This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2018-10688C

Distributed memory futures for compile-time, deterministic-by-default concurrency in distributed C++ applications

Jeremiah J. Wilke
jjwilke@sandia.gov

David S. Hollman
dshollm@sandia.gov

Cannada Lewis
canlewi@sandia.gov

Aram Markosyan
amarkos@sandia.gov

Nicolas Morales
nmmoral@sandia.gov

Sandia National Labs
Scalable Modeling and Analysis
Livermore, CA, 94550

ABSTRACT

Futures are a widely-used abstraction for enabling deferred execution in imperative programs. Deferred execution enqueues tasks rather than explicitly blocking and waiting for them to execute. Many task-based programming models with some form of deferred execution rely on explicit parallelism that is the responsibility of the programmer. Deterministic-by-default (implicitly parallel) models instead use data effects to derive concurrency automatically, alleviating the burden of concurrency management. Both implicitly and explicitly parallel models are particularly challenging for imperative object-oriented programming. Fine-granularity parallelism across member functions or amongst data members may exist, but is often ignored. In this work, we define a general permissions model that leverages the C++ type system and move semantics to define an asynchronous programming model embedded in the C++ type system. Although a default distributed memory semantic is provided, the concurrent semantics are entirely configurable through C++ constexpr integers. Correct use of the defined semantic is verified at compile-time, allowing deterministic-by-default concurrency to be safely added to applications. Here we demonstrate the use of these "extended futures" for distributed memory asynchronous communication and load balancing. An MPI particle-in-cell application is modified with the wrapper class using this task model, with results presented for a Haswell system up to 64 nodes.

CCS Concepts

•**Software and its engineering** → **Parallel programming languages; Object oriented languages;** *Imperative languages; Functional languages; Data flow languages;*

Keywords

task-based runtimes, template metaprogramming, parallel computing, programming models, execution models

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESPM2 '18 November 16, 2018, Dallas, TX, USA

© 2018 ACM. ISBN 978-1-5090-3858-9.

1. INTRODUCTION

Interest has surged recently in asynchronous or task-based programming models. Prominent examples include MPI+X models [3] as well as more comprehensive task-models that completely replace MPI such as Legion [4], HPX [14], or Charm++ [15]. On-node deferred execution programming models can enable reordering or parallel execution of work that would otherwise have run serially. Alternatively, it can allow large tasks graphs to be deferred and enqueued all-at-once on accelerators, for example in CUDA tasking [1], to reduce overheads associated with accelerator execution. In distributed memory, task-based models can provide a clear description of data effects that make execution units migratable, allowing load balancing or work stealing for challenging, dynamic problems.

A prominent design feature of some task-parallel languages is automatic extraction of deterministic-by-default concurrency. In Legion, this is usually termed apparently sequential semantics, emphasizing that the final result will be equivalent to a serial, sequential execution, but with tasks possibly executing in parallel or out-of-order when data effects show that it is safe to do so. Legion expresses these data effects in the form of region requirements — the data region used by a task and the mode (usually read or write) in which it is used. Attempts to embed sequential semantics directly in C++ templates were made with the MetaPASS header library [12]. These principles also feature prominently in deterministic parallel Java [19].

Task models relying on dependency analysis usually obey closure rules, with the child tasks only allowed to operate on data "owned" by the parent task. Parent tasks with, e.g., read-only privileges cannot create child tasks with write privileges. In general, task creation ("forking" in Cilk terms) splits a thread into a subtask (or closure) and a continuation. In explicitly parallel languages like Cilk and X10, the subtask and continuation are always allowed to execute in parallel with no restrictions. The programmer must insert explicit `sync` statements to ensure correct execution. In models like the C++ standard library or HPX, futures can be used to represent data that will eventually become available. These futures, though, do not enforce a sequential semantic or provide a deadlock-free guarantee. Typically in these models, a task created from a future will run whenever the future becomes available, but most of these models also provide a means of manually satisfying a future (usually called a promise), which can lead to dependency cycles.

Deadlock-free, deterministic-by-default models instead enforce conflict-freedom in a way that semantically prevents

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

dependency cycles. In models that use sequential semantics for this purpose, dependency cycles like "A must follow B and B must follow A" are impossible to express because "must follow" is expressed lexicographically. Preserving sequential semantics between subtask closure and continuation may occur through blocking. In Legion, mapping logical regions may block the parent task (continuation) until the child tasks have executed. Another approach that leads to fewer constraints on the execution model is to restrict operations in the continuation in the programming model directly, allowing a subtask and continuation to execute in parallel. If a subtask reading a piece of data might run in parallel with the continuation, the continuation must avoid modifying the data to preserve the sequential semantic. In either case, current systems like Legion or MetaPASS only perform error checking for invalid task creation and dependency analysis at runtime. Regent [22] is one approach by the Legion developers to provide language support. Herein, we present another approach that could be applied to any programming model with an effect system that can be expressed in terms of a state machine — which is frequently the case for sequential semantic models like Legion.

