Advances in Charm-based Languages (II)

JUSTIN J. SZADAY (SZADAY2@ILLINOIS.EDU)
PHD STUDENT, PPL
UNIVERSITY OF ILLINOIS AT URBANA CHAMPAIGN

Outline

1. Context
2. Our Approach
3. Enhancements
4. Results
5. Future Work
6. Conclusions

Why DSLs?

Hardware complexity continues to rise, posing challenges in:

- Programming heterogenous systems
- Updating specialized software for new hardware
- Choosing the best hardware configuration

In response to these challenges, we need:

- Better tools (e.g., auto-tuners) for navigating problem spaces
- Lift the abstraction level away from hardware complexities
- Languages better suited for HPC than C/C++ and Fortran

Domain-specific languages (DSLs) can help address these needs

DSLs in HPC

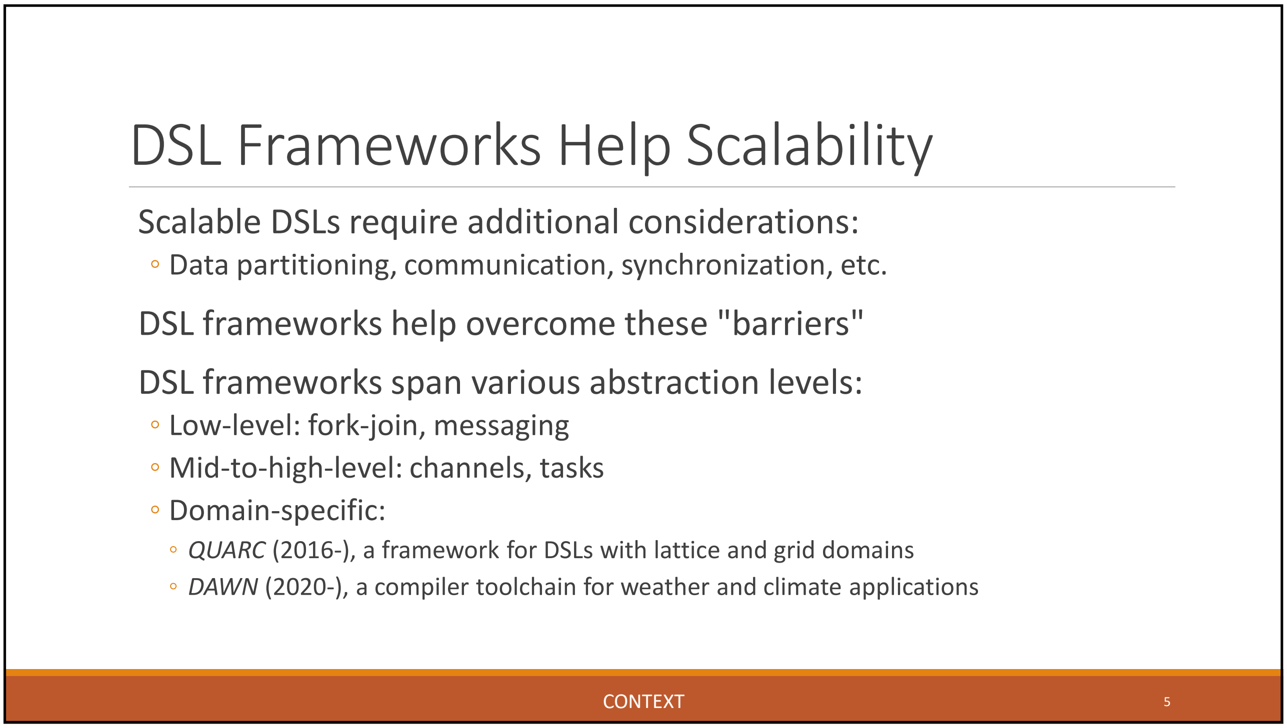
DSLs integrate domain knowledge to enhance:

- *Productivity* – direct access to common abstractions
- *Efficiency* – smaller search space that tools "know" how to navigate

Research in HPC supports distributed DSLs:

◦ *PyOP2* (2012-)	mesh-based codes, embedded in Python
◦ *GridTools* (2016-)	stencil computations, embedded in C++
◦ *OpenPME* (2017-)	particle and mesh methods, standalone

However, DSLs that scale beyond a single node are rare!



