC
      if (i==1)
        ptrD4[i] -= BC(xL);
      if (i==nInn-1)
        ptrD4[i] -= BC(xR);

      x += dx;
    }
  });
}
```

The function to solve the linear system is:

```
void solveTridiagonalSystem(AccessHandle<std::vector<double>> & subD,
                            AccessHandle<std::vector<double>> & diag,
                            AccessHandle<std::vector<double>> & supD,
                            AccessHandle<std::vector<double>> & rhs)
{
  create_work([=]
  {
    double * pta = subD->data();
    double * ptb = diag->data();
    double * ptc = supD->data();
    double * ptd = rhs->data();

    solveThomas(pta, ptb, ptc, ptd, nInn);
  });
}
```

Finally, we check for convergence by checking the L^1-norm of the error between the computed and true solution.

```
void checkFinalL1Error(AccessHandle<std::vector<double>> & solution)
{
  create_work([=]
  {
    double * ptd = solution->data();

    double error = 0.0;
    double x = dx;
    for (int i = 0; i < (int) solution->size(); ++i)
    {
      error += std::abs( trueSolution(x) - ptd[i] );
      x += dx;
    }
    std::cout << " L1 error = " << error << std::endl;
    assert( error < 1e-2 );
  });
}
```

A.6 1D Heat Equation

In this section, we solve the following simple problem:

$$\frac{\partial T(x,t)}{\partial t} = \alpha \frac{\partial^2 T(x,t)}{\partial x^2} \quad \text{in } \Omega = (0,1), \text{ with } T(0,t) = 100,\ T(1,t) = 10,\ \forall t \geq 0 \tag{A.5}$$

where $T(x,t)$ is the temperature, t is time, and α is the thermal diffusivity. The steady-state solution of this problem is a straight line connecting the left and right boundary conditions.

We discretize the spatial domain with N equally spaced points such that

$$x_i = i\Delta x, \quad \Delta x = \frac{1}{N-1}, \quad i = 0, 1, ..., N-1 \tag{A.6}$$

Similarly, in time with n_{iter} steps such that

$$t_m = m\Delta t, \quad \Delta t = \frac{t_{max}}{n_{iter}-1}, \quad m = 0, 1, ..., n_{iter}-1 \tag{A.7}$$

We use second-order finite-differences in space, and Euler method in time. Hence, the discrete version takes the form:

$$\frac{T_i^{m+1} - T_i^m}{\Delta t} = \alpha \frac{T_{i+1}^m - 2T_i^m + T_{i-1}^m}{\Delta x^2} \tag{A.8}$$

where T_i^m represents the approximate temperature at the i-th grid point, at the m-th time instant. Hence, for every grid point i, given the solution at the current time instant T_i^m, the solution at the next step is given by

$$T_i^{m+1} = T_i^m + \frac{\alpha \Delta t}{\Delta x^2}(T_{i+1}^m - 2T_i^m + T_{i-1}^m) \tag{A.9}$$

For demonstration purposes, we adopt here $\alpha = 0.0075$, discretize the domain with $N = 16$, use $n_{iter} = 2500$ time steps and consider $\Delta t = 0.05$ which is sufficiently small for the numerical method to be stable. The main constants are defined in the following header file:

```
#ifndef EXAMPLES_HEAT_1D_COMMON_H_
#define EXAMPLES_HEAT_1D_COMMON_H_

constexpr int n_iter  = 2500;      // num of iterations in time
constexpr double deltaT = 0.05;    // time step
constexpr double alpha  = 0.0075; // diffusivity

constexpr int nx = 16;             // total number of grid points
constexpr double x_min = 0.0;      // domain start x
constexpr double x_max = 1.0;      // domain end x
constexpr double deltaX = (x_max-x_min)/( (double) (nx-1) );  // cell spacing
constexpr double cfl = alpha * deltaT / (deltaX * deltaX );   // cfl condition
static_assert( cfl < 0.5, "cfl not small enough");
// alpha * DT/ DX^2
constexpr double alphadtOvdxSq = (alpha * deltaT) / (deltaX * deltaX );

constexpr double Tl = 100.0;      // left BC for temperature
constexpr double Tr = 10.0;       // right BC for temperature

// steady state solution
double steadySolution(double x)
{
  const double a = (Tl-Tr)/(x_min-x_max);
  const double b = Tl - a * x_min;
  return a*x + b;
}