Here we present a general deterministic-by-default futures model for distributed object collections, primarily exploring its use in load balancing. These lightweight C++ abstractions can embed *deferred execution* in an existing MPI C++ code. Deferred execution can provide resource virtualization, separating the logical handle for the data (similar to a future) and and its physical location. Resource virtualization through deferred execution enables load balancing, allowing data to by freely migrated since it is no longer assumed to always exist at the same location in the same address space. We introduce how these load balancing abstractions can be incrementally added to a larger MPI code base through "MPI futures" that interoperate between asynchronous kernels and MPI regions. We modify an MPI particle-in-cell (PIC) application to leverage asynchronous tasking and load balancing within one phase while interoperating with an MPI library in another phase. Along with load balancing results, we show how the customizable semantics can statically express more complex concurrency for distributed objects in the PIC code. While sequential semantics often depend on read or write usages, generalized concurrency properties can be particularly powerful for object-oriented programming. Fine-grained concurrency for objects — e.g., member functions of the same object executing in parallel or operations simultaneously executing on different data members — is often present but difficult to express to runtime systems.

The abstractions here were developed as part of the DARMA project (Distributed, Asynchronous, Resilient Models for Applications). DARMA itself is a general exploration of programming and execution model abstractions, largely within the context of C++, that are useful for dynamic or irregular applications. Here we introduce one of the DARMA libraries — the DARMA futures header library and associated backend runtime concept.

2. RELATED WORK

As previously discussed, Legion is a popular data effects models that enforces sequential semantics through read-/write access requests on so-called logical regions [4]. However, logical region substructure is specified in terms of Legion-specific index and field spaces instead of C++ types. Regent is a Lua-based higher-level language that generates code for the underlying Legion C++ runtime, which reduces much of the verbosity of the Legion C++ interface [22]. The state machine introduced herein bears a resemblance to flow-sensitive type systems, particularly the type qualifiers approach of Foster, et al [10]. The reference borrowing semantics in the Rust programming language [16] also implements a statically checked type system where references lose functionality when they are used.

As mentioned above, fine-grained concurrency within a C++ object can be difficult to express since the class as a whole "aliases" individual members. In Charm++, the fundamental units of concurrency (chares) are usally a "monolithic" whole, only allowing one operation (entry method) to execute at a time regardless of the parallelism that would otherwise be available based on data effects [15]. Legion defines a comprehensive data model with index and field spaces which can understand hierarchical parallelism, exploiting finer-granularity parallelism when available. However, this requires implementing all data structures in the Legion model. The type system introduced here tries to express general concurrency relationships, with concurrency added incrementally as needed.

OpenMP [18, 2] and OmpSs [6] are pragma-based tasking frameworks, with the possibility of defining in/out dependencies for each task. HPX [14, 11] provides futures for controlling program execution. While HPX implements type-safe task objects and is amenable to data-flow algorithms, it does not provide deterministic-by-default semantics.

Explicitly parallel tasking frameworks provide async-finish or fork-join keywords for creating tasks, including Cilk [5] and X10 [8] Another example, OCR [17], is not explicitly parallel, constructing tasks with dependencies rather than directly spawning subtasks. However, the dependencies must be explicitly defined rather than being derived from an imperative model with sequential semantics. UPC++ [23] and Chapel [7] also provide tasking.

Domain-specific libraries can also rely on implicit parallelism. Uintah implements patch-based adaptive mesh refinement [21]. A sequential "patch-specific" code is implemented and the runtime system automatically derives task parallelism within a patch and implements data parallelism across patches. Listzt defines operations on vertices, edges, faces, and cells, which the Liszt framework then compiles into a task-based execution [9]. Implicit parallelism has also been pursued through auto-parallel compilers [20].

In the current work, load balancing is based on shuffling objects in a distributed collection between processes. This strategy requires overdecomposition, with more objects than processes. Load balancing based on overdecomposition is a major feature of Charm++ [15], particularly in the AMPI model which directly encapsulates multiple MPI ranks within a single process [13].

3. EXTENDED FUTURES MODEL FOR DEFERRED EXECUTION

3.1 Closures and Continuations

We first consider our concurrency model for a simple on-node example. Consider the code

```
void update_a(data& a, const data& b);
void parent_task(data a,  data b){
  preamble(a,b);
  async child_task(a);
  update_a(a, b);
}
```

which demonstrates the challenges of deterministic-by-default concurrency. Here a parent task begins executing the function `preamble()`. Creating a child task that operates on `data a` forces two possibilities if a sequential semantic is to be preserved. The child task could execute first, blocking the parent continuation from starting. Alternatively, the child and parent could execute in parallel, but the parent task continuation would not be allowed to modify `data a` (but could perform any operation on `data b`). The first option

restricts potential parallelism and freedom in the execution model. The second option requires a difficult to enforce contract between the programmer and the runtime system not to modify `data a` in the continuation.

Consider instead the code that avoids a data race on 'a'.

```
void update_a(data& a, const data& b);
void parent_task(data a, data b){
  preamble(a,b);
  auto a_new = async child_task(std::move(a));
  update_a(a_new, b);
};
```

Can we use the auto type of variable `a_new` to enforce a particular semantic - a semantic that allows non-blocking child tasks but is still deterministic-by-default?

Data in imperative, asynchronous programs implicitly carry the notion of *immediate* and *scheduling* permissions. What operations are allowed immediately at the current line of code? What operations would be allowed in an asynchronously scheduled child task? Standard C++ types are all-or-nothing. A future has no immediate permissions (no operations permitted on the underlying data), but full scheduling permissions. In a continuation-passing model like HPX, an asynchronous subtask attached to the future would have no restrictions. A C++ class (non-future) would have full immediate permissions, but no scheduling permissions since references in a forked thread have no guarantee of being valid in the future.

