DARMA:
A software stack model for supporting asynchronous, data effects programming

Jeremiah Wilke (PI), Jonathan Lifflander, David S. Hollman, Giulio Borghesi, Aram Markosyan, Robert L. Clay (PM)
DARMA: Distributed, Asynchronous, Resilient Models for Applications

• Provide a programming model target that can express tasks with flexible granularity and their corresponding data effects
  – Assumption: granularity control is the key to performance, data effects are critical to efficient task mapping and synchronization
  – Corollary: *explicit* overdecomposition creates complicated multi-level code, obscures fundamental algorithm; programmer focus should be on “fundamental” unit of work and relationships (data effects) to other tasks
  – Corollary: Amortize cost of overdecomposition through “elastic” tasks that can exploit data parallelism, decrease the ratio of tasks/cores

• Provide a programming model and semantic with race-free execution, but with flexible underlying execution model
  – Assumption: programming focus should be task decomposition, not control structures; execution model should provide safe concurrency; scheduling heuristic must provide non-blocking and resource usage guarantees
Application domains driving DARMA worldview are characterized by dynamic parallelism or complicated synchronization patterns.

*Unknown unknowns*
Dynamic, rapidly changing parallelism. Load balancing not feasible without *dynamic* load balancing across tasks.

- Solid mechanics with multiscale physics
- Particle-in-cell quiescence detection
- Tree search
- AMR with “fast” shockwave
Application domains driving DARMA worldview are characterized by dynamic parallelism or complicated synchronization patterns

**Known unknowns**
Dynamic, but persistent behavior. Semi-static scheduling across phases possible using timers/performance counters

- Particle-in-cell load balancing across patches
- Block-based sparse linear solvers with irregular sparsity
- AMR with “slow” shockwave
Application domains driving DARMA worldview are characterized by dynamic parallelism or complicated synchronization patterns

**Difficult knowns**
Problems known “statically” but flexible granularity needed for data locality (e.g. tiling) or fine-grained synchronization

- Tiled-based linear algebra
- Finite element matrix assembly
- Complex chemistry
DARMA provides data effects programming within C++

- Data effects help *extract concurrency*
  - Maintain sequential, imperative model for expressing algorithms
  - Derive data flow and synchronization from *data usage relationships* between tasks
  - Deterministic-by-default semantics based on sequential task order

- Data effects programming puts focus on *locality* and *granularity*
  - Application-level focus should be on the right task granularity to preserve locality and maximize parallelism
  - Granularity must be balanced against overheads
DARMA design goal #1: Flexible *granularity*

• Granularity (DARMA definition):
  – Task size
  – Task shape
  – Task boundaries

• Choice of granularity is driven by
  – Data locality
  – Coarse-graining (limit synchronization)
  – Pipelining (limit delay of next steps)
  – Loops structured for fine-grained
  – Load balance

• Data locality and fine-grained (SMT-ish) parallelism are major exascale challenges
  – Data movement relatively more expensive
  – Hardware will accelerate *fine-grained* operations
  • GPU, vectors hard enough with branchy code
Coarse-grained parallelism will increasingly be “epiphenomenon” of local task structure

- Data locality is a driving concern for task decomposition
- Traversal order creates DOACROSS task parallelism between tiles
  - Notion of DOACROSS parallelism existing concept from 90’s
  - DOALL + pt-2-pt inter-task synchronization constraints
- Flexible task granularity is a requirement of data locality, e.g. not the fundamental requirement itself
- **Express ”fundamental” work unit targeted at main bottlenecks; let runtime (compiler?) distribute and aggregate**
DARMA Design Goal #1 (Corollary): Manage the complexity of multi-level parallelism in the programming model

- Should each of boxes fit L1? L2? Be a thread? A load balancing unit? An MPI rank?
  - Answer: Yes

- Programming should express fundamental units of work and their composition (as data effects)

Corollary: Manage overheads to keep tasks from being ”artificially” large

- Tasks are shared-memory by default to minimize overhead, but hooks generated at compile-time for “promoting” to distributed memory

- Memoization of data effects as lightweight synchronization graph, exploit persistence whenever possible (as done in MPI)
DARMA Design Goal #2: Flexible, safe asynchronous programming model balancing breadth- and depth-first execution