#endif /* EXAMPLES_HEAT_1D_COMMON_H_ */
```

The problem involves three main stages, namely initialization, time advancing, and convergence check. We use four DARMA ranks to distribute the grid points, such that each rank handles a local grid with 4 points. In brief, the problem is setup by having each rank generate tasks for its local grid, then communicate with the neighboring ranks to get the information for the ghost points needed to update the stencil. A high-level schematic of the work-flow is shown in Figure A.1.

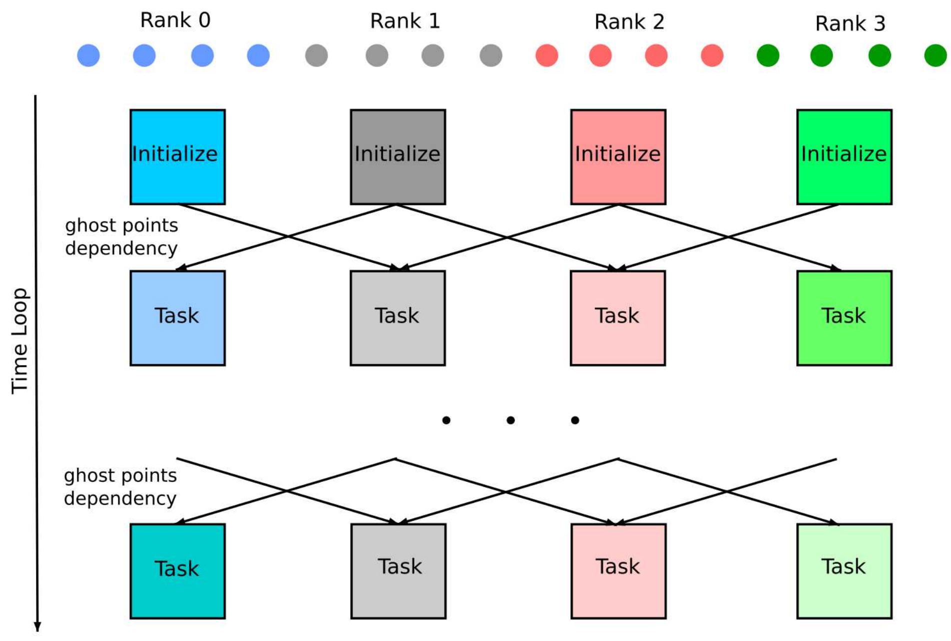

Figure A.1 Schematic of task generation for the heat 1D PDE.

The full main code is shown below.

```
#include <cmath>
#include <darma.h>
#include "../common_heat1d.h"

/*
```

```
  Full grid:
        o  o  o  o  o  o  o  o  o  o  o  o  o  o  o  o

  Distribute uniformly accross all ranks:

     +  o  o  o  o  *
                 *  o  o  o  o  *
                             *  o  o  o  o  *
                                         *  o  o  o  o  +

          r0            r1          r2          r3

  Locally, each rank owns elements:

    *  o  o  o  o  *

  where inner points are: o
        ghosts points are: *

  The points denoted with + are not needed because outside of domain,
  but exist anyway so that each local vector has same size.

  Below we use following shortcut for indices of key points:

    *    o    o   o    o   *
    lli  li           ri  rri
*/

int darma_main(int argc, char** argv)
{
  using namespace darma_runtime;
  using namespace darma_runtime::keyword_arguments_for_publication;
  darma_init(argc, argv);
  const size_t me  = darma_spmd_rank();
  const size_t n_spmd = darma_spmd_size();
  // supposed to be run with 4 ranks
  if (n_spmd!=4){
    std::cerr << "# of ranks != 4, not supported!" << std::endl;
    std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
    exit( EXIT_FAILURE );
  }

  // Figure out my neighbors. 0 or n_spmd-1, I am my own neighbor
  const bool is_leftmost = me == 0;
  const size_t left_neighbor = is_leftmost ? me : me - 1;
  const bool is_rightmost = me == n_spmd - 1;
  const size_t right_neighbor = is_rightmost ? me : me + 1;
  assert( nx % n_spmd == 0 );   // same number of points locally
  const int num_points_per_rank = nx / n_spmd;
  const int num_points_per_rank_wghosts = num_points_per_rank + 2;
  const int num_cells_per_rank = num_points_per_rank-1;

  // useful to identify local grid points
  const int lli = 0;
const int li  = 1;
const int ri  = num_points_per_rank;
const int rri = num_points_per_rank+1;

// left boundary of my local part of the grid
const double xL = is_leftmost ? 0.0 : num_points_per_rank * deltaX * me;



/************************************************************
        initialize temp field and ghost values
************************************************************/

// handle to my data
auto data = initial_access<std::vector<double>>("data", me);
// handle to ghost value for my left neighbor
auto gv_to_left = initial_access<double>("ghost_for_left_neigh", me, 0);
// handle to ghost value for my right neighbor
auto gv_to_right = initial_access<double>("ghost_for_right_neigh", me, 0);

create_work([=]
{
  data.emplace_value();
  data->resize(num_points_per_rank_wghosts, 50.0);
  auto & vecRef = data.get_reference();

  if(is_leftmost)
    vecRef[li] = Tl;
  if(is_rightmost)
    vecRef[ri] = Tr;

  // all tasks need to set the values of the ghosts
  gv_to_left.set_value(vecRef[li]);
  gv_to_right.set_value(vecRef[ri]);

});
// publish only the ghost points, since data remains local
gv_to_left.publish(n_readers=1);
gv_to_right.publish(n_readers=1);


/************************************************************
                        Time loop
************************************************************/
for (int iLoop = 0; iLoop < n_iter; ++iLoop)
{
  auto gv_from_left_neigh
    = is_leftmost ? read_access<double>("ghost_for_left_neigh",me,iLoop) :
         read_access<double>("ghost_for_right_neigh",left_neighbor,iLoop);

  auto gv_from_right_neigh
    = is_rightmost ? read_access<double>("ghost_for_right_neigh",me,iLoop) :
          read_access<double>("ghost_for_left_neigh",right_neighbor,iLoop);

    gv_to_left = initial_access<double>("ghost_for_left_neigh",me,iLoop+1);
    gv_to_right = initial_access<double>("ghost_for_right_neigh",me,iLoop+1);

    create_work([=]
    {
      auto & dataRef = data.get_reference();
      std::vector<double> my_T_wghosts(dataRef);
      my_T_wghosts[lli]   = gv_from_left_neigh.get_value();
      my_T_wghosts[rri] = gv_from_right_neigh.get_value();

