The Stanford Pervasive Parallelism Lab

Kunle Olukotun
Stanford University

25th ORAP, Paris. France
14th October, 2009

Goals and Organization

- Goal: the parallel computing platform for 2015
  - Parallel application development practical for the masses
    - Joe the programmer…
    - Simplify large-scale parallel computing for scientists
  - Parallel applications without parallel programming

- PPL is a collaboration of
  - Leading Stanford researchers across multiple domains
    - Applications, languages, software systems, architecture
  - Leading companies in computer systems and software
    - Sun, AMD, Nvidia, IBM, Intel, NEC, HP

- PPL is open
  - Any company can join; all results in the public domain

The PPL Team

- Applications
  - Ron Fedkiw, Vladlen Koltun, Sebastian Thrun
- Programming & software systems
  - Alex Aiken, Pat Hanrahan, John Ousterhout, Mendel Rosenblum
- Architecture
  - Bill Dally, John Hennessy, Mark Horowitz, Christos Kozyrakis, Kunle Olukotun (director)

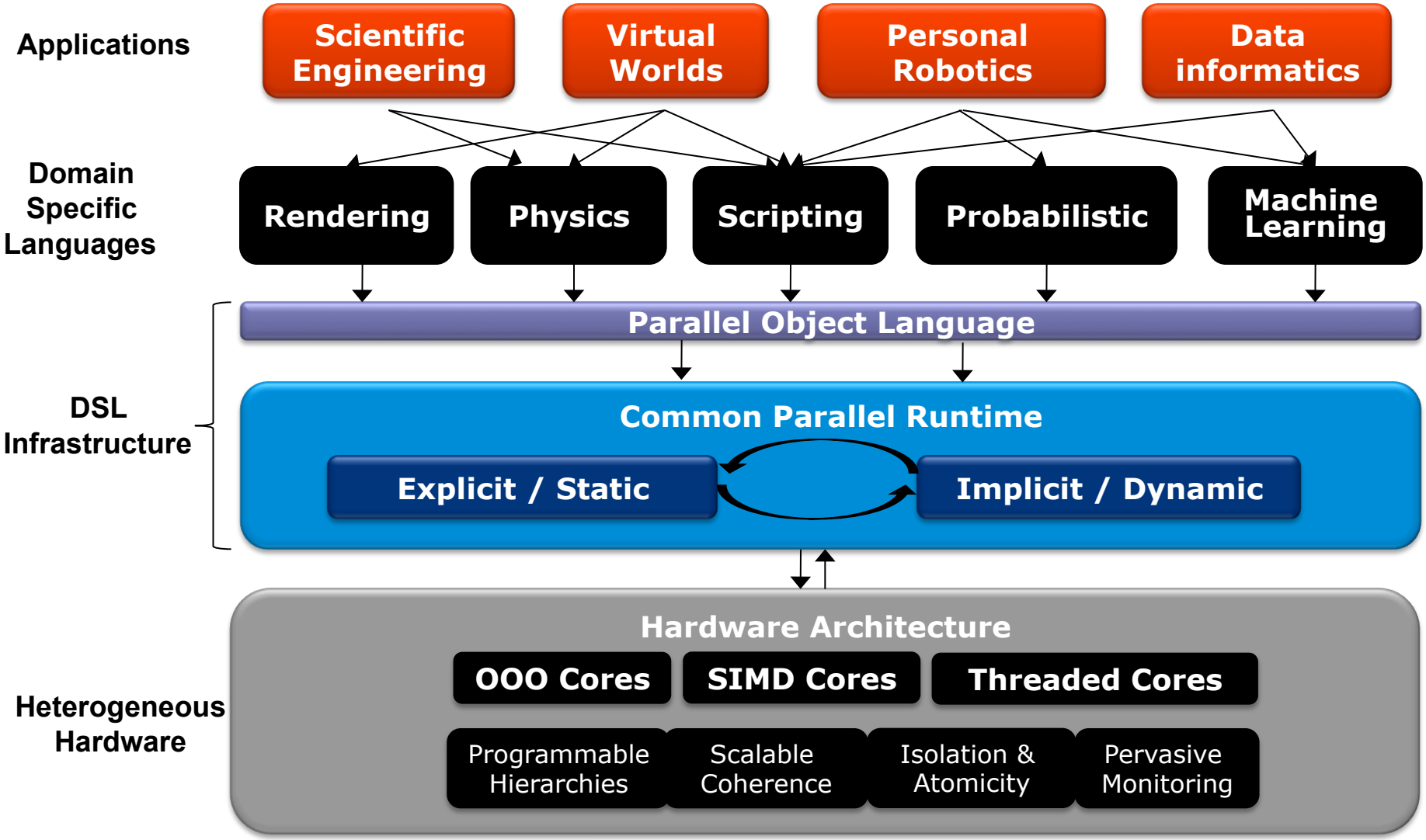
Heterogeneous Parallel Computing

- Cluster (e.g. 100K node cluster)
  - Distributed memory
  - MPI for communication
- Multi-core SMP (e.g. 32 core, 4-socket systems)
  - Shared memory, transaction memory in future
  - Threads/locks, OpenMP
- Many-core GPU (e.g. Nvidia Fermi, 512 cores)
  - Separate GPU memory
  - Data parallel programming model (SIMT in CUDA)
- Combinations of these systems + HW accelerators