We propose the future-like class template `async_ref`:

```
template <class T, int Imm, int Sched>
struct async_ref {
  ...
};
```

Like a future, the class wraps an underlying type `T`. However, a compile-time integer constant allows defining immediate and scheduling states — and correspondingly the allowed operations in the closure and continuation.

The semantics of the program are essentially dictated by *losing permissions* in the continuation. Subtasks (closures) request permissions (e.g., read/write), which requires those permissions to be removed from the reference in the parent continuation. Beyond these fixed semantics for closures and continuations, the framework allows different deterministic-by-default semantics to be defined by specifying combinations that the state machine should treat as conflicts.

```
template <class T, int Trait> struct Conflicts;
template <class T> Conflicts<T,ReadOnly> {
 static constexpr int value = Write;
};
template <class T> struct Conflicts<T,Write> {
 static constexpr int value = Write | ReadOnly;
};
```

These `Conflicts` structs define which operations (permissions modes) are safe to execute in parallel. Immediate permissions in the parent continuation cannot conflict with permissions in the subtask closure. This defines a permissions "state machine", with continuations gradually losing immediate permissions as they create subtasks. This state machine can be implemented (at compile time):

```
static constexpr int imm_continuation =
  imm_parent & ~Conflicts<T,imm_closure>::value;
```

Figure 1 shows this progression of permissions.

3.2 Load Balancing

The DARMA futures model is strongly influenced by the Legion concept of logical regions [1]. In DARMA futures we replace `async` with a function `create_work`:

```
auto a_updated = create_work<Task>(std::move(a));
```

[1]In fact, future work is planned to implement exactly Legion semantics in the compile-time state machine

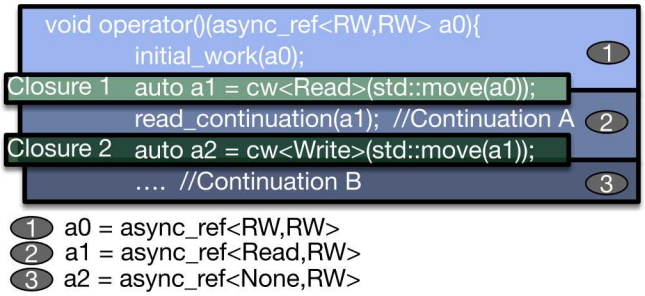

Figure 1: Illustration of immediate and scheduling permissions for async_ref objects in the continuations as subtasks (closures) are created.

A logical connection is created here between `a` and `a_updated`. `a` is expired using a C++ move, which causes a loss of immediate permissions in the continuation. `a_updated` now represents the logical data. Load balancing fundamentally depends on resource virtualization, abstracting the physical location of logical variables. The immediate permissions define where and when load balancing can occur. If a variable holds immediate permissions, e.g. `Read` or `Write`, its physical location cannot change since active tasks may operate on the data. If a variable has immediate permissions `None`, the only valid operations in the active task are *scheduling* future tasks. The runtime system is therefore free to migrate these tasks to load balance.

3.3 Asynchronous Attorneys

The attorney pattern in C++ can control access to private interfaces. Here we extend the notion to privatizing certain member functions when immediate permissions are insufficient. Consider the following:

```
template <int Imm, int Sched>
struct darma::attorney<MyType,Imm,Sched> {
  DARMA_ENABLE(stateNeeded, myMemberFxn);
};
```

where the enable macro is

```
#define DARMA_ENABLE(state, fxn) \
  template <typename = std::enable_if_t< \
    (state | Imm) != 0>, class... Args> \
  auto fxn(Args&&... args){ \
    return get()->fxn(std::forward<Args>(args)...); \
  }
```

This allows the arbitrary access properties of member functions to be explicitly defined, with the constraint only enabling them to be called with the appropriate permissions. If the immediate permissions are insufficient in the given context, a compiler error occurs.

4. DISTRIBUTED COLLECTIONS

The idea of expressing concurrency properties of a C++ object as a state machine in the type system can be extended to collections of objects for distributed memory. Throughout the examples we consider a distributed collection of `Patch` objects that could, e.g., represent a distributed mesh.

```
auto ctx = make_context(MPI_COMM_WORLD,argc,argv);
int size = ... ;
auto coll = ctx->make_collection<Patch>(size);
```

Here a distributed context object is created for a given MPI communicator. For the `make_collection` function, the returned type is an `async_ref` object that provides a handle for scheduling tasks across the collection. Globally across the parallel job, `size` number of instances of a `Patch` object are instantiated. However, there is no guarantee where or when the instantiation occurs, which means `coll` has imme-

diate permissions `None`. To create tasks, we have:

```
auto new_coll = ctx->conc_work<Task>(std::move(coll));

struct Task {
  void operator()(Context* ctx, int index,
        async_ref<Patch> patch){
    patch->doWork();
  }
};
```

The concurrent work function creates an instance of the task for each Patch in the collection, with an extra argument provided to indicate which index is currently active. The concurrent work function is an explicitly parallel operation and assumes that functors operating on different indices can safely run in parallel.