      // update field only for inner points based on FD stencil
      for (int i = li; i <= ri; i++ )
      {
        double FD = my_T_wghosts[i+1]-2.0*my_T_wghosts[i]+my_T_wghosts[i-1];
        dataRef[i] = my_T_wghosts[i] + alphadtOvdxSq * FD;
      }

      // fix the domain boundary conditions
      if(is_leftmost)
        dataRef[li] = Tl;
      if(is_rightmost)
        dataRef[ri] = Tr;

      gv_to_left.set_value( dataRef[li] );
      gv_to_right.set_value( dataRef[ri] );
    });

    if (iLoop < n_iter-1){
        gv_to_left.publish(n_readers=1);
        gv_to_right.publish(n_readers=1);
    }

  } //time loop

  /************************************************************
                  Check convergence & print
  ************************************************************/

  // calculate error locally
  auto myErr = initial_access<double>("myl1error", me);
  // need a separate variable for the collective result
  auto myGlobalErr = initial_access<double>("globall1error", me);
  create_work([=]
  {
    const auto & vecRef = data.get_reference();
    // only compute error for internal points
    double error = 0.0;
    for (int i = li; i <= ri; ++i)
    {
      double xx = xL+(i-1)*deltaX;
      error += std::abs( steadySolution(xx) - vecRef[i] );
    }
    myErr.set_value(error);
    myGlobalErr.set_value(error);
  });
  // will be read by all ranks except myself
  myErr.publish(n_readers=n_spmd-1);

  // each rank performs global sum: mimicing collective
  for (int iPd = 0; iPd < n_spmd; ++iPd)
  {
    if (iPd != me)
    {
      auto iPdErr = read_access<double>("myl1error",iPd);
      create_work([=]
      {
        myGlobalErr.get_reference() += iPdErr.get_value();
      });
    }
  }

  create_work([=]
  {
    std::stringstream ss;
    ss << " global L1 error = " << myGlobalErr.get_value() << std::endl;
    std::cout << ss.str();
    if (myGlobalErr.get_value() > 1e-2)
    {
      std::cerr << "PDE solve did not converge: L1 error > 1e-2" << std::endl;
      std::cerr << " " __FILE__ << ":" << __LINE__ << '\n';
      exit( EXIT_FAILURE );
    }
  });

  darma_finalize();
  return 0;

}//end main
```

Appendix B

Rules for Making Flows

To better illustrate when particular `make_X_flow` functions are called and which Uses they belong to, we provide an illustrative set of code samples. We denote permissions as scheduling/immediate e.g. Modify/Read means scheduling privileges of Modify, immediate privileges of Read. We also indicate the Use object associated with a handle as Use(x,y) where x and y label the input and output Flow of the Use.

B.1 Modify Capture with Immediate-Modify Permissions

Consider the following code:

```
auto handle = initial_access<T>(...);
...
//handle has Use(a,b) and Modify/Modify privileges
create_work([=]{ //modify capture
  //handle has Use(c,d)
})
//handle has Use(e,f)
```

In the code sample above, the Flow objects were created as follows:

```
c = make_forwarding_flow(a, ForwardingChanges);
d = make_next_flow(c, Output);
e = make_same_flow(d, Input);
f = make_same_flow(b, Output);
```

B.2 Modify Capture without Immediate Privileges

```
auto handle = initial_access<T>(...);
...
//handle has Use(a,b) and Modify/None privileges
create_work([=]{ //modify capture
  //handle has Use(c,d)
})
//handle has Use(e,f)
```

In the code sample above, the Flow objects were created as follows:

```
c = make_same_flow(a, Input);
d = make_next_flow(c, Output);
e = make_same_flow(d, Input);
f = make_same_flow(b, Output);
```

B.3 Read Capture with Immediate Modify Privileges

```
auto handle = initial_access<T>(...);
...
//handle has Use(a,b) and Modify/Modify privileges
create_work([=]{ //read capture
  //handle has Use(c,d)
})
//handle has Use(e,f) and Modify/Read privileges
```

In the code sample above, the Flow objects were created as follows:

```
c = make_forwarding_flow(a, ForwardingChanges);
d = make_same_flow(c, OutputFlowOfReadOperation);
e = make_same_flow(c, Input);
f = make_same_flow(b, Output);
```

B.4 Read Capture with Immediate Read Privileges

```
auto handle = initial_access<T>(...);
...
//handle has Use(a,b) and Read/Read privileges
create_work([=]{ //read capture
  //handle has Use(c,d)
})
//handle has Use(a,b) and Read/Read privileges
```

In the code sample above, the Flow objects were created as follows:

