DARMA

Janine C. Bennett, Jonathan Lifflander, David S. Hollman, Jeremiah Wilke, Hemanth Kolla, Aram Markosyan, Nicole Slattengren, Robert L. Clay (PM)

Charm++ Workshop 2017
April 17th, 2017
What is DARMA?

DARMA is a C++ abstraction layer for asynchronous many-task (AMT) runtimes.

It provides a set of abstractions to facilitate the expression of tasking that map to a variety of underlying AMT runtime system technologies.

Sandia’s ATDM program is using DARMA to inform its technical roadmap for next generation codes.
2015 study to assess leading AMT runtimes led to DARMA

Aim: inform Sandia’s technical roadmap for next generation codes

- Broad survey of many AMT runtime systems
- Deep dive on Charm++, Legion, Uintah

**Programmability:** Does this runtime enable efficient expression of ATDM workloads?

**Performance:** How performant is this runtime for our workloads on current platforms and how well suited is this runtime to address future architecture challenges?

**Mutability:** What is the ease of adopting this runtime and modifying it to suit our code needs?
2015 study to assess leading AMT runtimes led to DARMA

Aim: inform Sandia’s technical roadmap for next generation codes

- **Conclusions**
  - AMT systems show great promise
  - Gaps in requirements for Sandia applications
  - No common user-level APIs
  - Need for best practices and standards

- **Survey recommendations led to DARMA**
  - C++ abstraction layer for AMT runtimes
  - Requirements driven by Sandia ATDM applications
  - A single user-level API
  - Support multiple AMT runtimes to begin identification of best practices
sandia ATDM applications drive requirements and developers play active role in informing front end API

- Application feature requests
  - Sequential semantics
  - MPI interoperability
  - Node-level performance portability layer interoperability (Kokkos)
  - Collectives
  - Runtime-enabled load-balancing schemes

Abstractions that facilitate the expression of tasking
Mapping to a variety of AMT runtime system technologies
DARMA provides a unified API to application developers for expressing tasks.

Mapping to a variety of AMT runtime system technologies.
Application code is translated into a series of backend API calls to an AMT runtime.

Application

DARMA

Glue Code (Specific to each runtime)

Runtime

OS/Hardware

Common API across runtimes

Front End API (Application User)

Translation Layer

Back End API (Specification for Runtime)

Mapping to a variety of AMT runtime system technologies

Not all runtimes provide the same functionality.
Application code is translated into a series of backend API calls to an AMT runtime

Mapping to a variety of AMT runtime system technologies
Considerations when developing a backend API that maps to a variety of runtimes

- AMT runtimes often operate with a directed acyclic graph (DAG)
  - Captures relationships between application data and inter-dependent tasks
Considerations when developing a backend API that maps to a variety of runtimes

- AMT runtimes often operate with a directed acyclic graph (DAG)
  - Captures relationships between application data and inter-dependent tasks
- DAGs can be annotated to capture additional information
  - Tasks’ read/write usage of data
  - Task needs a subset of data

Mapping to a variety of AMT runtime system technologies
Considerations when developing a backend API that maps to a variety of runtimes

- AMT runtimes often operate with a directed acyclic graph (DAG)
  - Captures relationships between application data and inter-dependent tasks
- DAGs can be annotated to capture additional information
  - Tasks’ read/write usage of data
  - Task needs a subset of data
- Additional information enables runtime to reason more completely about
  - When and where to execute a task
  - Whether to load balance
- Existing runtimes leverage DAGs with varying degrees of annotation

Mapping to a variety of AMT runtime system technologies
DARMA captures data-task dependency information and the runtime builds and executes the DAG.

Mapping to a variety of AMT runtime system technologies
Abstractions that facilitate the expression of tasking
DARMA front end abstractions for data and tasks are co-designed with Sandia ATDM application scientists.

Abstractions that facilitate the expression of tasking:

- Provide abstractions to simplify capturing of data-task dependencies.

Diagram:

- Application
- DARMA
- Glue Code (Specific to each runtime)
- Runtime
- OS/Hardware

Front End API (Application User)
Translation Layer
Back End API (Specification for Runtime)

Common API across runtimes

Glue Code

Abstractions that facilitate the expression of tasking
DARMA Data Model

How are data collections/data structures described?

- Asynchronous smart pointers wrap application data
  - Encapsulate data effect information used to build and annotate the DAG
    - Permissions information (type of access, Read, Modify, Reduce, etc.)
  - Enable extraction of parallelism in a data-race-free manner

How are data partitioning and distribution expressed?

- There is an explicit, hierarchical, logical decomposition of data
  - AccessHandle<T>
    - Does not span multiple memory spaces
    - Must be serialized to be transferred between memory spaces
  - AccessHandleCollection<T, R>
    - Expresses a collection of data
    - Can be mapped across memory spaces in a scalable manner
- Distribution of data is up to individual backend runtime

Abstractions that facilitate the expression of tasking
DARMA Control Model

How is parallelism achieved?