4.1 Accessors and Send/Recv Operations

The DARMA futures framework allows concurrent tasks (indices) to "send" and "receive", similar to MPI. Much like task functors, send/receive operations are defined through an accessor struct that defines the data to be exchanged.

```
struct Ghost {
template <class Archive>
static void pack(Patch& p, Archive& ar, int nbr){
  ar | p.in_ghosts[nbr];
}

template <class Archive>
static void unpack(Patch& p, int nbr, Archive& ar){
  ar | p.in_ghosts[nbr];
}
};
```

Here the archive implements serialization, much like the Boost serialization library. Inside a task body, this allows:

```
int dest = ...;
auto send_patch =
  ctx->send<Ghost>(dest, std::move(patch));
```

The `async_ref` futures here have a strong connection to Legion logical regions. The variable `patch` represents an instance (state) of a *logical* piece of data in the system. When the data is sent, that instance is *expired* using the move operation, indicating the previous state is no longer valid. The returned variable `send_patch` represents the same *logical* piece of data, but after the send operation has completed. The deterministic-by-default semantics of send/recv can be defined for the accessor through constexpr integers. Send and receive on different ghost cells do not conflict, but would conflict with read/write tasks.

4.2 MPI interoperability

Collections can exist either in "MPI mode" or "DARMA mode." To create an MPI collection, the library provides:

```
auto mpi_patches = ctx->make_mpi<Patch>(size);
```

In contrast to a DARMA collection, the MPI collection can be modified without concurrent work functions. We expect the dominant use case to be overdecomposed collections (more objects than MPI ranks) with data collections initialized in MPI mode.

```
for (int local: myIndices){
  mpi_patches->emplaceLocal(local, ...);
}
```

MPI collections can be converted into DARMA collections:

```
auto darma_patches =
  ctx->from_mpi<MpiAccessor>(std::move(mpi_patches));
```

The MPI accessor defines how data is moved to/from DARMA kernels. The accessor can define a subset of the patch to move between modes. This movement of subsets is demonstrated in the particle-in-cell application. DARMA collections can be transitioned back to MPI.

```
auto mpi_patches =
  ctx->to_mpi<MpiAccessor>(std::move(darma_patches));
```

4.3 Load Balancing

The current DARMA library supplies a `Phase` object that can collect profiling information. Rather than the normal concurrent work function, tasks are created as:

```
Phase ph = ctx->make_phase<int>(size);
...
auto next_coll
   = ctx->phase_work<Task>(ph, std::move(coll));
```

Phases, unlike collections, are not expired and carry persistent information. For load balancing, the DARMA futures library provides a rebalance function:

```
auto balanced
   = ctx->rebalance<LB>(phase, std::move(patches));
```

Future concurrent work (or phase work) calls will then use the new distribution. The load balancer can either move the entire patch object or be supplied with a struct defining a subset, as in send/recv accessors. The struct `LB` is similar to the accessor for send/recv, moving either the whole class or a subset. Subset load balancing is used in the particle-in-cell application below (Figure 3)

4.4 Collectives

Collective operations can be executed over collections, similar to MPI collectives. Generally, collectives either convert a piece of data (`async_ref`) into a collection (e.g., broadcast or scatter) or convert a collection into a single piece of data (e.g., gather, reduce). For reductions, the operation to perform must be defined via a functor.

```
auto [residual, patches_red]
  = ctx->reduce<Residual>(std::move(patches));
```

The function returns two `async_ref` objects, representing a dependence (`residual` is produced) and anti-dependence (`patches_red` is read). The functor `Residual` is:

```
struct Residual {
  void operator()(const Patch& in, double& out){
    out += in.residual();
  }
  static double identity(){ return 0. }
  static int mpiSize(){ return 1; }
  static MPI_Datatype mpiType(){ return MPI_DOUBLE; }
  static MPI_Op mpiOp(){ return MPI_SUM; }
};
```

We can directly reduce across patches, extracting a single field. The reduction requires an identity operator to initialize. The prototype backend in this work supports mapping directly onto MPI collectives if hint functions are provided, leveraging optimized MPI implementations

5. BACKEND RUNTIME

5.1 Frontend Interface

In general, a backend runtime should be able to represent execution through a computational directed acyclic graph (CDAG), capturing tasks and their dependencies as a graph rather than just a sequential execution stream. The DARMA futures library provides a frontend header library, which implements the semantics and task creation functions. The frontend is a class template, relying on a backend class parameter that meets a required concept.

```
template <class Backend>
class Frontend : public Backend {
```

We illustrate the structure via the `create_work` function:

```
template <class Functor, class... Args>
auto create_work(Args&&... args){
  auto ret = mpl_semantics<Functor,Args...>(args...);
  auto frontend_task = make_task<
    Functor,Args...>(std::forward<Args>(args)...);
```

```
  auto* backend_task
    = Backend::make_task(std::move(frontend_task));
  mpl_register_dependencies(backend_task);
  Backend::register_task(backend_task);
  return ret;
```

Here the frontend metaprogramming library function `mpl_semantics` introspects the functor to determine what permissions are allowed in the continuation, creating the return type. The frontend task provides a run function that can be invoked by the backend that applies the task function to the argument tuple.

```
void run(){
  mpl_call(Functor(),
    std::make_index_sequence<sizeof(Args)...>{});
}
```

The function `mpl_register_dependencies` causes each `async_ref` dependency to be registered with the task. The backend must provide a function of the form:

```
template <class T, int ImmRequired, int immCurrent,
    int SchedRequired, int SchedCurrent>
void registerDep(Task* task, async_ref_base<T>&);
```

The backend is free to implement task graphs and scheduling however it chooses. Task preconditions are expressed entirely through the immediate permissions. When a task is registered, certain pieces of data may lack the required permissions, forcing them to be put into a pending queue. When tasks finish, the destructor of the frontend task automatically releases the permissions for all of its data.

```
template <class T, int Imm, int Sched>
void releaseDependency(async_ref_base<T>&);
```

This notifies the backend runtime that permissions are available for the piece of data, potentially allowing tasks to be moved from a pending to a ready queue.

