DELITE: Simplifying Parallel Programming with Domain Specific Languages (DSLs)

Pervasive Parallelism (PPL) Lab. (http://ppl.stanford.edu)

Hassan Chafi, Nathan Bronson, Arvind Sujeeth, Kunle Olukotun

The Era of Many-Cores is Now

Problem: Explicitly-parallel programming is too complex
- Parallel programming environments prioritize efficiency
  - Performance or performance/watt
- Full multi-core utilization requires **extra** work
  - As a result, domain experts first create sequential codes
  - *Expressivity+scalability might be better than raw efficiency*

Problem: Automatic parallelization is limited
- The holy grail of compiler research
  - Heroics to perform high-level optimizations
- SW/HW speculative techniques have some success
  - Scalability limited by incidental true dependencies
- All approaches have limited app coverage

Solution: Leverage DSL expressivity
- Domain Specific Languages allow for higher productivity by providing high-level data-types and ops tailored to domain
  - Relations, triangles, matrices, filtered traversal, …
- Expresses intent without implementation artifacts

Solution: Use DSLs to provide scalable implicit parallelism
- DSL author encodes parallelism hidden from DSL user
- High-level interface gives flexibility to implementation
- Declarative description of parallelism & locality patterns
- Portable and scalable specification of parallelism
  - Adjust data structures, scheduling as system scales up

DELITE: A Framework for Building Parallel DSLs

Audience
- **DSL user:** good performance + scalability, without writing parallel code
- **DSL writer**: DSL using DELITE framework is only incrementally more difficult to create than a sequential DSL

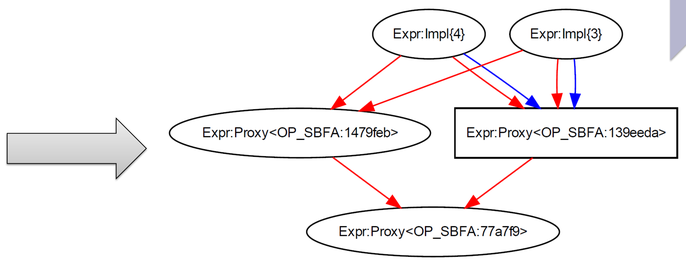
Approach
- **Toolbox for DSL creator**
  - Embed in Scala, an OO/functional hybrid language
  - Add parallel control and locality structures
  - Replace Scala's collection classes
    - Simplify the hierarchy, unify with numeric vectors
    - Add implicit parallelism
    - Remove boxing overheads
  - Provide aggressive task creation and good scheduling
- **Domain specific optimizations**
  - Use domain knowledge to optimize & annotate code
- **Run-time optimizations**
  - Locality-aware scheduling
  - Data movement and prefetching support

Dynamic Expression Tree Rewriting using Strongly Typed Proxies

```
// factory method hides implementation type during construction
object Expr { def apply(value: Int): Expr = new Impl(value) }

// trait hides implementation type during use, injects operations into both proxy and concrete impl
trait Expr extends DeliteDSLType with HasImpl[Expr.Impl] {
    def + (rhs: Expr) = run((a: Impl, b: Impl) => a.add(b)) }

class Impl(val value: Int) extends Expr { protected def add(rhs: Impl) = Expr(val + rhs.val) }

class Proxy extends DeliteProxy[Expr,Impl] with Expr
```

DSL user calls ops from sequential program → DSL methods submit ops to DELITE and return resulting proxy → DELITE applies generic and domain-specific transformations to DAG → DELITE scheduler executes ops in an order optimized for locality

```
val a = Expr(4)
val b = Expr(3)
val c = a + b
val d = a + b
val e = c + d
println(d.value)
```



Resulting expression tree. Red edges denote submitted ops, blue edges denote executed ops.

Implicit Parallelism from Collections ⇒ Unified Class Hierarchy

Implicit parallelism from bulk data
- Transform iteration/generation into foreach/map:

```
for (i <- 0 until n) { a(i) = b(i) * c(i) }
range(0, n).foreach(i => { a(i) = b(i) * c(i) })

val b = for (elem <- a) yield { 10*elem }
a.map(elem => { 10*elem })
```

- Collection implementation must support parallelism
  - Does type encode sequential/parallel choice? Bad option
    - Existing collection classes iterate sequentially
    - Added collection classes iterate in parallel

*Parallelism safety is a property of the use, not the collection. Explicit selection of a parallel implementation is not **implicit** parallelism.*

Solution: new collection class for all uses
- Potential parallelism is always present
- Execution mode selected dynamically using strategies

Complete encapsulation of implementation
- Factory methods hide implementation during construction

```
val loc = Vec(x, y, z)
```

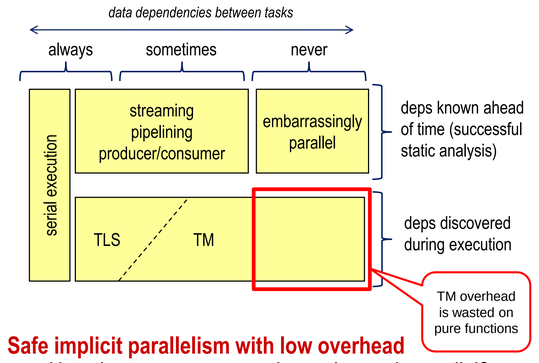
- Implicit factories propagate unboxed implementation through JVM's type erasure

```
abstract class Vec[A] {
  def map[B](f: A=>B)(implicit fact: Vec.Factory[B]) = ...
```

- Implementations specialized for bulk primitives
  - No boxing for Vec[Int], Vec[Double]
- Implementations specialized for bulk tuples
  - Automatic unboxing/reboxing for Vec[(Int,Int)], …

Is List[Double] different than Vector[Double]?
- Simplest for the user would be a single collection type, with numeric operations overloaded
  - Vec[String] * Double should be a compile-time error
  - Vec[DSLCreatedType] * Double should be implementable
- Op implementation supplied by implicit higher-order arg
  - Short-circuited for efficiency for primitive op primitive

Dynamic Verification of Data Race Freedom



Safe implicit parallelism with low overhead
- How do we prove user code can be run in parallel?
  - Type system extensions? *Scary for the DSL user*
  - Static analysis? *Possible, but hard to integrate and scale*
  - Dynamic techniques? *High overhead without HW*

"Poor Man's Transactions"
- Replace txn read and write sets with predicates
  - *Some predicates require few or zero read barriers!*
- Dynamic escape analysis proves writes are thread-local
- Help from the scheduler guarantees strong isolation

Use case: code that is dynamically weakly pure (mostly)
- This kind of code scales well
- Add exceptions for predicted accesses to PPL collections
- Low overhead and no synchronization for success
- Fall back to TM or sequential execution if necessary

Status and Next Steps

Status:
- Initial DSL targeting ML applications implemented
- Basic DELITE proxy system implemented
- Initial implementation of Unified Class Hierarchy

Next steps:
- Complete prototype implementation of DELITE runtime
- Pattern matching to express domain-specific op rewrite rules
- Collect initial performance numbers

Contact information

Group Website
`http://ppl.stanford.edu`

E-mail Addresses (Students)
`{hchafi, nbronson, asujeeth}@stanford.edu`

E-mail Address (Faculty Advisor)
`kunle@stanford.edu`