DSL Frameworks Help Scalability

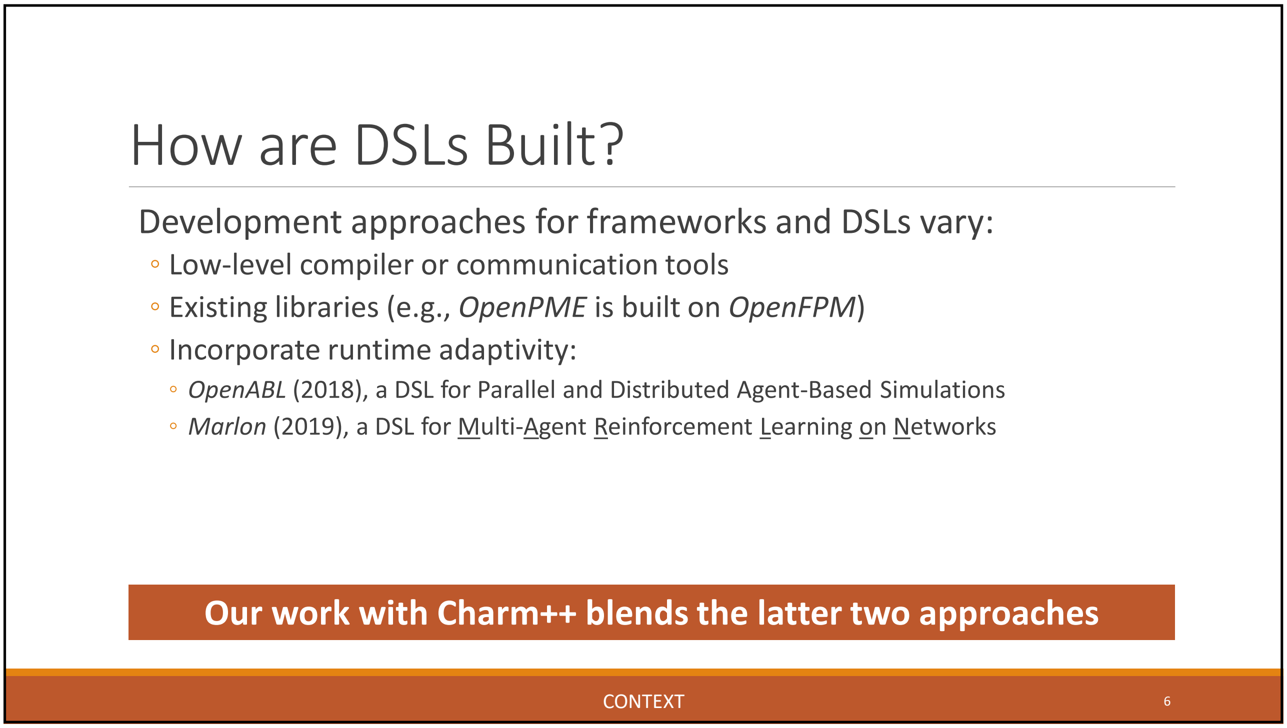
Scalable DSLs require additional considerations:

- Data partitioning, communication, synchronization, etc.

DSL frameworks help overcome these "barriers"

DSL frameworks span various abstraction levels:

- Low-level: fork-join, messaging
- Mid-to-high-level: channels, tasks
- Domain-specific:
  - *QUARC* (2016-), a framework for DSLs with lattice and grid domains
  - *DAWN* (2020-), a compiler toolchain for weather and climate applications



How are DSLs Built?