5.2 Overdecomposition and Load Balancing

For distributed memory, collections are primarily intended to be combined with overdecomposition. In contrast to most MPI problem decompositions, which create a single patch or partition per MPI rank, overdecomposition creates multiple partitions within a single process. To illustrate overdecomposition for load balancing, we present the basic load balancing algorithm used in the current work (Figure 2). Each MPI rank holds multiple tasks, each with a given weight (e.g. timers). The algorithm exchanges tasks between "complementary" pairs. A parallel sort ranks MPI processes from most to least work. Processors with the most work (rank i) either give away or trade larger tasks to a partner rank (N - i). This process is repeated until a sufficiently balanced distribution is found. Rather than implement a parallel sort for this work, the keys passed to `MPI_Comm_split` are used to sort MPI ranks and assign trading partners.

6. PARTICLE-IN-CELL MINI-APP

6.1 Overview

The example considered here is a particle-in-cell (PIC) mini-app. PIC poses serious load balancing challenges since it combines an imbalanced, dynamic phase that operates primarily on particle data with a well-balanced, static field solve phase that operates on mesh data. An optimal, balanced distribution of particles may not match an optimal, balanced distribution of the mesh.

A baseline MPI mini-app (9K lines) was developed first, with no load-balancing. PIC is a very challenging problem to implement with multi-threading since particles can migrate dynamically. Filling send buffers or backfilling the local particle storage requires careful implementation to be thread-safe. DARMA futures aim to provide a productive programming model that easily and transparently enables task parallelism and load balancing for such an existing code. As much as possible, the original (thread-parallel) code should be preserved. Here only 500 lines of "wrapper" DARMA code was implemented in a single source file to add load balancing and asynchronous communication. The multithreaded particle move kernels were preserved without modification from the parent MPI code. Both the DARMA runtime and the PIC mini-app can be obtained on a feature branch of the DARMA github repository: https://github.com/DARMA-tasking/darma-futures

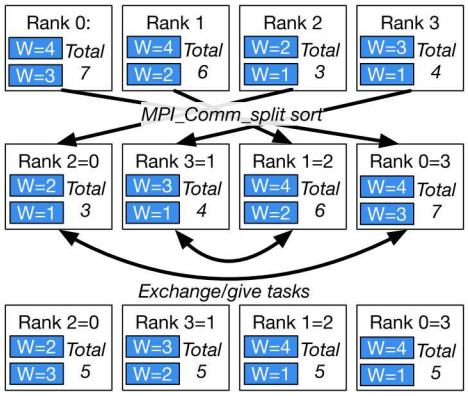

Figure 2: Overview of load balancing algorithm. Ranks exchange tasks of weight (W) to balance distribution.

In the current work, particles move through an unstructured mesh and interact only with an electric field, which is updated via a Poisson solve (rather than a full Maxwell solve). In general, particles move through space, accelerating through electric and magnetic fields discretized via cells in the mesh. Residual charge densities and currents are updated based on the particle movement. After each timestep, electric and magnetic fields are updated through a solver. The solver phase is easy to decompose into a balanced work distribution in standard MPI code. The particle move phase is highly irregular with dynamic particle trajectories and is implemented as a DARMA kernel.

6.2 Data Distribution between DARMA/MPI

Certain data structures are required only for the MPI solver, only for the DARMA particle move, or are needed in both the DARMA and MPI phases. These data structures are summarized in Figure 3. The data movement pattern from MPI to DARMA and back is not symmetric. The residual densities generated by the move kernel must be passed to the MPI solver, but need not be sent back when the DARMA kernel resumes. Similarly, the residual fields must be sent from the MPI solver to the DARMA move kernel, but need not be sent back. The particles, which consume most of the memory and computational time (see below) are not directly required by the solver.

6.3 PIC Iterations

The PIC algorithm is shown in Figure 4. Patches are initialized as an MPI collection. The code then alternates between moving particles as an asynchronous kernel and solving updated fields in MPI mode.

Here load balancing is triggered at regular intervals. Every time an asynchronous (deferred) operation is performed on the collection, the value must be expired through C++ move semantics. Type changes express the loss of immediate permissions. One example of a conditional is shown since the load balancer may or may not run. If the load balancer did not run, a no-op must occur that keeps the return type consistent (i.e. loses permissions).