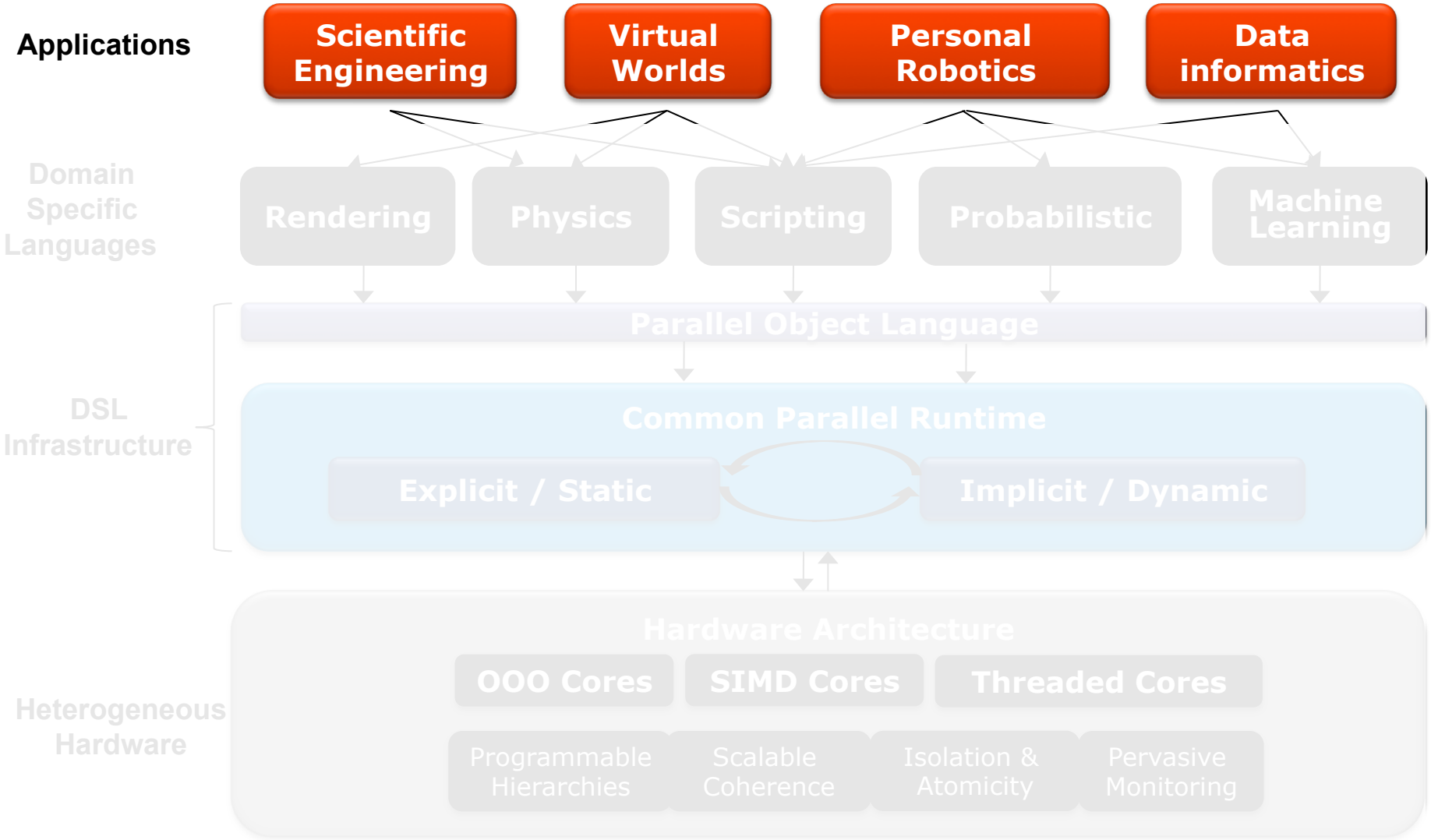
Homogeneous Parallel Programming

- The computing world is heterogeneous
  - Driven by energy and silicon area efficiency
- Question: Is it possible to write one parallel program that runs in many modes?
- Hypothesis: Yes, but needs domain specific programming environments and languages
  - PPL goal: prove this