- **create_work**
  - A task that doesn’t span multiple execution spaces
  - Sequential semantics: the order and manner (e.g., read, write) in which data (AccessHandle) is used determines what tasks *may* be run in parallel

- **create_concurrent_work**
  - Scalable abstraction to launch across distributed systems
  - A collection of tasks that make simultaneous forward progress
  - Sequential semantics supported across different task collections based on order and manner of AccessHandleCollection usage

How is synchronization expressed?

- DARMA *does not* provide explicit temporal synchronization abstractions
- DARMA *does* provide data coordination abstractions
  - Sequential semantic coordination between participants in a task collection
  - Asynchronous collectives between participants in a task collection

*Abstractions that facilitate the expression of tasking*
Using DARMA to inform Sandia’s technical roadmap
Currently there are three backends in various stages of development

Using DARMA to inform Sandia’s ATDM technical roadmap
2017 study: Explore programmability and performance of the DARMA approach in the context of ATDM codes

Using DARMA to inform Sandia’s ATDM technical roadmap
2017 study: Explore programmability and performance of the DARMA approach in the context of ATDM codes

- Kernels and proxies
  - Form basis for programmability assessments
  - Will be used to explore performance characteristics of the DARMA-Charm++ backend

- Simple benchmarks enable studies on
  - Task granularity
  - Overlap of communication and computation
  - Runtime-managed load balancing

- These early results are being used to identify and address bottlenecks in DARMA-Charm++ backend in preparation for studies with kernels/proxies

*Using DARMA to inform Sandia’s ATDM technical roadmap*
DARMA’s Concurrency Abstractions
Asynchronous smart pointers enable extraction of parallelism in a data-race-free manner

darma::AccessHandle<T> enforces **sequential semantics**: it uses the order in which data is accessed in your program and how it is accessed (read/write/etc.) to automatically extract parallelism

<table>
<thead>
<tr>
<th>Permission Level</th>
<th>Permission Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td><strong>Scheduling</strong></td>
</tr>
<tr>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td><strong>Immediate</strong></td>
</tr>
<tr>
<td>Reduce</td>
<td></td>
</tr>
</tbody>
</table>

*Abstractions that facilitate the expression of tasking*
Tasks are annotated in the code via a lambda or functor interface

Tasks can be recursively nested within each other to generate more subtasks

C++ Lambdas

darma::create_work(
    [=]{
        /*do some work*/
    }
);

This is the C++ 11 syntax for writing an anonymous function that captures variables by value.

C++ Functors

struct MyFun {
    void operator()(...) {
        /* do some work */
    }
};
darma::create_work<MyFun>(...)

Functors are for larger blocks of code that may be reused and migrated by the backend to another memory space.

Abstractions that facilitate the expression of tasking
Example: Putting tasks and data together

Example Program

```cpp
AccessHandle<int> my_data;

Darma::create_work([=]{
    my_data.set_value(29);
});

Darma::create_work(
    reads(my_data), [=]{
        cout << my_data.get_value();
    }
);

Darma::create_work(
    reads(my_data), [=]{
        cout << my_data.get_value();
    }
);

Darma::create_work([=]{
    my_data.set_value(31);
});
```

DAG (Directed Acyclic Graph)

These two tasks are concurrent and can be run in parallel by a DARMA backend!

Abstractions that facilitate the expression of tasking
Smart pointer collections can be mapped across memory spaces in a scalable manner

AccessHandleCollection\(<T, R>\) is an extension to AccessHandle\(<T>\) that expresses a collection of data

```cpp
AccessHandleCollection<vector<double>, Range1D> mycol =
    darma::initial_access_collection(
        index_range = Range1D(10)
    );
```

Every element in the collection contains a `vector<double>`

`Range1D` is a potentially user-defined (or domain-specific) index range, a C++ object that describes the extents of the collection along with providing a corresponding index class for accessing an element.

Each indexed element is an `AccessHandle<vector<double>>`

Abstractions that facilitate the expression of tasking
Tasks can be grouped into collections that make concurrent forward progress together

Task collections are a scalable abstraction to efficiently launch communicating tasks across large-scale distributed systems

```cpp
create_concurrent_work<MyFun>(
    index_range = Range1D(5)
);
```

This call to `create_concurrent_work` launches a set of tasks, the size of which is specified by an index range, `Range1D`, that is passed as an argument.

```cpp
struct MyFun {
    void operator() (Index1D i) {
        int me = i.value;
        /* do some work */
    }
};
```

Each element in the task collection is passed an `Index1D` within the range, used by the programmer to express communication patterns across elements in the collection.

*Abstractions that facilitate the expression of tasking*
Putting task collections and data collections together

Example Program

```cpp
auto mycol = initial_access_collection(
    index_range = Range1D(10));

create_concurrent_work<MyFun>(
    mycol, index_range = Range1D(10));

create_concurrent_work<MyFun>(
    mycol, index_range = Range1D(10));
```

A mapping must exist between the data index ranges and task index range. In this case, since the three ranges are identical in size and type, a one-to-one identity map is automatically applied.