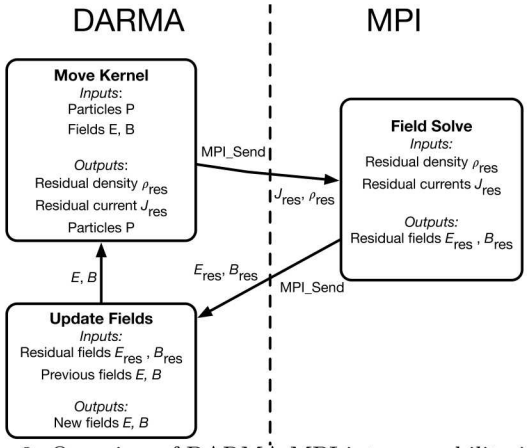

Figure 3: Overview of DARMA-MPI interoperability in the PIC application. The solver kernel uses the initial MPI data distribution. DARMA dynamically migrates data to achieve load balance in the move kernel. When interoperating, only a subset of data must be transferred from the DARMA location to the MPI location.

```
auto ctx = make_context(MPI_COMM_WORLD,argc,argv);
int size; MPI_Comm_size(MPI_COMM_WORLD, &size);
int overdec = N;
auto mpi_pic = ctx->make_mpi<PIC>(size*overdec);
auto phase = ctx->make_phase(mpi_pic);
for (int iter=0; iter < nIter; ++iter){
 solveFields(mpi_pic); //not asynchronous
 auto darma_pic = ctx->from_mpi<MpiGet>(move(mpi_pic));
 auto darma_upd = ctx->phase_work<StartMove>(
   phase, iter, std::move(darma_pic));
 auto darma_balanced = iter % lb_interval == 0
   ? ctx->rebalance<LB>(phase, move(darma_upd));
   : ctx->no_op<Rebalance>(move(darma_upd));
 mpi_pic = ctx->to_mpi(move(darma_balanced));
}
```

Figure 4: Particle-in-cell Code

Although not explicitly shown, the functor `StartMove` is a very thin wrapper around the original MPI code. The `MPI_Isend` and `MPI_Irecv` operations are replaced by send/recv operations from Section 4.1 and MPI collectives are similarly replaced. The math kernels implemented in the `PIC` object are exactly reused. In this basic usage of the DARMA futures library, we have easily added *implicit* overdecomposition to the application. In the original MPI code, each MPI rank assumes that particles and mesh are partitioned across distributed `PIC` objects, with one `PIC` object per rank. Overdecomposition and load balancing could have been added *explicitly* to the MPI code, but this would have mixed the physics algorithm with complicated bookkeeping (and in an ad hoc manner specific to the PIC application). Even though the DARMA version overdecomposes into multiple `PIC` objects, it maintains a single-level programming model which cleanly hides the bookkeeping of overdecomposition and load balancing with a safe compile-time semantic. The original MPI code can also be "recovered" from the DARMA code by using an overdecomposition factor of 1 and choosing a different backend. Consider:

```
auto ctx = make_context(MPI_COMM_WORLD,argc,argv);
```

The type of the context object can be selected at compile-time and actually change the semantics of the backend. A "bulk-synchronous" context could be chosen, which turns off deferred execution and re-creates MPI execution.

7. RESULTS

All results were collected on the Haswell partition of the Mutrino platform at Los Alamos National Laboratory. We wish to understand the performance benefits for an unbalanced problem. The critical parameter is the overdecomposition factor. More work units gives more load balancing flexibility but increases runtime overhead. Here we consider the MPI version without load balancing as the baseline.

The unbalanced problem simulates a large emission of charged particles from a 2D plate. A high particle density exists in a small region, which gradually migrates outward to create a uniform density. Figure 5 shows the effect of overdecomposition on improving the load balance for the move kernel. Note that too little overdecomposition does not grant sufficient flexibility but too much overdecomposition incurs performance overhead.

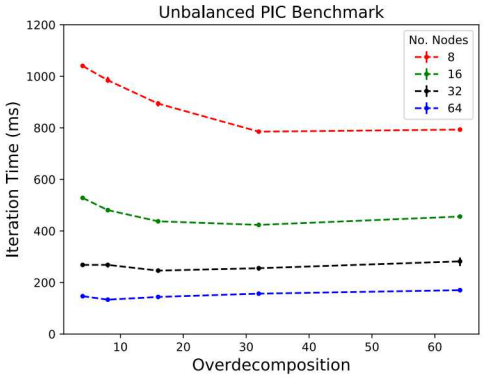

Figure 5: Performance of unbalanced PIC problem with load balancing and changing overdecomposition. All results are collected on the Haswell partition of the Mutrino platform.

Figure 6 shows time to execute the complete iteration, the solver portion, and the move kernel with and without load balancing assuming an overdecomposition factor of 32. In general, the MPI handoff is only a small portion of the overall time. At the largest scales, DARMA load-balancing improves performance by 3–4x relative to the baseline MPI code without load balancing. The load balancer has two phases: computing the balanced distribution and reshuffling the data after rebalancing. Reshuffling the data for the Cray XC30 system here was essentially as costly as an entire timestep, but showed good strong scaling. Computing the balanced distributed was inexpensive up to 32 nodes, but took a significant fraction of a timestep for 64 nodes. This load balance computation is then likely not scalable for larger node counts. The load balancing computation relies on an efficient sorting of keys within the MPI_Comm_split collective, and the current work uses the default Cray MPICH implementation. Scaling out would require more efficient parallel sorting of rank weights.

A theoretically optimal performance improvement can be estimated by DARMA by comparing the maximum task size to the average task size in the system. Here the load balancer is based on timers. These results are shown in Figure 7. The DARMA load balancer is generally able to compute a work distribution within 5–10% of a theoretically optimal distribution. The observed speedups, though, are quite different from that expected based on the load balance computation. DARMA achieves performance within 25% of optimal for most node counts. Further analysis at the largest node counts is needed to identify discrepancies between the load balancer prediction and the observed speedups.