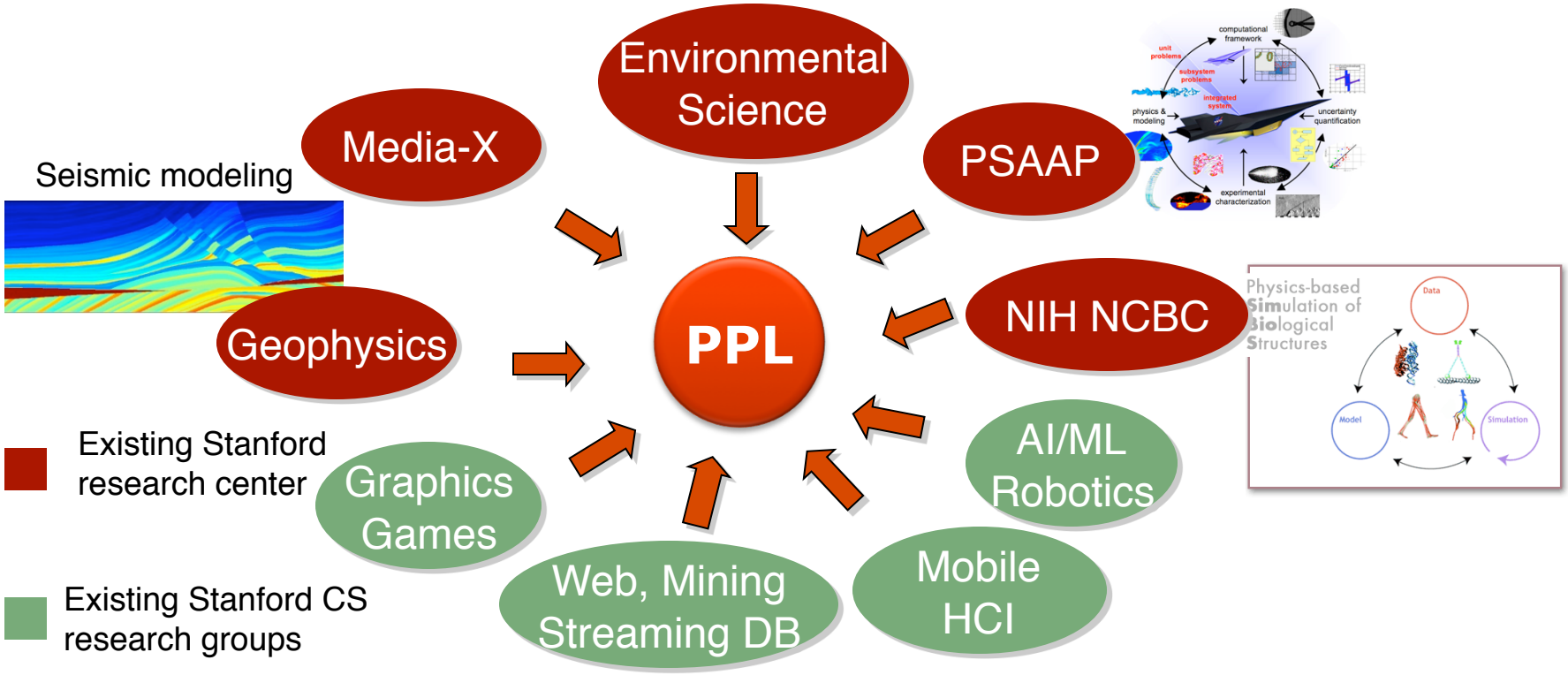
The PPL Vision



Applications

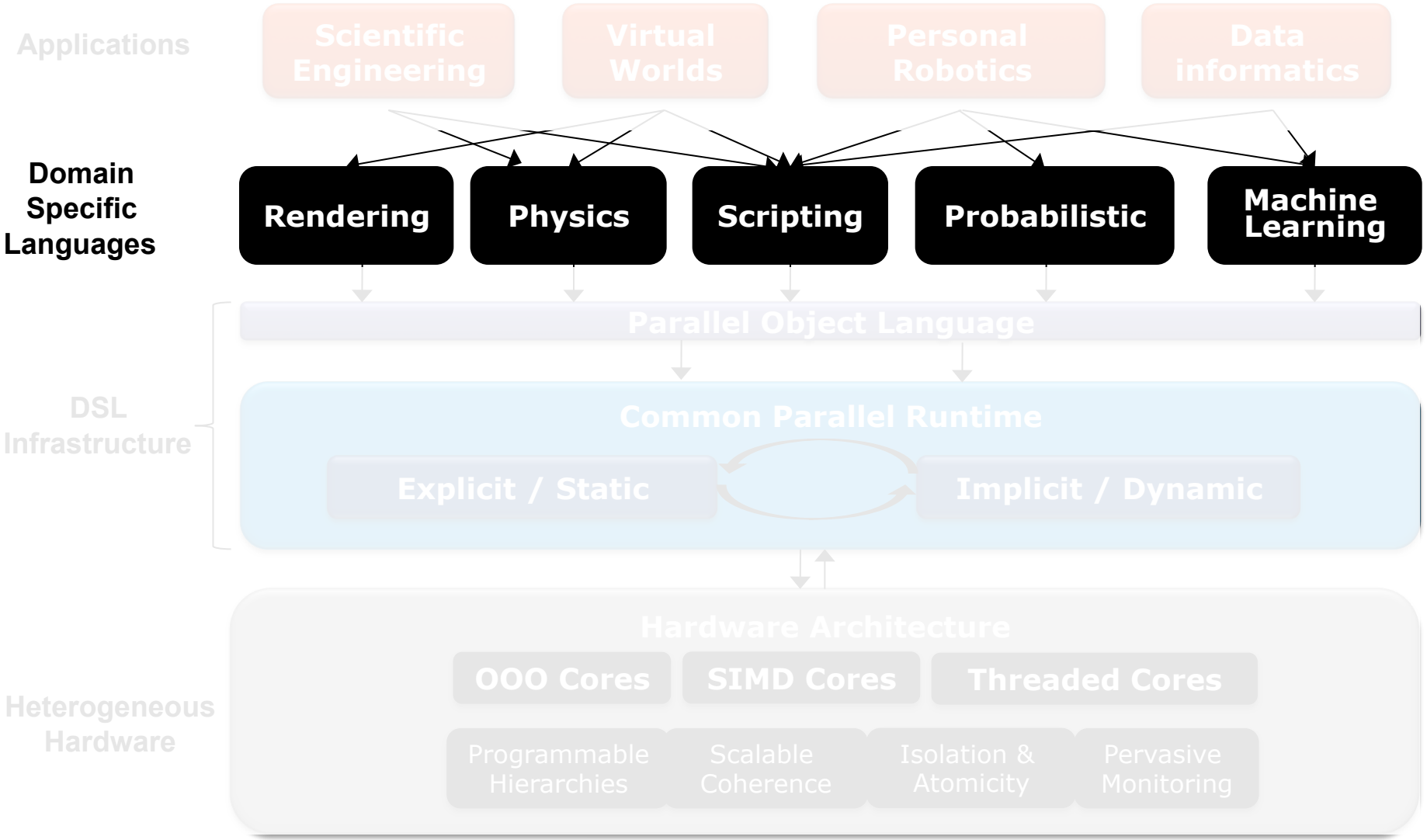


Demanding Applications



- Leverage domain expertise at Stanford
  - National centers for scientific computing & CS research groups

Domain Specific Languages



Domain Specific Languages (DSLs)