```
c = make_same_flow(a, Input);
d = make_same_flow(c, OutputFlowOfReadOperation);
```

In contrast to previous cases, the newly created task does not create a new Use for the continuing context. The previous Use is considered to have continued.

Glossary

abstract machine model A model of a computer system that is designed to allow application developers to focus on the aspects of the machine that are important or relevant to performance and code structure [21].

access group An abstract (as of yet unspecified) concept. An access group is a group of tasks that *may* read a particular piece of data. Until all tasks in the access group release read privileges on the data (or a copy is made), the data can not be overwritten.

actor model An actor model covers both aspects of programming and execution models. In the actor model, applications are decomposed across objects called actors rather than processes or threads (Message Passing Interface (MPI) ranks). The actor model shares similarities with active messages. Actors send messages to other actors, but beyond simply exchanging data they can invoke remote procedure calls to create remote work or even spawn new actors. The actor model mixes aspects of SPMD in that many actors are usually created for a data-parallel decomposition. It also mixes aspects of fork-join in that actor messages can "fork" new parallel work; the forks and joins, however, do not conform to any strict parent-child structure since usually any actor can send messages to any other actor.

AMT See AMT model.

AMT model Asynchronous many-task (AMT) is a categorization of programming and execution models that break from the dominant CSP or SPMD models. Different asynchronous many-task runtime system (AMT RTS) implementations can share a common AMT model. An AMT programming model decomposes applications into small, migratable units of work (many tasks) with associated inputs (dependencies or data blocks) rather than simply decomposing at the process level (MPI ranks). An AMT execution model can be viewed as the coarse-grained, distributed memory analog of instruction-level parallelism, extending the concepts of data prefetching, out-of-order task execution based on dependency analysis, and even branch prediction (speculative execution). Rather than executing in a well-defined order, tasks execute when inputs become available. An AMT model aims to leverage all available task parallelism and pipeline parallelism, rather than rely solely on data parallelism for concurrency. The term asynchronous encompasses the idea that 1) processes (threads) can diverge to different tasks, rather than executing in the same order; and 2) concurrency is maximized (minimum synchronization) by leveraging multiple forms of parallelism. The term *many-task* encompasses the idea that the application is decomposed into many migratable units of work, to enable the overlap of communication and computation as well as asynchronous load balancing strategies.

AMT RTS A runtime system based on AMT concepts. An AMT RTS provides a specific implementation of an AMT model.

anti-dependency See Write-After-Read.

API An application programmer interface (API) is set of functions and tools provided by a library developer to allow an application programmer to interact with a specific piece of software or allow a developer to utilize prebuilt functionality.

archive In DARMA serialization, an object that performs either 1) packing operations, storing serialized values in the archive or 2) unpacking operations, deserializing values stored in the archive..

ASC The Advanced Simulation and Computing (ASC) Program supports the Department of Energy's National Nuclear Security Administration (NNSA) Defense Programs' shift in emphasis from test-based confidence to simulation-based confidence. Under ASC, computer simulation capabilities are developed to analyze and predict the performance, safety, and reliability of nuclear weapons and to certify their functionality. ASC integrates the work of three Defense programs laboratories (Los Alamos National Laboratory, Lawrence Livermore National

Laboratory, and Sandia National Laboratories) and university researchers nationally into a coordinated program administered by NNSA.

associative array An abstract data type composed of a collection of key-value pairs, such that each possible key appears just once in the collection [**?**, associative-array] Data is retrieved from an associative array via its key, rather than its address in the array.

asynchronous Asynchronous indicates two operations can happen independently without requirizing synchronization..

back end A software stack may comprise many layers, separating the user from the hardware. Each layer comprises a front end and a back end. The front end provides a set of abstractions and the user interface for the functionality implemented by the back end.

barrier Generally a synonym for global barrier. A group of processes must reach a particular execution point before any one process can continue.

bulk synchronous The bulk synchronous model of parallel computation (BSP) is defined as the combination of three attributes: 1) A number of components, each performing processing and/or memory functions; 2) A router that delivers messages point to point between pairs of components; and 3) Facilities for synchronizing all or a subset of the components at regular intervals of L time units where L is the periodicity parameter. A computation consists of a sequence of supersteps. In each superstep, each component is allocated a task consisting of some combination of local computation steps, message transmissions and (implicitly) message arrivals from other components. After each period of L time units, a global check is made to determine whether the superstep has been completed by all the components. If it has, the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep. See Reference [22] and [23] for more details.

capture In C++ the capture list specifies which variables defined outside the lambda are available for use within the lambda. Variables may be captured by value or reference. See [24] for more detail.

captured context See deferred work.

captured work See deferred work.

chare The basic unit of computational work within the Charm++ framework. Chares are essentially C++ objects that contain methods that carry out computations on an objects data asynchronously from the method's invocation.

child task A successor task in a task graph. Predecessor tasks are parent tasks. More rigorously, in a directed acyclic graph (DAG) representing task-order constraints, child task means there is a directed edge from parent to child indicating a parent happens-before child relationship..

co-design Co-design refers to a computer system design process where scientific problem requirements influence architecture design and technology and constraints inform formulation and design of algorithms and software. Co-design methodology requires the combined expertise of vendors, hardware architects, system software developers, domain scientists, computer scientists, and applied mathematicians working together to make informed decisions about features and tradeoffs in the design of the hardware, software and underlying algorithms [25].

concept A concept is a description of the supported operations on a type to be used in generic programming. In C++, there is no language level support for concepts (yet), but the idea can still be applied to C++ templates and deduced types in the context of API specification. DARMA performs most of its concept checking using the `void_t` detection idiom [26]..

