The Cascade High Productivity Programming Language

Hans P. Zima

University of Vienna, Austria
and
JPL, California Institute of Technology, Pasadena, CA

CMWF Workshop on the Use of High Performance Computing in Meteorolog
Reading, UK, October 25, 2004

Contents

Abstraction in Programming

Programming models and languages bridge the gap between "reality" and hardware – at different levels of abstraction - e.g.,

- *assembly languages*
- *general-purpose procedural languages*
- *functional languages*
- *very high-level domain-specific languages*

Abstraction implies loss of information –
gain in simplicity, clarity, verifiability, portability
versus potential performance degradation

The Emergence of High-Level Sequential Languages

The designers of the very first high level programming language were aware that their success depended on acceptable performance of the generated target programs:

John Backus (1957): *"… It was our belief that if FORTRAN … were to translate any reasonable scientific source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger …"*

High-level algorithmic languages became generally accepted standards for sequential programming since their advantages outweighed any performance drawbacks

For parallel programming no similar development took place

The Crisis of High Performance Computing

- **Current HPC hardware: large clusters at reasonable cost**
  - *commodity clusters or custom MPPs*
  - *off-the-shelf processors and memory components, built for mass market*
  - *latency and bandwidth problems*
- **Current HPC software**
  - *application efficiency sometimes in single digits*
  - *low-productivity "local view" programming models dominate*
    - *explicit processors: local views of data*
    - *program state associated with memory regions*
    - *explicit communication intertwined with the algorithm*
    - *wide gap between domain of scientist and programming language*
  - *inadequate programming environments and tools*
  - *higher level approaches (e.g., HPF) did not succeed, for a variety of reasons*

State-of-the-Art

Current parallel programming language, compiler, and tool technologies are unable to support high productivity computing

➔

New programming models, languages, compiler, and tool technologies are necessary to address the productivity demands of future systems