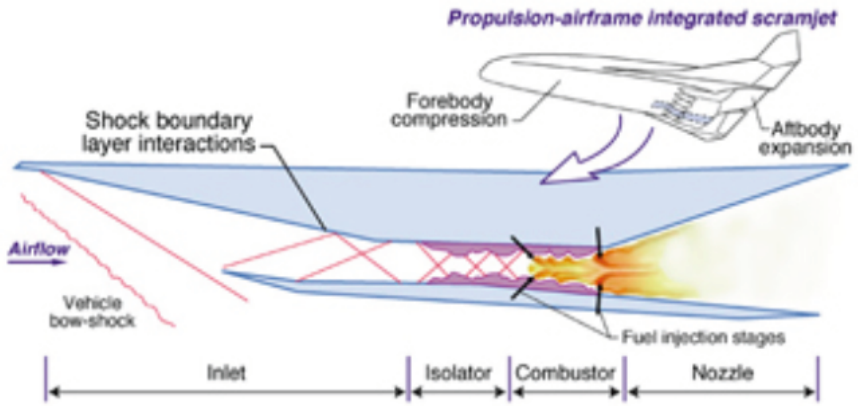
- High level language targeted/restricted to specific domain
  - E.g.: SQL, Matlab, OpenGL, Ruby/Rails, …
  - Usually declarative and simpler than GP languages
  - High-level data types & ops (e.g. relations, matrices, triangles, …)
  - Separate domain expertise (DSL user) from computer science expertise (DSL developer)
- DSLs are a hot topic now
  - Programming language community (C#, Scala)
  - Web programming environments (Ruby)

DSL Advantages

- DSLs ⇒ higher productivity for users
  - DSL user
  - High-level data types & ops (e.g. relations, triangles, …)
  - Express high-level intent w/o implementation artifacts
- DSLs ⇒ scalable parallelism for the system
  - DSL developer
  - Declarative description of parallelism & locality patterns
    - Can be ported or scaled to available machine
  - Allows for domain specific super-optimization
    - Automatically adjust structures, mapping, and scheduling

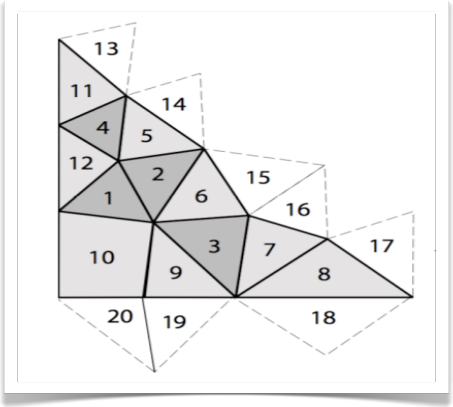
Hypersonic (Mach 7) Vehicle Simulation

- Characterize the limits of a hypersonic propulsion system using predictive computations (DOE PSSAP)
  - Primary focus is *unstart* phenomena triggered by thermal choking in a hydrogen-fueled scramjet
- Limitations of current unstructured RANS solver code
  - Optimized for Cluster/MPI but cannot run on other architectures
  - 20% scientific code, 80% plumbing code
  - Minor changes in application level code requires major changes in framework
  - Large learning curve for incoming students who are not parallel-computing aware

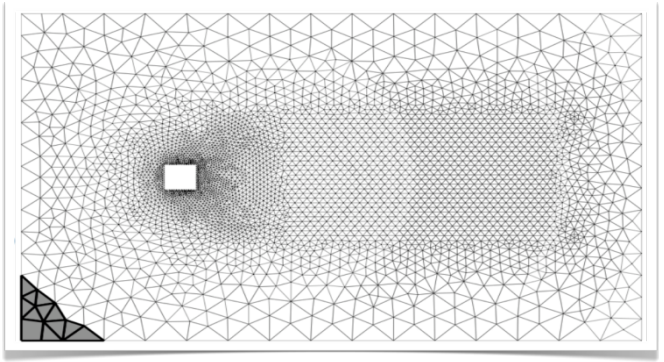


Example DSL: Liszt

- Goal: simplify code of mesh-based PDE solvers
  - Write once, run on any type of parallel machine
  - From multi-cores and GPUs to clusters



- Language features
  - Built-in mesh data types
    - Vertex, edge, face, cell
  - Collections of mesh elements
    - cell.faces(), face.edgesCCW()
  - Mesh-based data storage
    - Fields, sparse matrices



Liszt Code Example (Scala)


```
val position = vertexProperty[double3](“pos”)
val A = new SparseMatrix[Vertex,Vertex]

for (c <- mesh.cells) {
    val center = average position of c.vertices
    for (f <- c.faces) {
        val face_dx = average position of f.vertices – center
          for (e <- f.edges With c CounterClockwise) {
               val v0 = e.tail
               val v1 = e.head
               val v0_dx = position(v0) – center
               val v1_dx = position(v1) – center
               val face_normal = v0_dx cross v1_dx
               // calculate flux for face …
               A(v0,v1) += …
               A(v1,v0) -= …
```

Liszt Code Example (Scala)

```
val position = vertexPr
val A = new SparseMatri

for (c <- mesh.cells) {
    val c   n = average position of c.vertices
                                                  nter
            val v0 = e.tail
            val
            val
            val v1_dx =          n(v1) - center
            val face_normal = v0_dx cross v1_dx
            // calculate flux for face …
            A(v0,v1) += …
            A(v1,v0) -= …
```