8. FUTURE WORK: CONCURRENT SEMANTICS FOR PIC

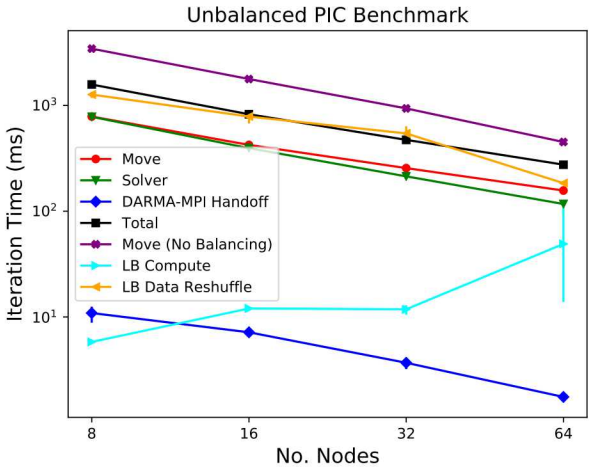

Figure 6: Performance of PIC with load balancing with overdecomposition 32 relative to MPI baseline without load balancing. Scaling of the individual phases (move kernel, solver, DARMA-MPI handoff) are shown separately.

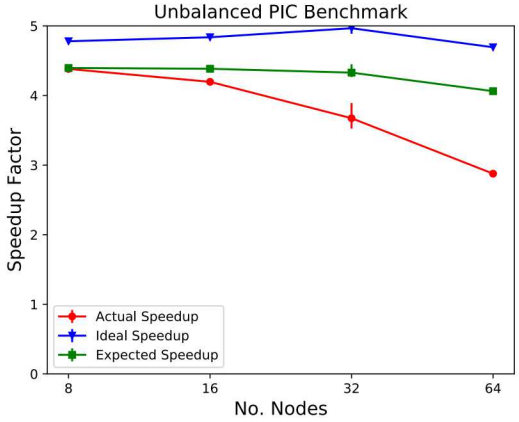

Figure 7: Performance of PIC with load balancing with overdecomposition 32 relative to a theoretically optimal load balanced distribution. Speedup is computed relative to MPI baseline without load balancing. All results are from the Haswell partition of Mutrino.

The original MPI particle-in-cell application implemented the majority of operations through a class `PIC`. The concurrency specification for `PIC` has special semantics. The DARMA futures library reserves integer bits for builtin states, largely modeled on the apparently sequential semantics of Legion: ReadOnly, Write, DisjointSend, DisjointRecv, and a special Atomic flag. The atomic flag indicates that multiple tasks may hold the same set of immediate permissions on a piece of data, but only one may execute at a time. For PIC, however, we add a special state `Moving`, illustrated below. The DARMA PIC attorney is:

```
DARMA_ATTORNEY(PIC) {
DARMA_ENABLE(ReadOnly | Moving, getNeighbors)
DARMA_ENABLE(Moving, getMigrants)
DARMA_ENABLE(Moving | Atomic, moveMigrants)
DARMA_ENABLE(Moving | Atomic, moveLocal)
```

The thread-parallel implementation of the parent MPI code required two functions: `StartMove` and `MoveMigrants`. `StartMove` moves all local particles, creating "holes" in the particle list as they migrate to other patches. `MoveMigrants` moves incoming particles and backfills the particle list in a thread-safe manner. The thread-level data parallelism in the MPI+X code is directly translated to a "DARMA+X" code. `StartMove` from Section 6.3 could be written:

Variable	Immediate	Scheduling
`coll`	Moving	Moving
`my_pic`	Moving	None
`coll_updated`	None	Moving

Table 1: Permissions type for PIC code listing

```
struct StartMove {
using async_pic=async_collection<
  PIC,
  Moving | Atomic, //immediate
  Moving | Atomic //scheduling
>;
void operator()(Ctx* ctx, int iter, async_pic coll){
 auto [my_pic, coll_updated] =
   ctx->getLocal(move(coll));
 my_pic.moveLocal();
 for (int nbr : my_pic.getNeighbors()){
  if (my_pic.numMigrants(nbr)){
   coll_updated = ctx->send_work<MoveMigrants>(
     nbr, move(coll_updated), iter);
  }
 }
}
```

The immediate and scheduling permissions of each variable is shown in Table 1. The type system here handles a special case of aliasing. We wish to operate on a local patch in the collection, but we now have two variables (`my_pic`, `coll_updated`) that refer to the same data. To handle aliasing, the `getLocal` function splits permissions. `my_pic` has only immediate permissions, and is not allowed to be used in creating more tasks. `coll_updated` no longer has immediate permissions. Any future tasks modifying a local patch must be scheduled from the collection. In contrast to a basic send/recv operation, we have a special function `send_work` that not only sends data but causes a task to be enqueued at the destination on arrival.