Goals

- **Make Scientists and Engineers more productive:**

  provide a higher level of abstraction

- **Support "Abstraction without Guilt" [Ken Kennedy]:**

  increase programming language usability without sacrificing performance

Contents

1 Introduction

2 Programming Models for High Productivity Computing Systems

3 Cascade and the Chapel Language Design

4 Programming Environments

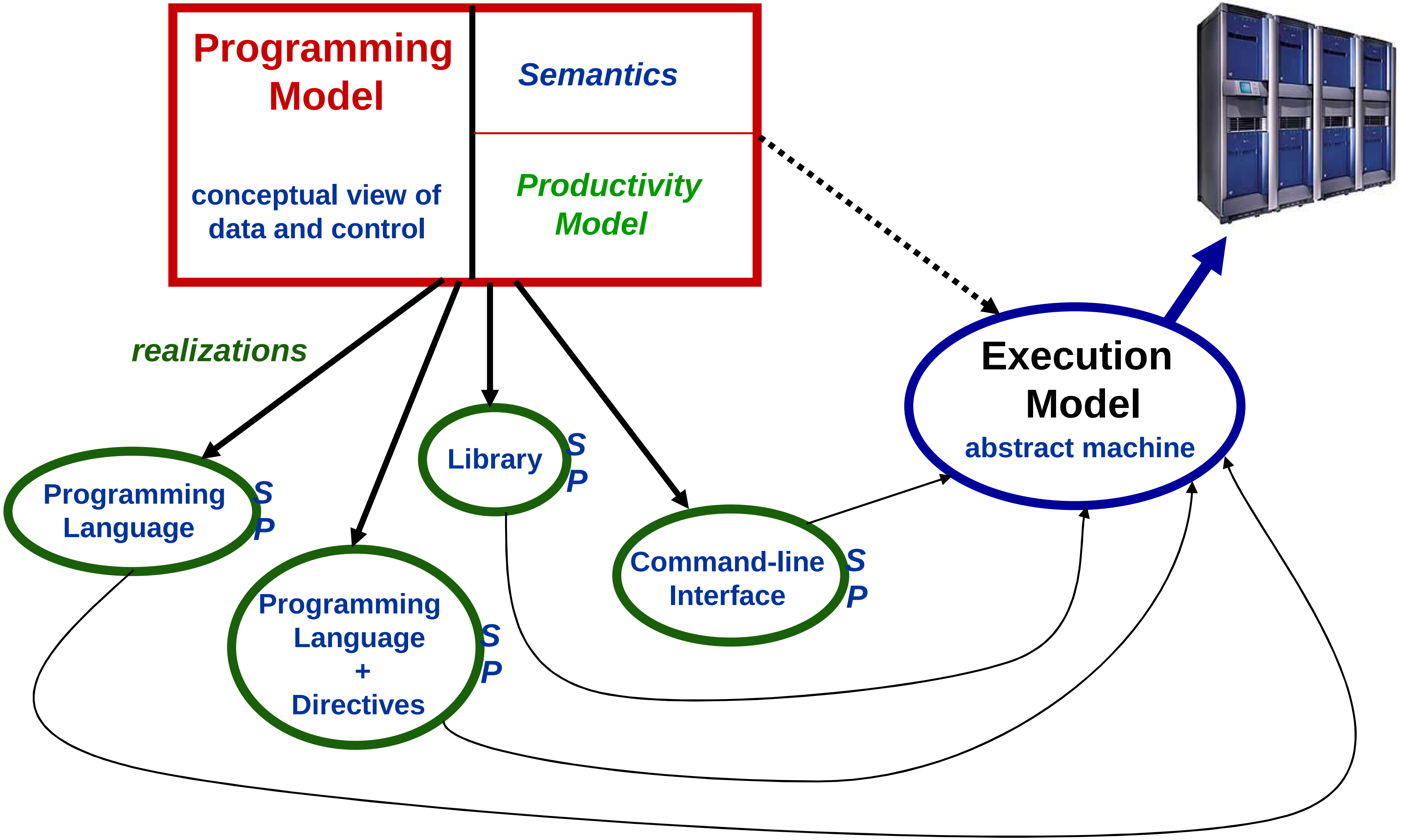
5 Conclusion

Productivity Challenges of Peta-Scale Systems

- **Large scale architectural parallelism**
  - *hundreds of thousands of processors*
  - *component failures may occur in relatively short intervals*
- **Extreme non uniformity in data access**
- **Applications are becoming larger and more complex**
  - *multi-disciplinary, multi-language, multi-paradigm*
  - *dynamic, irregular, and adaptive*
- **Legacy codes pose a problem: long lived applications, surviving many generations of hardware**
  - *from F77 to F90, C/C++, MPI, Coarray Fortran etc.*
  - *automatic re-write under constraint of performance portability is difficult*

Performance, user productivity, robustness, portability

Programming Models for High Productivity Computing



Programming Model

conceptual view of data and control

Semantics

Productivity Model

realizations

Library S P

Programming Language S P

Programming Language + Directives S P

Command-line Interface S P

Execution Model

abstract machine

Programming Model Issues

- **Programming models for high productivity computing and their realizations can be characterized along (at least) three dimensions:**
  - ***semantics***
  - ***user productivity (time to solution)***
  - ***performance***
- **Semantics: a mapping from programs to functions specifying input/output behavior of the program:**
  - ***S: P → F , where each f in F is a function f: I → O***
- **User productivity (programmability): a mapping from programs to a characterization of structural complexity:**
  - ***U: P → N***
- **Performance: a mapping from programs to functions specifying the complexity of the program in terms of its execution on a real or abstract target machine:**
  - ***C: P → G, where each g in G is a function g : I → N****

Contents

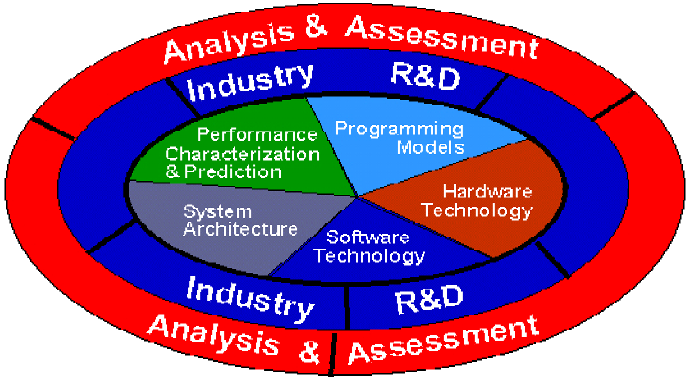
High Productivity Computing Systems

Goals:

- **Provide a new generation of economically viable high productivity computing systems for the national security and industrial user community (2007 – 2010)**

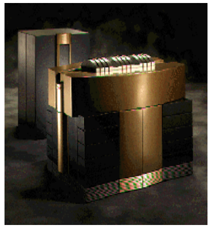
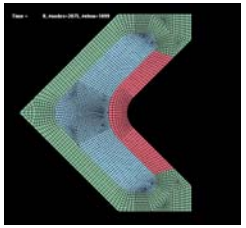
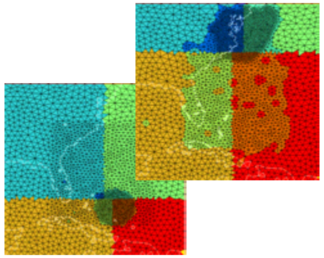
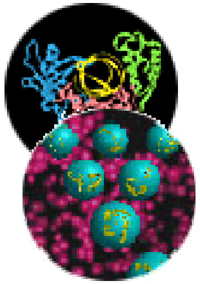
Impact:

- **Performance** (efficiency): critical national security applications by a factor of 10X to 40X
- **Productivity** (time-to-solution)
- **Portability** (transparency): insulate research and operational application software from system
- **Robustness** (reliability): apply all known techniques to **protect against outside attacks**, hardware faults, & programming errors



HPCS Program Focus Areas



Applications:

- Intelligence/surveillance, reconnaissance, cryptanalysis, airborne contaminant modeling and biotechnology

Fill the Critical Technology and Capability Gap

Today (late 80’s HPC technology)…..*to*…..Future (Quantum/Bio Computing)

The Cascade Project

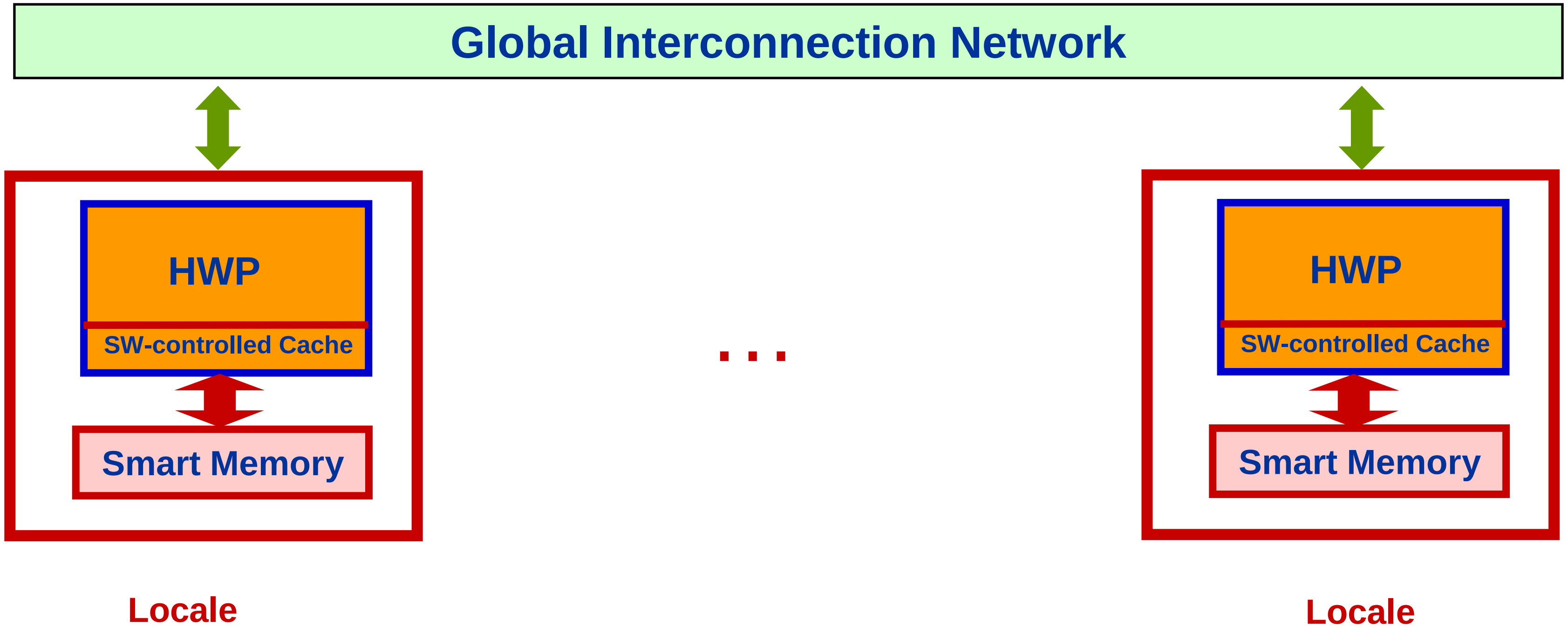
- One year Concept Study, July 2002 -- June 2003
- Three year Prototyping Phase, July 2003 -- June 2006
- Led by Cray Inc. (Burton Smith)
- **Partners**
  - Caltech/JPL
  - University of Notre Dame
  - Stanford University
- **Collaborators in the Programming Environment Area**

David Callahan, Brad Chamberlain, Mark James, John Plevyak

Key Elements of the Cascade Architecture

- High performance networks and multithreading contribute to tolerating memory latency and improving memory bandwidth
- Hardware support for locality aware programming and program-controlled selection of UMA/NUMA data access avoid serious performance problems present in current architectures
- Shared address space without global cache coherence eliminates a major source of bottlenecks
- Hierarchical two-level processing structure exploits temporal as well as spatial locality
- Lightweight processors in “smart memory” provide a computational fabric as well as an introspection infrastructure

A Simplified Global View of the Cascade Architecture



Global Interconnection Network

HWP

SW-controlled Cache

Smart Memory

Locale

. . .