Abstractions that facilitate the expression of tasking

Generated DAG

Sequential Semantics

Scalable Graph Refinement

Index 0  Index 1  Index 9
Tasks in different execution streams can communicate via publish/fetch semantics

**Execution Stream A**

```
AccessHandle<int> my_data =
    initial_access<int>("my_key");

darma::create_work([&]{
    my_data.set_value(29);
});

my_data.publish(version="a");

darma::create_work([&]{
    my_data.set_value(31);
});
```

**Execution Stream B**

```
AccessHandle<int> other_data =
    read_access("my_key", version="a");

darma::create_work([&]{
    cout << other_data.get_value();
});

other_data = nullptr;
```

Abstractions that facilitate the expression of tasking

If the read_access is on another node it might be send across the network.
Tasks in different execution streams can communicate via publish/fetch semantics

**Execution Stream A**

```cpp
tasks::create_work(=[
    my_data.set_value(29);
]);
my_data.publish(version="a");
tasks::create_work(=[
    my_data.set_value(31);
]);
```

**Execution Stream B**

```cpp
tasks::create_work(=[
    cout << other_data.get_value();
]);
other_data = nullptr;
```

Abstractions that facilitate the expression of tasking
Tasks in different execution streams can communicate via publish/fetch semantics

Execution Stream A

AccessHandle<int> my_data =
  initial_access<int>("my_key");

darma::create_work(
  [=]{
    my_data.set_value(29);
  });

my_data.publish(version="a");

darma::create_work(
  [=]{
    my_data.set_value(31);
  });

Execution Stream B

AccessHandle<int> other_data =
  read_access("my_key", version="a");

darma::create_work(
  [=]{
    cout << other_data.get_value();
  });

other_data = nullptr;

Abstractions that facilitate the expression of tasking

Potential DAG 2

If the read_access is on the same node a back end runtime can generate an alternative DAG without the transfer.
A mapping between data and task collections determines access permissions between tasks and data.

```cpp
auto mycol = initial_access_collection<int>(
    index_range = Range1D(10));
create_concurrent_work<MyFun>(
    mycol, index_range = Range1D(10));

struct MyFun {
    void operator()(
        Index1D i, AccessHandleCollection<int> col)
    {
        int me = i.value, mx = i.max_value;
        auto my_elm = col[i].local_access();
        my_elm.publish(version="x");
        auto neighbor = me-1 < 0 ? mx : me-1;
        auto other_elm = col[neighbor].read_access(version="x");
        create_work([=]{
            cout << “neighbor = ” << other_elm.get_value() << endl;
        });
    }
};
```

Abstractions that facilitate the expression of tasking
The Charm++ Backend

- About 13k lines of code
- Maps a task-based system to an object-oriented one
- Much of the code is dealing with lookahead and data versioning (or generations)
  - Scalable effect management and refinement
- Each `create_concurrent_work` maps to a chare array in the “present” or future
  - Lookahead allows the system to determine where the next task collection will execute and pipeline work
  - The set of data inputs to the `create_concurrent_work` dictate which chare array instance is used (of if a new one is created)
  - By reusing a chare arrays that have the same data inputs from the past, persistence is retained
  - An `AccessHandleCollection` may span multiple chare arrays, element by element depending on the versioning
DARMA’s programming model enables runtime-managed, measurement-based load balancing.

Using DARMA to inform Sandia’s ATDM technical roadmap.
A latency-intolerant benchmark highlights overheads as grain size decreases.

At this scale, each iteration is less than 5ms long. This benchmark has tight synchronization every iteration.

Using DARMA to inform Sandia’s ATDM technical roadmap...
Increased asynchrony in the application enables the runtime to overlap communication and computation.

Scalability improves with asynchronous iterations. Requires only minor changes to application code.

Using DARMA to inform Sandia’s ATDM technical roadmap.
DARMA’s programming model enables runtime-managed, measurement-based load balancing.

Load balancing does not require changes to the application code.

Using DARMA to inform Sandia’s ATDM technical roadmap
Stencil benchmark is not latency tolerant and highlights runtime overheads when task-granularity is small.

Using DARMA to inform Sandia’s ATDM technical roadmap.

At this scale, each iteration is less than 5ms long.
Increased asynchrony in application enables runtime to overlap communication and computation.

Using DARMA to inform Sandia’s ATDM technical roadmap.
Summary: DARMA seeks to accelerate discovery of best practices

- **Application developers**
  - Use a unified interface to explore a variety of different runtime system technologies
  - Directly inform DARMA’s user-level API via co-design requirements/feedback

- **System software developers**
  - Acquire a synthesized set of requirements via the backend specification
  - Directly inform backend specification via co-design feedback
  - Can experiment with proxy applications written in DARMA

- Sandia ATDM is using DARMA to inform its technology roadmap in the context of AMT runtime systems
DARMA’s programming model enables runtime-managed, measurement-based load balancing.

The Charm++ load balancer incrementally runs as particles migrate and the work distribution changes.

Using DARMA to inform Sandia’s ATDM technical roadmap