High-level data types & operations

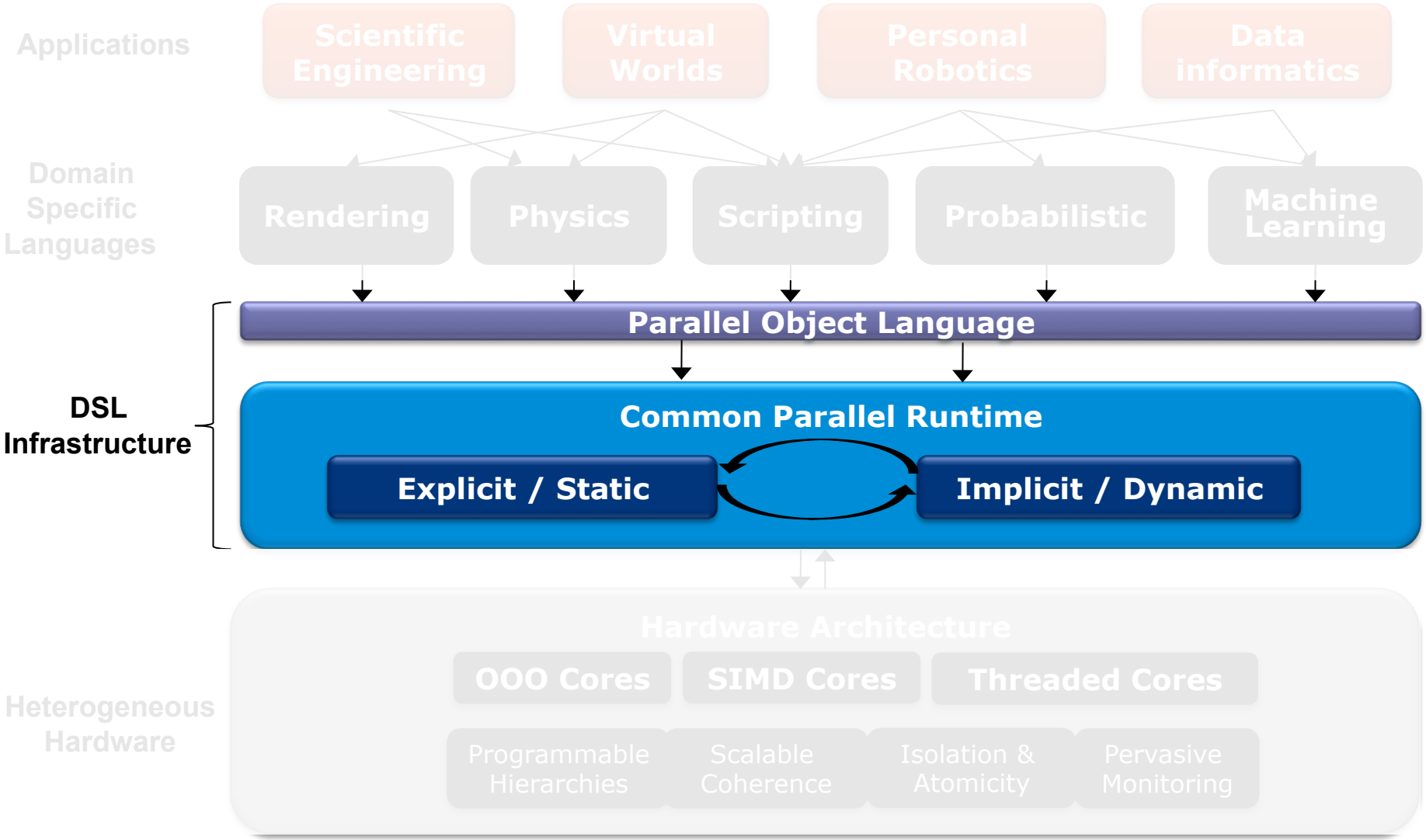
Explicit parallelism using map/reduce/forall
Implicit parallelism with help from DSL & HW

No low-level code to manage parallelism

Liszt Parallelism & Optimizations

- Liszt compiler & runtime manage parallel execution
  - Data layout & access, domain decomposition, communication, …
- Domain specific optimizations
  - Select mesh layout (grid, tetrahedral, unstructured, custom, …)
  - Select decomposition that improves locality of access
  - Optimize communication strategy across iterations
- Optimizations are possible because
  - Mesh semantics are visible to compiler & runtime
  - Iterative programs with data accesses based on mesh topology
  - Mesh topology is known to runtime

DSL Infrastructure



DSL Infrastructure Goals

- Provide a shared framework for DSL development
  - Simplify DSL development
- Features
  - Common parallel language that retains DSL semantics
  - Mechanism to express domain specific optimizations
  - Static compilation + dynamic management environment
    - For regular and unpredictable patterns respectively
  - Synthesize HW features into high-level solutions
    - E.g. from HW messaging to fast runtime for fine-grain tasks
  - Exploit heterogeneous hardware to improve efficiency

Features of a Parallel Object Language

- Support for functional programming (FP)
  - Declarative programming style for portable parallelism
  - High-order functions allow parallel control structures
- Support for object-oriented programming (OOP)
  - Familiar model for complex programs
  - Allows mutable data-structures and domain-specific attributes
- Managed execution environment
  - For runtime optimizations & automated memory management

Scala: A Parallel Object Language

- Embed DSLs in the Scala language
  - Encourages use pf best FP and OOO approach for a particular problem
  - Can implement powerful abstractions as a library
  - Designed to embed DSLs
    - Optional syntax, implicit type conversions, implicit parameters, …
  - Compiles to Java bytecode ⇒ Hotspot JVM
- Scala based DSLs provide scalable implicit parallelism
  - DSL developer 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

- Domain Extracted Locality Informed Task Execution
- Goals
  - 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
- Scala toolbox for DSL creator
  - Parallel control and locality structures
  - Replace Scala's collection classes
    - Simplify the hierarchy, unify with numeric vectors
    - Add implicit data parallelism
  - Provide aggressive task creation with proxies
  - Facilitate domain specific optimizations