The struct `MoveMigrants` shows the special concurrency (and semantics) of PIC. In `MoveMigrants`, the collection has `Moving` permissions. In addition, the permissions are atomic, only allowing one task at a time but allowing any particle move to happen in any order.

```
struct MoveMigrants {
using async_pic=async_collection<
  PIC,
  Moving | Atomic, //immediate
  Moving | Atomic //scheduling
>;
void operator()(Ctx* ctx, int from,
        async_pic coll, int iter){
 auto [my_pic, coll_updated] =
   ctx->getLocal(move(coll));
 my_pic.moveMigrants(from);
 for (int nbr : my_pic.getNeighbors()){
   coll_updated = ctx->send_work<MoveMigrants>(
     nbr, move(coll_updated), iter);
 }
}
void pack(PIC& my_pic, Archive& ar, int nbr){
  ar | my_pic.getMigrants(nbr);
}
};
```

Critical here is *termination detection*. Consider the code in Figure 4. The deterministic-by-default semantics require that the `StartMove` task and its child tasks relinquish permissions to `darma_pic` before the load balancer can use that data (via the handle named `darma_upd`). The collection holds scheduling permissions that allow it to schedule tasks *at* other indices in the collection. In contrast to the more common semantic of read/write privileges and strictly hierarchical tasks, a special concurrent semantic is expressed via the `Moving` permissions. In a strictly hierarchical model, there is no requirement to run distributed termination detection. A parent task only needs to check for completion of

its own child tasks. Holding scheduling permissions to other indices in the collection forces the runtime system to detect quiescence, with quiescence occurring when all tasks relinquish scheduling permissions across all indices. Quiescence (termination detection) is not a "first-class" construct in the programming model, being only implicit in the permissions model. Only the backend is aware of the quiescence requirement and is free to apply whatever termination detection algorithm is most suitable and efficient.

9. CONCLUSIONS

Here we have introduced the DARMA futures library, an extended futures model in C++ providing deterministic-by-default semantics via compile-time states. The semantics can be extendible and type-dependent, defined via compile-time constants. The library is primarily designed to enable concurrency for distributed collections of C++ objects. In particular, the C++ object collections are designed to enable overdecomposition for an existing MPI code, providing load balancing and asynchronous tasking as a lightweight (compile-safe) template wrapper. We have applied the prototype header library and reference runtime implementation to an MPI+X particle-in-cell mini-app, providing 3-4x performance improvements over the unbalanced MPI version.

10. ACKNOWLEDGMENTS

This work was funded by Sandia National Laboratories, which is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's (DOE) National Nuclear Security Administration (NNSA) under contract DE-NA-0003525. Addition funding came from the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC.

11. REFERENCES

[1] HiHat and Graphs. https://hihat-wiki.modelado.org/images/a/a1/2018.06.19_HiHAT_and_graphs.pdf.

[2] E. Ayguadè et al. A Proposal for Task Parallelism in OpenMP. In *A Practical Programming Model for the Multi-Core Era*. Springer Berlin Heidelberg, 2008.

[3] R. F. Barrett et al. Toward an evolutionary task parallel integrated MPI + X programming model. In *PMAM '15: Programming Models and Applications for Multicores and Manycores*, pages 30–39, 2015.

[4] M. Bauer et al. Legion: expressing locality and independence with logical regions. In *SC '12: High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[5] R. D. Blumofe et al. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Notices*, 30:207–216, 1995.

[6] J. Bueno et al. Productive Programming of GPU Clusters with OmpSs. In *IPDPS: 26th International Parallel & Distributed Processing Symposium*, pages 557–568, 2012.

[7] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[8] P. Charles et al. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA 2005: 20th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 519–538, 2005.

[9] Z. DeVito et al. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *SC '11: High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[10] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDIâĂŹ02*, 2002.

[11] G. R. Gao et al. ParalleX: A Study of A New Parallel Computation Model. In *IPDPS 2007: 21st International Parallel and Distributed Processing Symposium*, pages 1–6, 2007.

[12] D. S. Hollman et al. Metaprogramming-enabled Parallel Execution of Apparently Sequential C++ Code. In *Extreme Scale Programming Models and Middleware*, ESPM2, 2016.

[13] C. Huang et al. Performance Evaluation of Adaptive MPI. PPoPP '06, 2006.

[14] H. Kaiser et al. HPX: A Task Based Programming Model in a Global Address Space. In *International Conference on Partitioned Global Address Space Programming Models*, pages 1–11, 2014.

[15] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA 1993: Object-Oriented Programming Systems, Languages, and Applications*, pages 91–108, 1993.

[16] N. D. Matsakis and F. S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[17] T. Mattson and R. Cledat. OCR: The Open Community Runtime Interface v1.1.0.

[18] S. L. Olivier et al. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *Int. J. High Perform. Comput. Appl.*, 26:110–124, 2012.

[19] J. Robert et al. A type and effect system for deterministic parallel Java. *SIGPLAN Not.*, 44:97–116, 2009.

[20] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPoPP '99: Principles and practice of parallel programming*, pages 72–83, 1999.

[21] J. Schmidt et al. Large Scale Parallel Solution of Incompressible Flow Problems Using Uintah and Hypre. In *CCGrid '13: Cluster, Cloud and Grid Computing*, pages 458–465, 2013.

[22] E. Slaughter et al. Regent: a high-productivity programming language for HPC with logical regions. In *SC '15: High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[23] Y. Zheng et al. UPC++: A PGAS Extension for C++. In *IPDPS 2014: International Parallel and Distributed Processing Symposium*, pages 1105–1114, May 2014.