concurrency A condition of a system in which multiple tasks are logically active at one time.

conservative execution The runtime system only spawns tasks in parallel that are guaranteed not to conflict. The application exposes Read-After-Write (RAW)/Write-After-Read (WAR) conflicts, allowing the runtime system to decide which tasks can safely run in parallel. Independent threads do not need to explicitly synchronize. Execution begins with zero concurrency and grows conservatively to the maximum allowed concurrency.

continuing context the code in the outer scope after a `create_work`.

coordination semantics The operations to support communication between different computation activities. Independent parallel workers never directly communicate, rather they "coordinate" indirectly via a key-value store or tuple space. Linda is a notable programming language with coordination semantics.

copy-on-write data-flow execution This is an intermediate between conservative execution and phased execution, with the additional constraint that the application guarantees no WAR conflicts. Tasks are written to follow a write-once, read-many policy when necessary to avoid anti-dependencies. The only synchronizations required are RAW, ensuring that a value exists before a task can run. Similar to conservative execution, tasks spawn once all their RAW dependencies are met, forking new concurrency. Once running, tasks do not synchronize because there are no WAR conflicts to avoid. This approach often has higher memory requirements, and the necessary garbage collection adds complications.

CSP CSP (communicating sequential processes) is the most popular concurrency model for science and engineering applications, often being synonymous with SPMD. CSP covers execution models where a usually fixed number of independent workers operate in parallel, occasionally synchronizing and exchanging data through inter-process communication. Workers are *disjoint processes*, operating in separate address spaces. This also makes it generally synonymous with message-passing in which data exchanges between parallel workers are copy-on-read, creating disjoint data parallelism. The term sequential is historical and CSP is generally applied even to cases in which each "sequential process" is composed of multiple parallel workers (usually threads).

DARMA DARMA is an AMT portability layer serving as a vehicle for community-based co-design activities. The layer aims to 1) insulate applications from runtime system and hardware idiosyncrasies, 2) improve AMT runtime programmability by co-designing an API directly with application developers, 3) synthesize application co-design activities into meaningful requirements for runtimes, and 4) facilitate AMT design space characterization and definition, accelerating the development of AMT best practices.

data model A model capturing assumptions or restrictions on the structure of data.

data parallelism A type of parallelism that involves carrying out a single task and/or instruction on different segments of data across many computational units. Data parallelism is best illustrated by vector processing or single-instruction, multiple-data (SIMD) operations on central processing units (CPUs) and Many Integrated Core Architecture (MIC)s or typical bulk synchronous parallel applications.

data-flow dependency A data dependency where a set of tasks or instructions require a certain sequence to complete without causing race conditions. Data-flow dependency types include Write-After-Read, Read-After-Write and Write-After-Write.

declarative A style of programming that focuses on using statements to define what a program should accomplish rather than how it should accomplish the desired result.

deferred execution Execution of work is not performed until all dependencies are met.

deferred task See deferred work. A task instantiated in the application code which, instead of executing immediately as would be done in a sequential C++ code, is delayed while other tasks execute and are created. The term is applied to tasks that, even once created, immediately execute. A more precise term would be "deferrable task", but without ambiguity we use the adjective deferred to match previous literature.

deferred work See deferred task. Work performed by code inside the capturing lambda passed to the `create_work` construct (as well as other deferred constructs which may be added to future versions of the specification).

DEP The method by which changes are made to the DARMA specification.

dependency See Read-After-Write.

DHT An implementation of a key-value map (table) that relies a consistent hash of keys and a partition of the key space to distribute storage of the table across a distributed system.

distributed memory model Each processor has its own private memory. Computational tasks can only operate on their local data. When remote data is required, it is communicated between the remote and local tasks.

DSL Domain specific languages (DSL) are a subset of programming languages that have been specialized to a particular application domain. Typically, DSL code focuses on what a programmer wants to happen with respect to their application and leaves the runtime system to determine how the application is executed.

EDSL A domain specific language (DSL) that is defined as a library for a generic host programming language. The embedded domain specific language inherits the generic language constructs of its host language - sequencing, conditionals, iteration, functions, etc. - and adds domain-specific primitives that allow programmers to work at a much higher level of abstraction.

elastic task a task with inherent parallelism, as opposed to a sequential task with no parallelism that will execute serially. These parallel tasks are termed elastic since they usually involve a flexible amount of parallelism, executing faster as more processors are allocated to running the task.

event-based The term event-based covers both programming models and execution models in which an application is expressed and managed as a set of events with precedence constraints, often taking the form of a directed graph of event dependencies.

execution model A parallel execution model specifies how an application creates and manages concurrency. This includes, e.g., CSP (communicating sequential processes), strict fork-join, or event-based execution. These classifications distinguish whether many parallel workers begin simultaneously (e.g., CSP) and synchronize to reduce concurrency or if a single top-level worker forks new tasks to increase concurrency. These classifications also distinguish how parallel hazards (WAR, RAW, Write-After-Write (WAW)) are managed. Execution models fall into the follwing broad categories: conservative execution, phased execution, copy-on-write dataflow execution, and speculative execution. In many cases, the programming model and execution model are closely tied and therefore not distinguished. In other cases, the way execution is managed is decoupled from the programming model in runtime systems with declarative programming models like Legion or Uintah. The execution model is implemented in the runtime system .

execution space A abstract machine model abstraction used to describe where work is executed.