- Particle-in-cell, tree-traversal problems are both branchy codes that create “holes” in parallel lanes
  - “Breadth-first” loop/tree traversal *creates parallelism*, but *loses locality*
  - “Depth-first” loop/tree traversal *preserves locality*, but *loses parallelism*
DARMA provides data effects programming within C++

```cpp
void dpotrf(Tile& Aii);
void dtrsm(const Tile& Aii, Tile& Aji);
void dsyrk(const Tile& Aji, Tile& Ajj);
void dgemm(const Tile& Aji, const Tile& Aki, Tile& Ajk);

int cholesky(...) {
  ...
  Matrix A(n,n,N);
  ...
  for (int i=0; i < n; ++i){
    dpotrf(A(i,i));
    for (int j=i+1; j < n; ++j){
      dtrsm(A(i,i), A(j,i));
    }
    for (int j=i+1; j < n; ++j){
      dsyrk(A(j,i), A(j,j));
      for (int k=j+1; k < n; ++k){
        dgemm(A(j,i), A(k,i), A(j,k));
      }
    }
  }
  return 0;
}
```
DARMA provides data effects programming within C++

```cpp
struct dpotrf{ operator()(Tile& Aii); };
struct dtrsm{ operator()(const Tile& Aii, Tile& Aji); };
struct dsyrk{ operator()(const Tile& Aji, Tile& Ajj); };
struct dgemm{ operator()(const Tile& Aji, const Tile& Aki, Tile& Ajk); };

int cholesky(...)
{
  ...
  DarmaMatrix A(n,n,N);
  ...
  for (int i=0; i < n; ++i){
    create_work<dpotrf>(A(i,i));
    for (int j=i+1; j < n; ++j){
      create_work<dtrsm>(A(i,i), A(j,i));
    }
    for (int j=i+1; j < n; ++j){
      create_work<dsyrk>(A(j,i), A(j,j));
      for (int k=j+1; k < n; ++k){
        create_work<dgemm>(A(j,i), A(k,i), A(j,k));
      }
    }
  }
  return 0;
}```
Metaprogramming provides data effects directly within standards-compliant C++

- Data effects captured by using special C++ wrapper classes `AccessHandle<T>`
- Converting from data-centric programming to execution-centric backend requires dependency *capture* and dependency *analysis*
- Metaprogramming in headers embeds *capture* in access handles (what data used in task and how?)
- Metaprogramming in headers embeds *analysis* through task creation functions (relation between data used in task A to data used in task B?)

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
DARMA defines “asynchronous” semantics for C++

- DARMA defines variable asynchronous permissions with two parts:
  - Immediate: What’s valid right now?
  - Scheduling: What’s valid in subtask?
- Permissions can be:
  - Modifiable (M)
  - Read-only (R)
  - None (N)
- DARMA semantics dictate how permissions change in closures and continuations
- Permissions are defined at language level (compile-time) regardless of execution order
- Semantics ensure deterministic-by-default (non-racy) code

```c++
void start(AH<int> v){
    ...
    create_work(reads(v), [=]{
        work(v);
    });
    std::cout << v.get_value();
    create_work([=]{
        moreWork(v);
    });
    ...
}
```

1. Immediate = Scheduling = Read-only
2. Immed = Read-only, Sched = Modify
3. Immediate = Scheduling = Modify
4. Immed = None, Sched = Modify
Semantics support flexible execution models in DARMA

- **Depth-first execution**
  - Emphasize locality, local work first
  - Defer continuation, execute subtask
  - Requires context switching

- **Breadth-first execution**
  - Increase parallelism and lookahead
  - Defer subtasks, execute continuation
  - No context switching required
Semantics supports flexibility in execution model and therefore flexibility in backend runtime implementation

- Execution model supports any mix of depth/breadth-first (if architecture supports)
- Semantics allow concurrent and asynchronous:
  - *Execution* (running tasks)
  - *Scheduling* (analyzing DAG)
  - *Creation* (append new tasks)
The programming/execution models are conceptual guides designed to express all available parallelism.

DARMA software stack for community involvement in development, best practices.

Not all runtimes provide the same functionality.

Front End API (Application User)
Translation Layer
Back End API (Specification for Runtime)
The programming/execution models are conceptual guides designed to express all available parallelism.
The programming/execution models are *conceptual* guides designed to express parallelism to a variety of runtimes

DARMA software stack model for engaging apps + runtime teams to develop best practices

Not all runtimes provide the same functionality
Constructs exist for task parallelism or data-parallel collections. Data collections require specifying **granularity**.

- Data collections specify granularity.
- Granularity can be completely decoupled from nodes/processes.
  - Easy overdecomposition.
- "Correct" granularity is a balance of:
  - Maximizing parallelism.
  - Increasing locality.
  - Balancing overheads.