Delite Operation

- Static analysis at compile time
  - Identify data parallel Ops
  - Optimize data layout
- Proxy delayed execution allows runtime optimization
  - Easy to create a new DSL type that uses proxies
  - Transparent to DSL user
- Runtime optimization and Scheduling
  - Parse dependencies and build task DAG
  - Domain specific
    - Tree-height reduction
    - Op Fusing
    - Pattern matching
  - Generic
    - Common subexpression
  - Locality-aware scheduling

DSL Execution with the Delite Parallel Runtime

PERVASIVE PARALLELISM LABORATORY PPL

Application

```
def example(a: Matrix[Int],
            b: Matrix[Int],
            c: Matrix[Int],
            d: Matrix[Int]) =
{

  val ab = a * b
  val cd = c * d
  return ab + cd

}
```

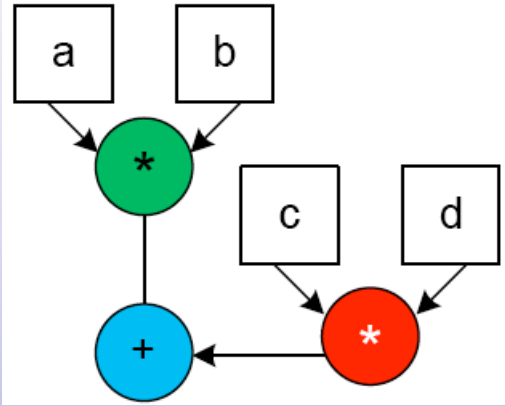
Calls Matrix DSL methods

Matrix DSL

```
def *(m: Matrix[Int]) =
  delite.defer(OP_mult(this, m))

def +(m: Matrix[Int]) =
  delite.defer(OP_add (this, m))
```