execution stream A top-level task (no predecessors) that is guaranteed to make forward progress. No other restrictions apply. An execution stream (being a task) may communicate and may be interrupted. Execution streams are often part of an SPMD launch and therefore automatically given a unique integer ID for each particular stream. We use the term rank for this unique ID to match MPI terminology.

fetch A fetch operation reads values from the key-value store. Fetches are requests that must be satisfied with a matching publish into the key-value store. In DARMA, fetches implicitly occur when read-only handles are created through `read_access`..

fork-join A model of concurrent execution in which child tasks are forked off a parent task. When child tasks complete, they synchronize with join partners to signal execution is complete. Fully strict execution requires join edges be from parent to child while terminally strict requires child tasks to join with grandparent or other ancestor tasks. This style of execution contrasts with SPMD in which there are many parallel sibling tasks running, but they did not fork from a common parent and do not join with ancestor tasks.

front end A software stack may comprise many layers, separating the user from the hardware. Each layer comprises a front end and a back end. The front end provides a set of abstractions and the user interface for the functionality implemented by the back end.

fully strict Fully strict fork-join execution requires join edges between parent and child tasks.

functional A style of programming that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

handle In DARMA, types are wrapped in a lightweight wrapper we term handle. Handles replace conventional C++ variables as the means for accessing a data value. The handle wrapper provides a control block used by DARMA in creating tasks..

immediate permissions The permissions for a handle that applies immediately at the current point in execution. For immediate Read permissions, `handle.get_value()` can be called. For immediate Write permissions, `handle.set_value()` and `handle.get_reference()` can also be called..

imperative A style of programming where statements change the state of a program to produce a specific result. This contrasts to declarative programming that focuses on defining the desired result without specifying how the result is to be accomplished.

interference test A test on two operations to see if they can safely run in parallel or if they conflict and must run in sequence. Two operations on different data never interfere. For operations on the same data, if the operation is read-only, there is no interference. Operations writing and reading the same data do interfere.

introspection The ability of a program to examine properties of an object at runtime.

key-value store A database that has an associative array as its underlying data model. In DARMA, a key-value store.

keyword argument An argument that is passed to a function as a keyword=value.

lambda In C++ a lambda is a mechanism for defining an unnamed function object at the location where it is invoked. Lambdas are capable of capturing (see capture) variables in scope. See [24] for more detail.

leaf task A task with no direct successors, i.e. at the end of a task graph branch.

Linda Linda is a model of coordination and communication among several parallel processes operating upon objects stored in and retrieved from shared, virtual, associative memory [27].

memory model Describes the interactions of processing entities (e.g., threads) with memory, including how they store and retrieve data.

memory space An abstract machine model abstraction used to describe where data resides.

MIC Intel Many Integrated Core Architecture or Intel MIC is a coprocessor computer architecture developed by Intel incorporating earlier work on the Larrabee many core architecture, the Teraflops Research Chip multicore chip research project, and the Intel Single-chip Cloud Computer multicore microprocessor. Prototype products codenamed Knights Ferry were announced and released to developers in 2010. The Knights Corner product was announced in 2011 and uses a 22 nm process. A second generation product codenamed Knights Landing using a 14 nm process was announced in June 2013. Xeon Phi is the brand name used for all products based on the Many Integrated Core architecture.

migratable Migratable is used in DARMA to indicate that something can be serialized, transported or to a remote process, deserialized, and used on that remote process..

operation The fundamental, indivisible (not-interruptible) work unit. Operations are closed, unable to communicate with other operations and unable to add/release variables.

overdecomposition A problem which is decomposed into more tasks than compute units. i Usually applied to a data-parallel overdecomposition in which, e.g., an array is broken in 4x times as many chunks as there are compute units. Rather than have each compute unit execute one task of cost 4, each compute units 4 tasks of cost 1. The increased granularity enables dynamic load balancing at the cost of increased scheduling overheads.

parent task A predecessor task in a task graph. Successors tasks are child tasks. More rigorously, in a DAG representing task-order constraints, parent task means there is a directed edge from parent to child indicating a parent happens-before child relationship..

perfect forwarding A mechanism for forwarding arguments of one function to another in C++ that avoids copying and maintains lvalue/rvalue nature of the arguments. See [28] for more detail.

phase barrier A fine-grained barrier used by a group of processes (potentially only two processes) to agree that a particular phase of execution has completed. Phase barriers are not rigorously defined and may be non-blocking and multiple phase barriers may be active at a given time.

phased execution The runtime system spawns many tasks in parallel. Where RAW or WAR conflicts may exist, a phase barrier is executed to guarantee safe execution. The term phase barrier has previously been used in Legion [**?**] and X10 [**?**]. Barriers may be local operations or global collectives. Execution begins with maximum parallelism and concurrency decreases when necessary to satisfy synchronization constraints.

pipeline parallelism Pipeline parallelism is achieved by breaking up a task into a sequence of individual sub-tasks, each of which represents a stage whose execution can be overlapped.

POD In C++, POD stands for Plain Old Data—that is, a class or struct without constructors, destructors and virtual members functions and all data members of the class are also POD.

positional argument An argument passed to a function, whose corresponding parameter is inferred by the argument's position within the function call.

precondition When applied to tasks, preconditions are the set of events that must occur before a task can safely run, leading to the data the task operates on being in the correct state. Preconditions usually are either other tasks, data staging or copying, or communication operations.

procedural A style of programming where developers define step by step instructions to complete a given function/task. A procedural program has a clearly defined structure with statements ordered specifically to define program behavior.

process Used here as a process in the UNIX sense. Each process will have its own address space and global variables. The process begins from a singly-defined int main(...) function.