```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)
);
```

**Example Program**

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.
Granularity control is critical to performance in many applications.

Load balance, pipeline parallelism comes from overdecomposition factor (ODF), no need to repartition problem.

Particle-in-cell problem is major example of granularity control dictating performance.
Granularity control is critical to performance in many applications

- Communication pipelining comes from overdecomposition
- Granularity dictated by algorithm
Granularity control is critical to performance in many applications.

### Strong-scaling on Trinity

- **HybridLB**
- **No Load Balancer**
- **Ideal**

### Compute intensity over time

- Load balance flexibility comes from overdecomposition
- Load balancing must balance *quality* of heuristic with *cost* of heuristic
Critical step in lowering overheads of “overdecomposition” approach is overdecomposing *nodes*, not overdecomposing cores

- DARMA scheduler allocates multiple cores to task, configures “environment” of Kokkos execution space
DARMA’s Higher-level Programmatic Approach: Defining a “software stack model” for asynchronous programming

Mapping to a variety of AMT runtime system technologies
Mitigate risks by leveraging existing runtime infrastructure as much as possible, encourage collaboration

Diagram not necessarily to scale!

*Mapping to a variety of AMT runtime system technologies*
Current efforts geared towards reference implementation on MPI + OpenMP for interoperability, baseline example for community

Primary 2018 milestone: MPI+OpenMP interoperability of dynamic, asynchronous DARMA phase/kernel

Mapping to a variety of AMT runtime system technologies
C++ committee activity: async_ptr
(P0971, not yet released)

- We are currently working on a proposal for a class template called async_ptr that has similar semantics to AccessHandle (but less implicit syntax).
- Proposal uses syntax that reflects the analogy to reference borrowing (from Rust), which has similar semantics.

```cpp
doctest
void dpotrf(Tile& Aii);
void dtrsm(const Tile& Aii, Tile& Aji);
void dsyrk(const Tile& Aji, Tile& Ajj);
void dgemm(const Tile& Aji, const Tile& Aki, Tile& Ajk);

int cholesky(...) { ... }
... Matrix<async_ptr<Tile>> A(n,n,N);
... for (int i=0; i < n; ++i){
  A(i,i).async_borrow_value(dpotrf);
  for (int j=i+1; j < n; ++j){
    with_all(as_const(A(i,i)), A(j,i)).async_borrow_value(dtrsm);
  }
  for (int j=i+1; j < n; ++j){
    with_all(as_const(A(j,i)), A(j,j)).async_borrow_value(dsyrk);
    for (int k=j+1; k < n; ++k){
      with_all(as_const(A(j,i)), as_const(A(k,i)), A(j,k))
        .async_borrow_value(dgemm);
    }
  }
}
return 0;
}
C++ committee activity: Executors
(P0443)

- We are actively contributing to the development of the C++ Executor concept and supporting facilities.
- Executors, as a concept, provide constraints on a low-level interface for describing execution patterns in generic code.
- Not intended to be a user-level API (mostly a library-author-level API)

```cpp
priority_scheduler sched;
auto ex = execution::require(sched.executor(), oneway, single);
auto low = execution::require(ex, low_priority);
auto med = execution::require(ex, normal_priority);
auto high = execution::require(ex, high_priority);
execution::executor<oneway_t, single_t, priority> poly_high(high);
...
low.execute([ ]{ std::cout << "11\n"; });
low.execute([ ]{ std::cout << "111\n"; });
med.execute([ ]{ std::cout << "2\n"; });
med.execute([ ]{ std::cout << "22\n"; });
high.execute([ ]{ std::cout << "3\n"; });
high.execute([ ]{ std::cout << "33\n"; });
high.execute([ ]{ std::cout << "333\n"; });
poly_high.execute([ ]{ std::cout << "3333\n"; });
execution::require(ex, priority{-1}).execute([&]{ sched.stop(); });
...
sched.run();
```
Other C++ Committee Activities

- Futures
  - (Proposal number not yet assigned; see github.com/executors/futures)
- span/mdspan, in collaboration with Kokkos
  - P0009, P0546
- Generic access properties for views of memory
  - P0856, P0860, P0900
- Atomic references to arbitrary memory
  - P0019
- Deferred reclamation through hazard pointers and RCU
  - P0566
Tutorial Time!

bit.ly/darma-tutorial

Acknowledgments