Development approaches for frameworks and DSLs vary:

- Low-level compiler or communication tools
- Existing libraries (e.g., *OpenPME* is built on *OpenFPM*)
- Incorporate runtime adaptivity:
  - *OpenABL* (2018), a DSL for Parallel and Distributed Agent-Based Simulations
  - *Marlon* (2019), a DSL for Multi-Agent Reinforcement Learning on Networks

Our work with Charm++ blends the latter two approaches

What Charm++ Offers DSLs

Charm’s migratable objects have worthwhile properties for DSLs:

- Intrinsic over-decomposition and runtime adaptivity
- Flexible local view of control and data

Noting these benefits, past Charm-based works built:

- Parallel abstractions: Trees, Futures, PGAS-like Structures (i.e., MSA)
- DSLs: Charisma, DivCon

**These DSLs did not leverage a framework:
repetitive to develop, harder to maintain**

Migratable Objects Pose Challenges

Migratable objects require rich OOP support, but DSL frameworks:

- Often leave classes and generic types as “exercises”
- Performance suffers without first-class support

Embedding migratable objects in……

C++

- Lacking standards
- STL containers force copying
- Syntactic constraints cause verbosity

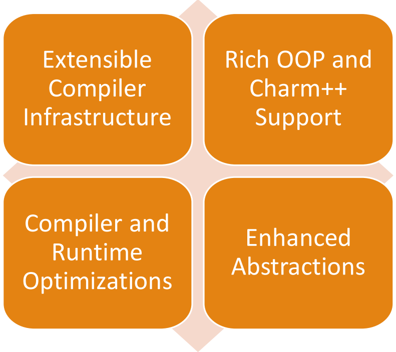
Interpreted or JIT'd languages

- Harms performance
- Although Python and the JVM have improved for some domains

Our Approach

EÍR: A compiler framework to build DSLs based on Charm++

- Extensible compiler supports DSL analysis and transformation passes
- Rich support for OOP and Charm++:
  - Automatically registers and de/serializes user-defined types
- C++ backend with runtime and compiler-level communication optimizations
- Enhanced runtime-level support for performance-critical abstractions



Our Goals

To prove and drive the generality of *EÍR*:

- We built a Scala-inspired, general-purpose language with *EÍR*
- Embedded DSLs within it, like *Charisma* (for data-independent data-flows)

Beyond this, we have explored:

- Elevate visibility of internal mechanisms when:
  - Useful for constructing other parallel abstractions
  - Avoidable performance impact
- Facilitate optimizations through stronger guarantees:
  - Discourage user-level locks, regulate data-sharing, etc.
- Extend the underlying model to minimize "semantic gaps."

Eliminating Tedium

Charm++

```
...
template<typename T> entry void print(T t);
...
// need to register all specializations
extern entry void printer print<std::string>(std::string);
extern entry void printer print<double>(double);
...
```

EÍR

```
...
@entry def print<A>(a: A);
...
printerProxy.print(“hi!”);   // no registration
printerProxy.print(42.0);    // necessary
...
```

De/serialization Optimizations

Identity aware/finer-graph packing

- Effortlessly handles recurrent relationships (doubly-linked graphs)
- Each unique object is packed only once
- Back-references are used for duplicates

Global pointer-to-offset optimizations

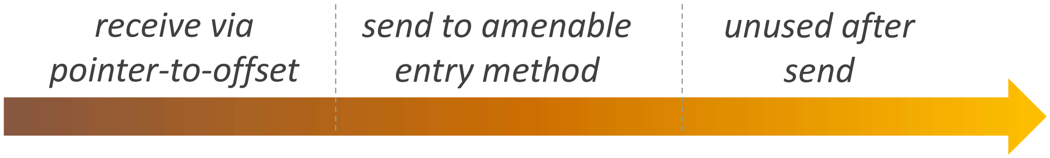
- Deserialize POD-types as pointers to an offset within a message buffer
- Retains ownership of the message with a reference-counted pointer
- **No overhead for encapsulation**
- Charm++ requires an extra copy

```
struct array_holder {
    val arr: array<double>;
    ...
```

In-place message optimizations

- Edit a retained message **“in-place”** then resend it (e.g., arrays)
- EIR uses a control-flow analysis pass to find “repacking” opportunities

What are in-place optimization opportunities?