programming language A programming language is a syntax and code constructs for implementing one or more programming models. For example, the C++ programming language supports both functional and procedural imperative programming models.

programming model A parallel programming model is an abstract view of a machine and set of first-class constructs for expressing algorithms. The programming model focuses on how problems are decomposed and expressed. In MPI, programs are decomposed based on MPI ranks that coordinate via messages. This programming model can be termed SPMD, decomposing the problem into disjoint (non-conflicting) data regions. Charm++ decomposes problems via migratable objects called chares that coordinate via remote procedure calls (entry methods). Legion decomposes problems in a data-centric way with logical regions. All parallel coordination is implicitly expressed via data dependencies. The parallel programming model covers how an application *expresses* concurrency. In many cases, the execution model and programming model are closely tied and the same term has been used to describe both an execution model and programming model, e.g. CSP (communicating sequential processes).

rank A unique integer identifier for an execution stream created in an SPMD launch. The term rank matches the MPI notion of a unique process ID in an MPI communicator.

RDMA Remote direct memory access (RDMA) is a direct memory access from the memory of one computer into that of another without involving either one's operating system. This permits high-throughput, low-latency networking, which is especially useful in massively parallel computing.

Read-After-Write Read after write (RAW) is a standard data dependency (or potential hazard) where one instruction or task requires, as an input, a data value that is computed by some other instruction or task.

reference counted pointer An abstract data type that stores a traditional pointer, along with the number of shared references to that pointers memory location. Objects referenced by the contained raw pointer are only destroyed when all copies of the reference counted pointer are destroyed.

remote procedure invocation See RPC.

runtime system A parallel runtime system primarily implements portions of an execution model, managing how and where concurrency is managed and created. Runtime systems therefore control the order in which parallel work (decomposed and expressed via the programming model) is actually performed and executed. Runtime systems can range greatly in complexity. A runtime could only provide point-to-point message-passing, for which the runtime only manages message order and tag matching. A full MPI implementation automatically manages collectives and global synchronization mechanisms. Legion handles not only data movement but task placement and out-of-order task execution, handling almost all aspects of execution in the runtime. Generally, parallel execution requires managing task placement, data placement, concurrency creation, concurrency managed, task ordering, and data movement. A runtime comprises all aspects of parallel execution that are not explicitly managed by the application.

scheduling permissions The permissions for a handle when scheduling new tasks, but which may not apply immediately. A task may schedule further tasks with read privileges in certain cases even if the data cannot be immediately read..

semantics A mathematical model representing the intended computational behavior of program.

sequential semantics Computational behavior of code is equivalent to running it sequentially, in program order.

serialization The process of converting a C++ object into a sequence of bytes that can be transmitted over the network or stored.

SIMD The term single-instruction multiple-data (SIMD) refers to a type of instruction level parallelism where an individual instruction is synchronously executed on different segments of data. This type of data parallelism is best illustrated by vector processing.

slicing A subset of an array. The slice can either be across array indices or, if each array entry, a subset of the fields within each class. Slices are defined only abstractly here, as slices may be in-place, referring to the original array data or the slice may create a copy of values.

speculative execution Potential data hazards are ignored and, in some cases, work is performed prior to whether or not it is known whether it will be required. By performing the work speculatively, the delay associated with waiting to know whether or not the work was in fact required are avoided. Conflicts that are detected after the fact lead to rollback or recovery.

SPMD The term single-program multiple-data (SPMD) refers to a parallel programming model where the same tasks are carried out by multiple processing units but operate on different sets of input data. This is the most common form of parallelization and often involves multithreading on a single compute node and/or distributed computing using MPI communication.

subtask Any task instantiated with `create_work` will be a subtask of the task running at the time `create_work` was invoked. For sequential semantics, a task cannot complete until all of its subtasks have completed..

task The work unit explicitly instantiated by the application developer through `create_work`. Currently a task has no restrictions on behavior, other than a gurantee of forward progress. It can be interrupted and communicate (indirectly through coordination) with other tasks. The fundamental (indivisible and interruptible) work unit (operation) is not instantiated directly in the application. Thus tasks are the more fundamental concept in the programming model.

task elasticity See elastic task.

task parallelism A type of parallelism that focuses on completing multiple tasks simultaneously over different computational units. These tasks may operate on the same segment of data or many different datasets.

task stealing See work stealing.

template metaprogramming In template metaprogramming templates are used by a compiler to generate additional source code, (e.g., compile-time constants, data structures, funcitons), which is merged by the compiler with the rest of the user-provided source code prior to compilation.

terminally strict Terminally strict fork-join execution requires child tasks to join with grandparent or other ancestor tasks.

thread pool A preallocated (usually already spawned) group of threads used for implementing thread-parallel applications. Instead of allocating a new thread (with corresponding stack resources and initialization overheads), a pool of ready and waiting threads are maintained. Threads are chosen from the thread pool to execute new tasks.

translation layer The C++ template metaprogramming layer between the DARMA front end and the set of abstract classes that must be implemented by an implementation of the back end.

Trilinos The Trilinos Project is an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems [29].

tuple A tuple is a finite ordered list of elements. See [30] for more detail.

tuple space A repository of tuples that can be accessed concurrently, used to relate input to output patterns. A tuple space served as the underpinning to Linda programming language. Tuple spaces can be considered a generalization of a key-value stores. Implementations of tuple spaces have been developed for a number of other programming models including Java and Python. See [31] for more detail.