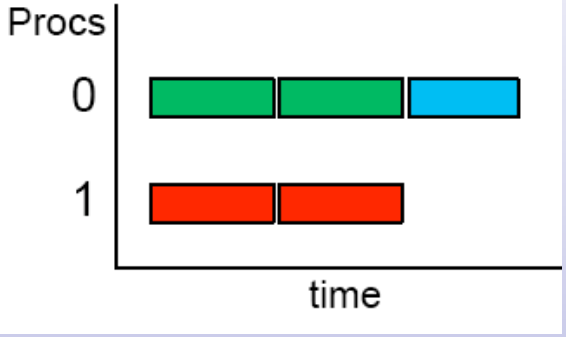
DSL defers OP execution to Delite

Delite Runtime



Delite applies generic & domain transformations and generates mapping

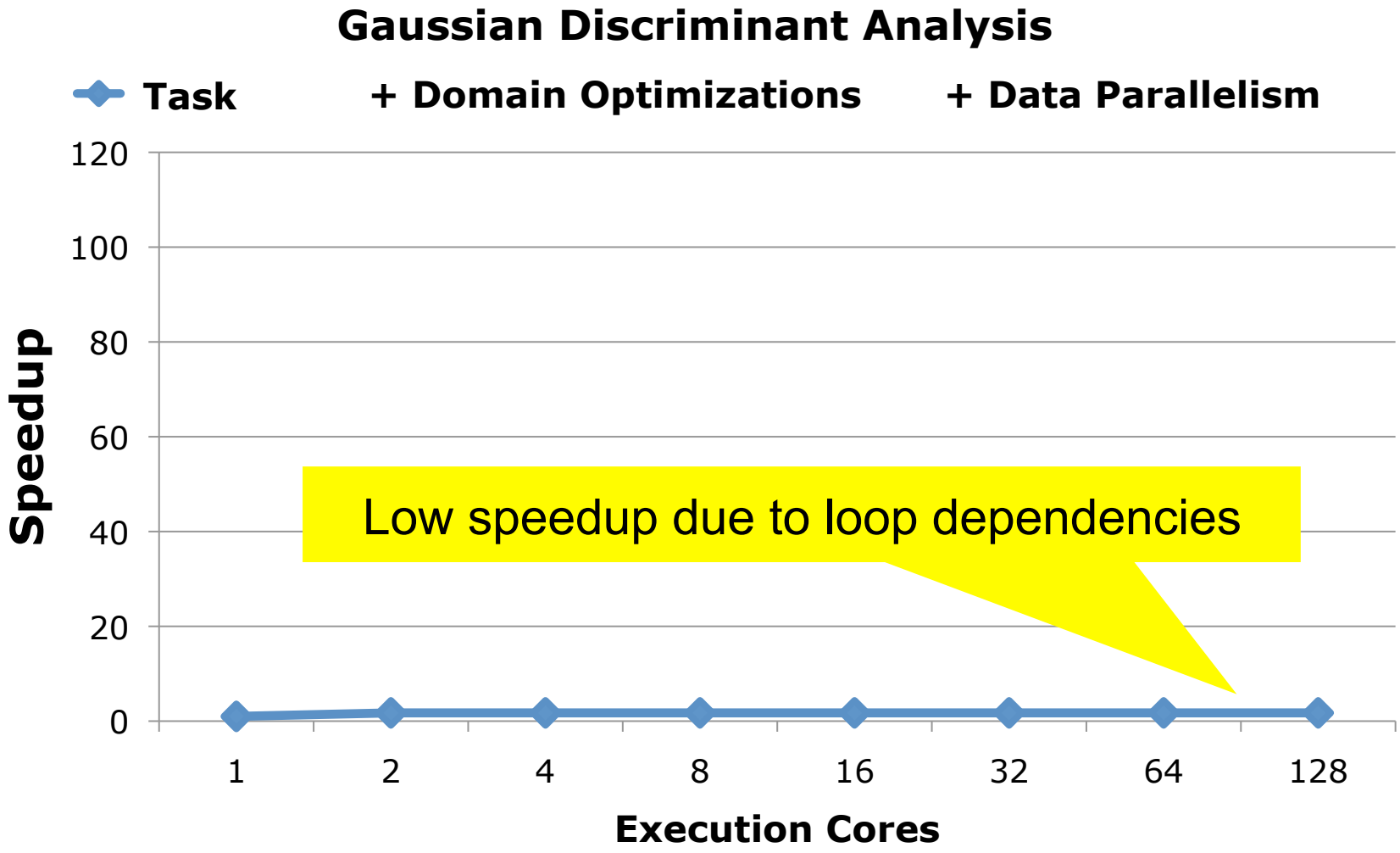
Hardware Schedule



Gaussian Discriminant Analysis


```
for (i <- 0 until m){
  if (y(i) == false){              // y is vector
    y_zeros += 1
    mu0_num = mu0_num + x(i)       // x is a matrix
  }
  if (y(i) == true){
    y_ones += 1
    mu1_num = mu1_num + x(i)
  }
}

val phi = 1./m * y_ones
val mu0 = mu0_num / y_zeros
val mu1 = mu1_num / y_ones

var sigma = Matrix.zeros(n,n)
for (i <- 0 until m){
  if (y(i) == false){
    sigma = sigma + ((x(i)-mu0).trans).outer(x(i)-mu0)
  }
  else if (y(i) == true){
    sigma = sigma + ((x(i)-mu1).trans).outer(x(i)-mu1)
  }
}
```

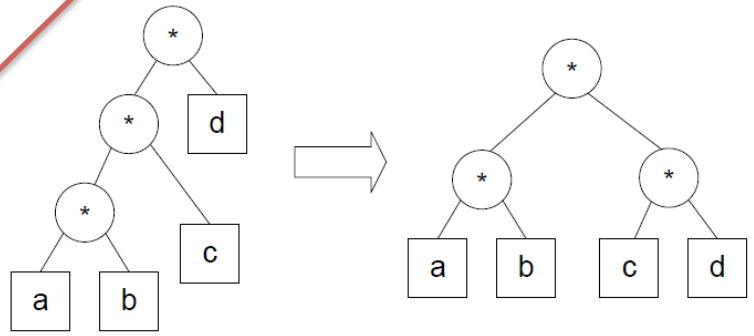
Performance with Delite



Gaussian Discriminant Analysis

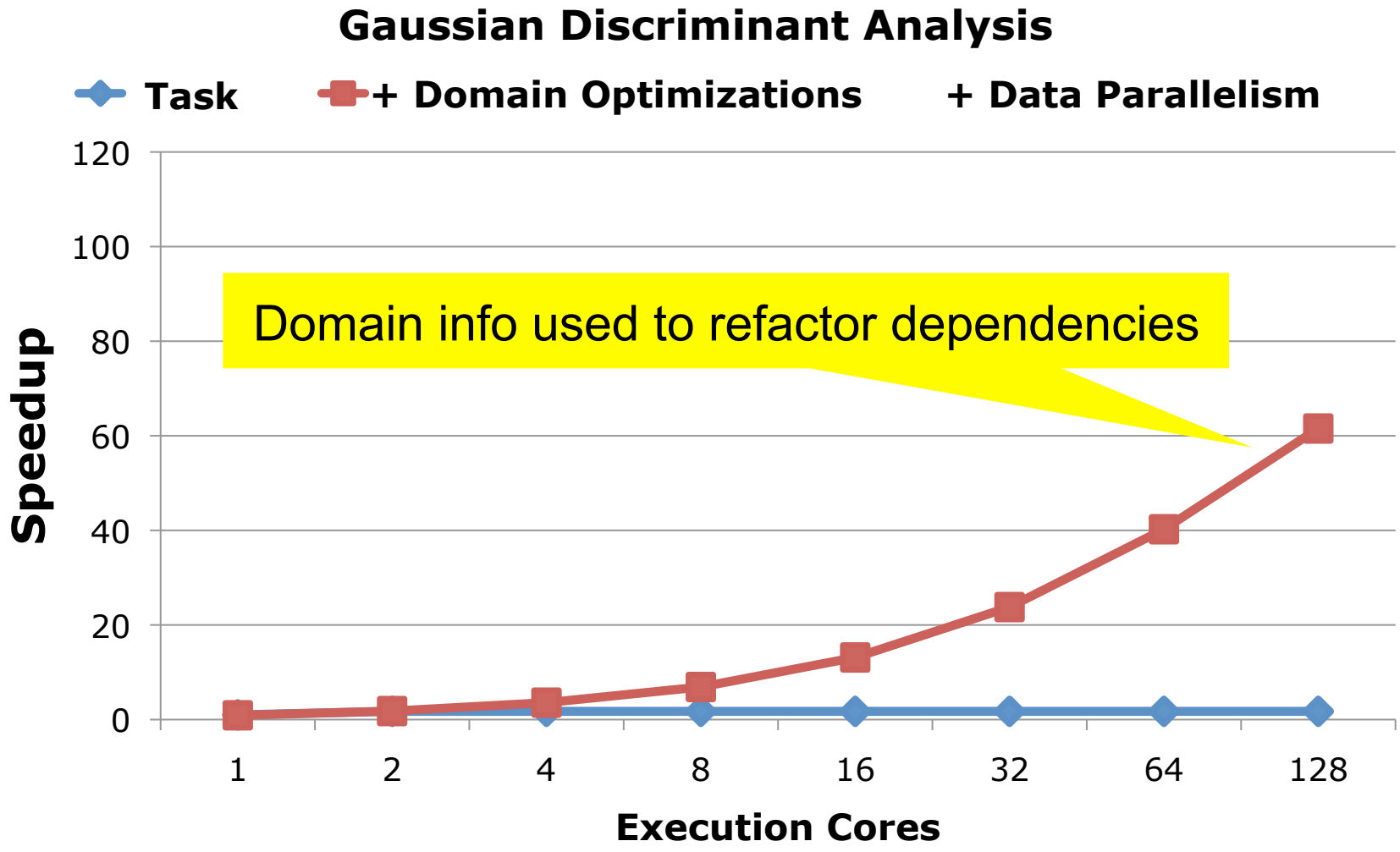

```
for (i <- 0 until m){
  if (y(i) == false){
    y_zeros += 1
    mu0_num = mu0_num + x(i)
  }
  if (y(i) == true){
    y_ones += 1
    mu1_num = mu1_num + x(i)
  }
}

val phi = 1./m * y_ones
val mu0 = mu0_num / y_zeros
val mu1 = mu1_num / y_ones

var sigma = Matrix.zeros(n,n)
for (i <- 0 until m){
  if (y(i) == false){
    sigma = sigma + ((x(i)-mu0).trans).outer(x(i)-mu0)
  }
  else if (y(i) == true){
    sigma = sigma + ((x(i)-mu1).trans).outer(x(i)-mu1)
  }
}
```