receive via pointer-to-offset

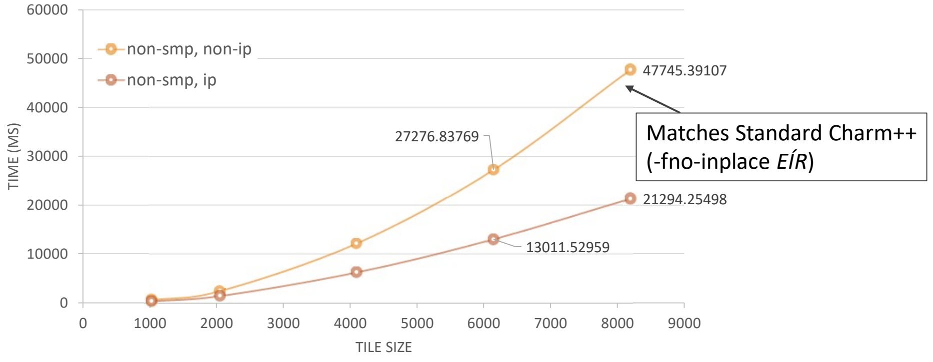
send to amenable entry method

unused after send

```
@overlap for (var block = 0; block < nBlocks; block += 1) {
    when inputA(_ == block, blockA: array<double, 2>),
         inputB(_ == block, blockB: array<double, 2>) => {
        ...
        if (...) {
            self@[...].inputA(block + 1, blockA);
            self@[...].inputB(block + 1, blockB);
        }
    }
}
```

Effects of In-Place Optimizations

EÍR, Cannon's Matmul, Comm-Only
(4 Physical Nodes, 169PEs)



Matches Standard Charm++ (-fno-inplace *EÍR*)

SDAG (Structured Dagger)

SDAG is used to represent parallel control flow in **Charm++**

- Expresses the dependencies between messages and local control flow
- Can direct progress through phases of a computation

```
for (it = 0; it < numIts; it += 1) {
    serial { update_neighbors(...); }
    for (imsg = 0; imsg < numNeighbors; imsg += 1) {
        when recv_update(...) { ... }
    }
    serial { check_and_compute(...); }
    if (it != (numIts - 1)) {
        when recv_iter_summary(...) { ... }
    }
}
when recv_total_summary(...) { ... }
```

Extending SDAG with Mailboxes

Mailboxes enable Erlang-like *selective receives*:

- Blend pattern matching and message processing (rich predicates!)
- SDAG, only one reference number per “when” clause

```
entry void receive(CMK_REFNUM_TYPE);

entry void run(int nIters) {
   serial { this->send(nIters); }
   forall [i] (0:(nIters - 1),1) {
      forall [j] (0:(CkNumPes() - 1),1) {
         when receive [join(i, j)] (CMK_REFNUM_TYPE) {}
      }
   }
}
```

[0-255] Range

```
@mailbox def receive(it: int, pe: int);

@threaded @entry def run(numIters: int) {
   self.send(numIters);
   @overlap for (var i = 0; i < numIters; i += 1) {
      @overlap for (var j = 0; j < ck::numPes(); j += 1) {
         when receive(_ == i, _ == j) => ();
      }
   }
}
```

Unconstrained

Example Pattern Matching

```
trait response<T> { ...}
class pong with response<string> {...}
class ping with response<string> {...}


@mailbox def receive(src: int, msg: response<string>);

@threaded @entry def exchange() {
    @overlap for (var it = 0; it < ck::numPes(); it += 1) {
        await all {
            when receive(src, msg: ping) if src == it => { ... }
            when receive(src, msg: pong) if src == it => acknowledge(src, msg);
        }
    }
    self[@]contribute(...);
}
```

Only matches ping/pong

Other Mailbox Features

Compound Clauses	`when foo(…), bar(…) => { … }`

Await All	`await all {`
	`when foo(…) => {…}`
	`when bar(…) => {… when baz(…) => {…}}`
	`}`

“Overlapped” Execution	Only the ordering of ***baz*** after ***bar*** is enforced, ***foo*** can execute between or after them.

Sections

Sections facilitate efficient operations over subsets of collectives

Their current Charm++ API is constrained:

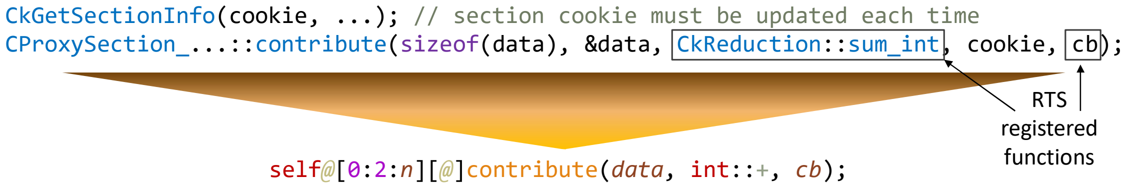
- Explicit, centralized creation
- User-managed “cookies” for correctness

Work is underway to improve sections in Charm++,

- Integrating innovations from Charm4py:
  - Elimination of “cookies”
  - Distributed section creation

compiler support can help too!