vector processing A vector processing is performed by a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called vectors, compared to scalar processors, whose instructions operate on single data items. Vector processing can greatly improve performance on certain workloads, notably numerical simulation and similar tasks. Vector machines appeared in the early 1970s and dominated supercomputer design through the 1970s into the 1990s, notably the various Cray platforms. As of 2015 most commodity CPUs implement architectures that feature instructions for a form of vector processing on multiple (vectorized) data sets, typically known as SIMD. Common examples include MMX, Streaming SIMD Extensions (SSE), AltiVec and Advanced Vector Extensions (AVX).

work stealing The act of one computational unit (thread/process), which has completed it's workload, taking some task/job from another computational unit. This is a basic method of distributed load balancing.

Write-After-Read Write after read (WAR), also known as an anti-dependency, is a potential data hazard where a task or instruction has required input(s) that are later changed. An anti-dependency can be removed at instruction-level through register renaming or a task-level through copy-on-read or copy-on-write.

Write-After-Write Write after write (WAW), also known as an output dependency, is a potential data hazard where data dependence is only written (not read) by two or more tasks. In a sequential execution, the value of the data will be well defined, but in a parallel execution, the value is determined by the execution order of the tasks writing the value.

zero-copy Zero-copy transfers are data transfers that occur directly from send to receive location without any additional buffering. Data is put immediately on the wire on the sender side and stored immediately in the final receive buffer off the wire on the receiver side. This usually leverages RDMA operations on pinned memory.

References

[1] R. Stevens, A. White, S. Dosanjh, A. Geist, B. Gorda, K. Yelick, J. Morrison, H. Simon, J. Shalf, J. Nichols, and M. Seager, "Architectures and technology for extreme scale computing," U. S. Department of Energy, Tech. Rep., 2009. [Online]. Available: http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Arch_tech_grand_challenges_report.pdf 9

[2] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, V. Pascucci, J. Ahrens, E. W. Bethel, H. Childs, J. Huang, K. Joy, Q. Koziol, G. Lofstead, J. S. Meredith, K. Moreland, G. Ostrouchov, M. Papka, V. Vishwanath, M. Wolf, N. Wright, and K. Wu, *Scientific Discovery at the Exascale, a Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization*, 2011. [Online]. Available: http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Exascale-ASCR-Analysis.pdf 9

[3] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257 9, 13

[4] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status," LLNL, Tech. Rep. 782261, September 2014. [Online]. Available: https://e-reports-ext.llnl.gov/pdf/782261.pdf 9, 13

[5] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar, "OCR: The Open Community Runtime Interface," Tech. Rep., June 2015. [Online]. Available: https://xstack.exascale-tech.com/git/public?p=xstack.git;a=blob;f=ocr/spec/ocr-1.0.0.pdf;hb=HEAD 9

[6] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "Stapl: an adaptive, generic parallel c++ library," in *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, 2003, pp. 193–208. [Online]. Available: https://parasol.tamu.edu/publications/download.php?file_id=663 9

[7] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *SC '12: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2389086 9

[8] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *PACT 2014: 23rd International Conference on Parallel Architectures and Compilation*, 2014, pp. 263–276. 9

[9] T. Heller, H. Kaiser, and K. Iglberger, "Application of the parallex execution model to stencil-based problems," *Comput. Sci.*, vol. 28, pp. 253–261, 2013. 9

[10] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *OOPSLA 1993: 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1993, pp. 91–108. 9

[11] J. D. D. S. Germain, S. G. Parker, C. R. Johnson, and J. McCorquodale, "Uintah: a massively parallel problem solving environment," 2000. [Online]. Available: http://content.lib.utah.edu/u?/ir-main,29551 9

[12] E. A. Luke, "Loci: A deductive framework for graph-based algorithms," in *Computing in Object-Oriented Parallel Environments (3rd ISCOPE'99)*, ser. Lecture Notes in Computer Science (LNCS), S. Matsuoka, R. R. Oldehoeft, and M. Tholburn, Eds. San Francisco, California, USA: Springer-Verlag (New York), Dec. 1999, vol. 1732, pp. 142–153. 9

[13] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computer Science and Engineering*, vol. 15, pp. 36–45, 2013. 9

[14] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A Generic Distributed DAG Engine for High Performance Computing," *Parallel Comput.*, vol. 38, pp. 37–51, 2012. 9

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Notices*, vol. 30, pp. 207–216, 1995. 9

[16] J. Bennett, R. Clay *et al.*, "ASC ATDM Level 2 milestone #5325: Asynchronous Many-Task runtime system analysis and assessment for next generation platforms," Sandia National Laboratories, Tech. Rep. SAND2015-8312, 2015. 10

[17] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/344588.344618 11

[18] N. Vydyanathan, S. Krishnamoorthy, G. M. Sabin, U. V. Catalyurek, T. Kurc, P. Sadayappan, and J. H. Saltz, "An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 8, pp. 1158–1172, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2008.219 11

[19] N. Fauzia, V. Elango, M. Ravishankar, J. Ramanujam, F. Rastello, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 53:1–53:29, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2541228.2555309 11

[20] W. Zhang, A. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "Boxlib with tiling: An AMR software framework," Apr. 12 2016, comment: Accepted for publication in SIAM J. on Scientific Computing. [Online]. Available: http://arxiv.org/abs/1604.03570 13

[21] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. T. Stark, D. Unat, and N. J. Wright, "Abstract machine models and proxy architectures for exascale computing," in *Co-HPC@SC*. IEEE, 2014, pp. 25–32. [Online]. Available: http://dl.acm.org/citation.cfm?id=2689669 105

[22] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181 106

[23] Bulk synchronous parallel. [Online]. Available: https://en.wikipedia.org/wiki/Bulk_synchronous_parallel 106

[24] Lambda functions. [Online]. Available: http://en.cppreference.com/w/cpp/language/lambda 106, 109

[25] DOE ASCR co-design. [Online]. Available: http://science.energy.gov/ascr/research/scidac/co-design/ 106

[26] W. E. Brown. (2015) Proposing standard library support for the c++ detection idiom. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4436.pdf 106

[27] N. J. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman, "The linda alternative to message-passing systems," *Parallel Comput.*, vol. 20, pp. 633–655, 1994. 109

[28] Perfect Forwarding. [Online]. Available: http://en.cppreference.com/w/cpp/utility/forward 110

[29] Trilinos. [Online]. Available: https://trilinos.org 112

[30] Tuple: Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Tuple 112

[31] Tuple Space: Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Tuple_space 112

DISTRIBUTION:

,

1	MS	0899	Technical Library, 8944 (electronic copy)