Domain info used to refactor dependencies

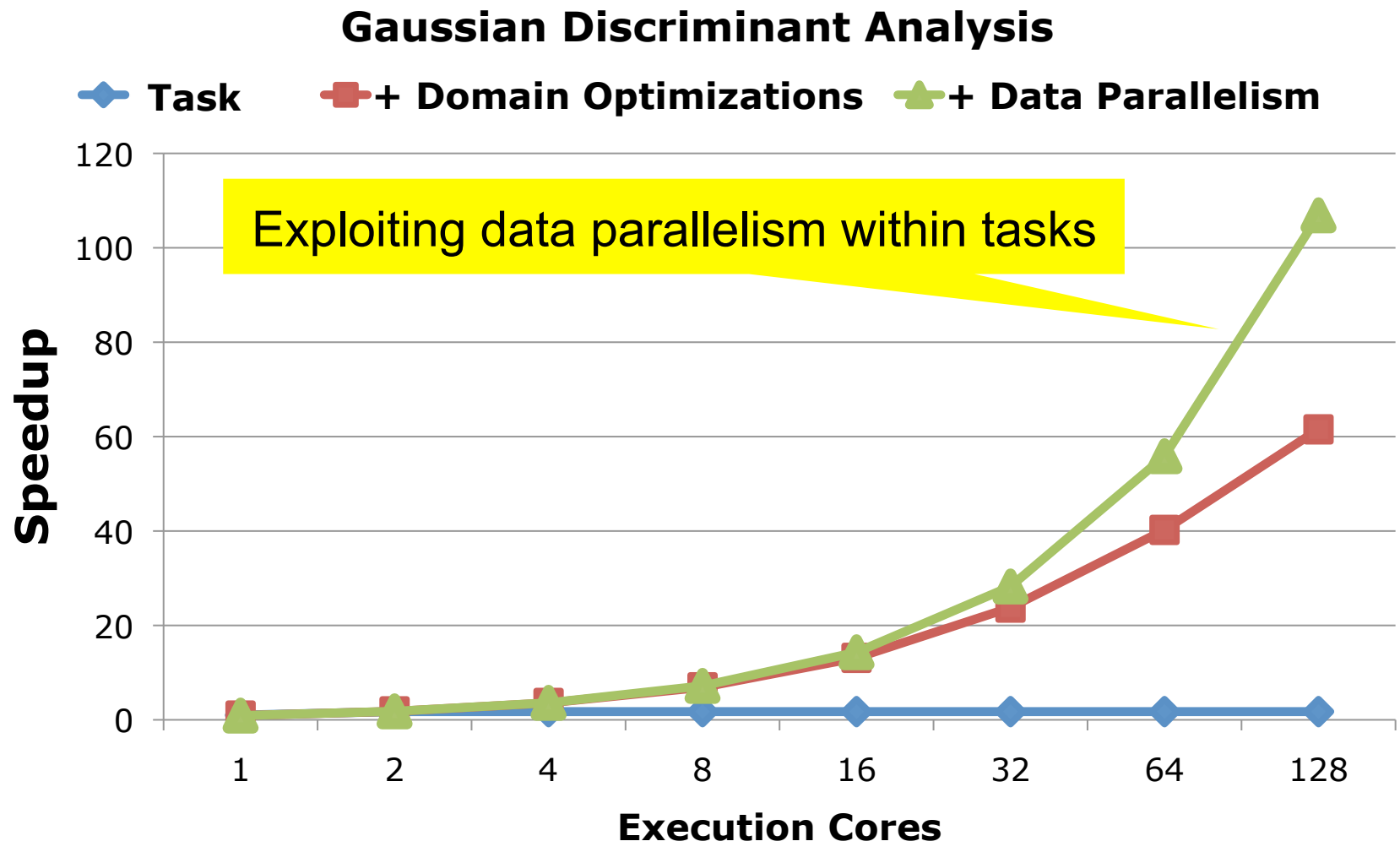


Domain info used to fuse operations

Performance with Delite



Performance with Delite



PPL Summary

- Goal: make parallel computing practical for the masses
- Technical approach
  - **Domain specific languages (DSLs)**
    - Simple & portable programs
  - **Heterogeneous hardware**
    - Energy and area efficient computing
  - Working on the SW & HW techniques that bridge them
- More info at: **http://ppl.stanford.edu**

Acknowledgements

- Faculty
  - Alex Aiken
  - Juan Alonso
  - Bill Dally
  - Pat Hanrahan
  - Mark Horowitz
  - Christos Kozyrakis
- Students
  - Nathan Brosnon
  - Hassan Chaffi
  - Zach Devito
  - Montse Medina
  - Michael Barrientos
  - Daniel Sanchez
  - Arvind Sujeeth