Sections (cont.)



```
CkGetSectionInfo(cookie, ...); // section cookie must be updated each time
CProxySection_...::contribute(sizeof(data), &data, CkReduction::sum_int, cookie, cb);
```

```
self@[0:2:n][@]contribute(data, int::+, cb);
```

Goals

- Eliminate section registration, “create” sections at the call-site
- Statically-typed callbacks and combiner function
- Enable using lambda functions for these parameters:

```
self[@]contribute(..., (i, j) => ..., i => println(`the result was $i!`));
```

Other Features of *EÍR*

Rich support for generic types,
- Support for parameter packs and tuple manipulation
- Swift-like “where” clauses to enforce constraints on generics
- Like “std::enable_if” or “requires” in C++20 (i.e., SFINAE)

Operator overloading,
- Objects can be used as “functors”
- Arbitrary identifiers for infix operators (ranges by “1 **to** n”)

Pattern matching,
- User-defined “extractor” methods (like Scala)
- Optional values can be matched with:
  “case some(x) => …” or “case none() =>”

Other Features of *EÍR* (cont.)

Features woven together to form a rich STL that…
- Also includes parallel abstractions:
  - Distributed Hash Tables and Tuple Spaces
  - Channels
  - Futures

Written various mini-apps:
- Jacobi’s method in 2D
- Cannon’s matrix-matrix multiplication

Results (cont.)

Performance typically matches or exceeds Charm++:

- Gains usually in the 5~20% range...
- *Unless* larger optimizations are involved (e.g., Cannon’s)
- Complexities in message delivery preclude larger gains
- Required to facilitate mailboxes, lightweight sections, etc.

Thus, we have recently focused on micro-benchmarking:

- Roundtrip “ping pong” time
- OSU-style bandwidth

We are “closing the gap” between our infrastructure and Charm++

Future Work

Aside from correcting the issues mentioned…

- Limitations of the C++ backend, microbenchmark performance...

We have many ideas for future work with/on our infrastructure:

- Write more mini-apps: LeanMD, BarnesHut, HPCCG, etc.
- Integrate DSLs from Charm’s “back-catalog” (DivCon and MSA)
- Expose a textual representation for *EÍR*
  - Perhaps an XML-based AST like *Omni*’s XcodeML?
- Ensure *EÍR’s* generated code is robustly migratable
- Provide facilities for using hardware accelerators

Conclusions

EÍR, a compiler framework to build DSLs based on Charm++

- Optimizations for message packing
- Enhanced message processing via mailboxes
- Offers “lightweight” sections

Plan to publish soon – more to say then!

Please reach out if you’re interested!

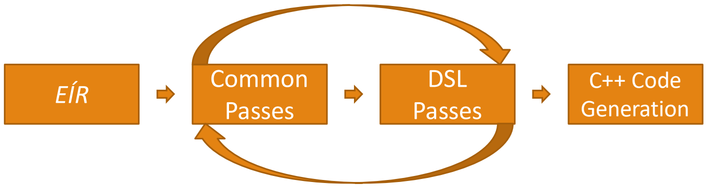
What’s in the Box?

- Lightweight sections…
  - Creation, reductions, and multicasts
- Mailboxes
- Futures
- Aggregations
- Hashing
- De/serialization
- Migratable threading primitives
- And… more!

Code Generation in *EÍR*

Like Charj before it, *EÍR* is written in Scala and targets C++:

- Competitive performance
- Trivializes interoperability with Charm++ libraries



EÍR → Common Passes → DSL Passes → C++ Code Generation

- *EÍR's* C++ backend currently has some limitations (e.g., generics and lambdas) compared to targeting a purpose-built backend (e.g., MLIR)
- However, with additional effort, we can overcome them