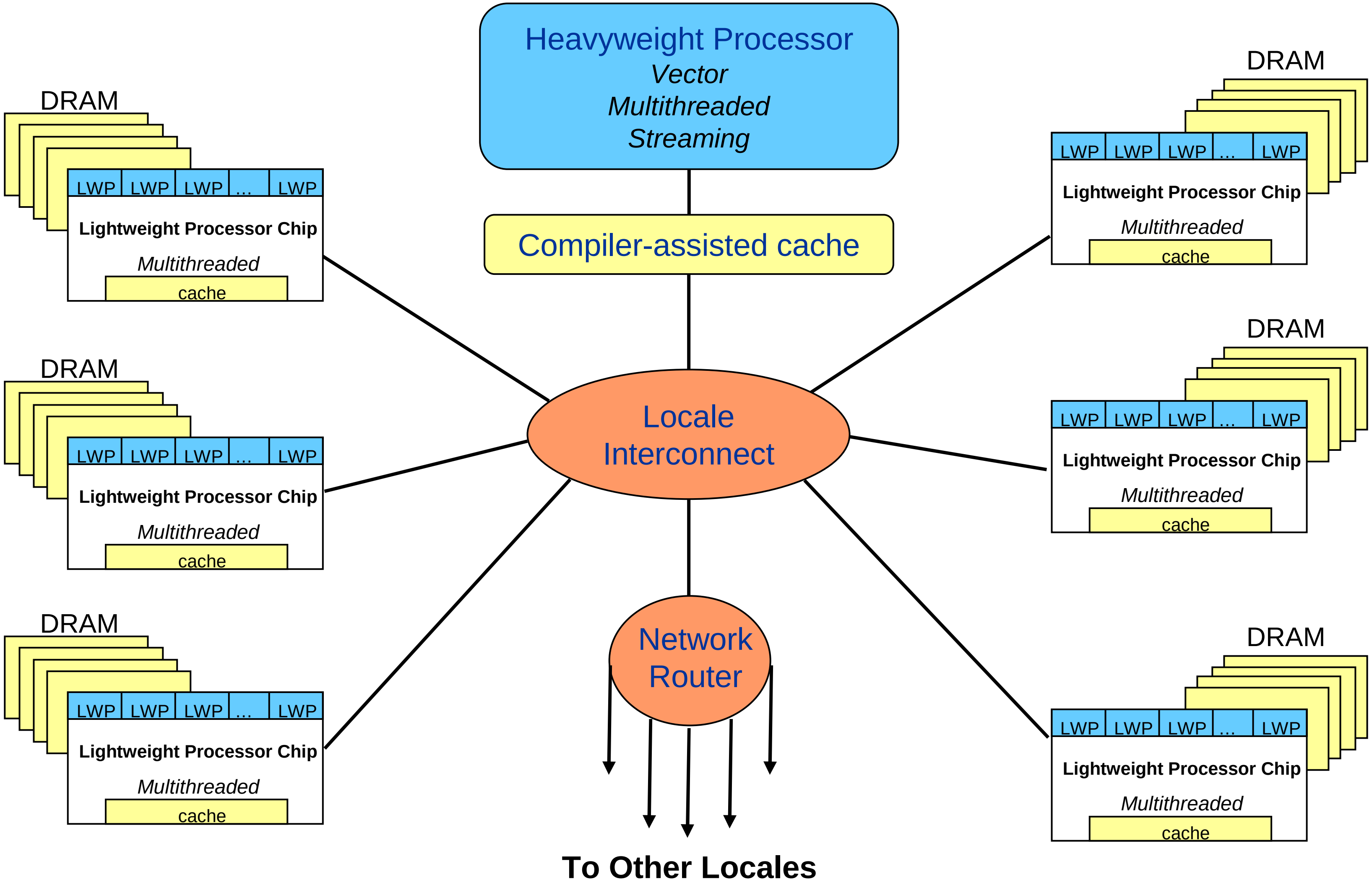
HWP

SW-controlled Cache

Smart Memory

Locale

A Cascade Locale



Source: David Callahan, Cray Inc.

Lightweight Processors and Threads

- **Lightweight processors**
  - *co-located with memory*
  - *focus on availability*
  - *full exploitation is not a primary system goal*
- **Lightweight threads**
  - *minimal state – high rate context switch*
  - *spawned by sending a parcel to memory*
- **Exploiting spatial locality**
  - *fine-grain: reductions, prefix operations, search*
  - *coarse-grain: data distribution and alignment*
- **Saving bandwidth by migrating threads to data**

Key Issues in High Productivity Languages

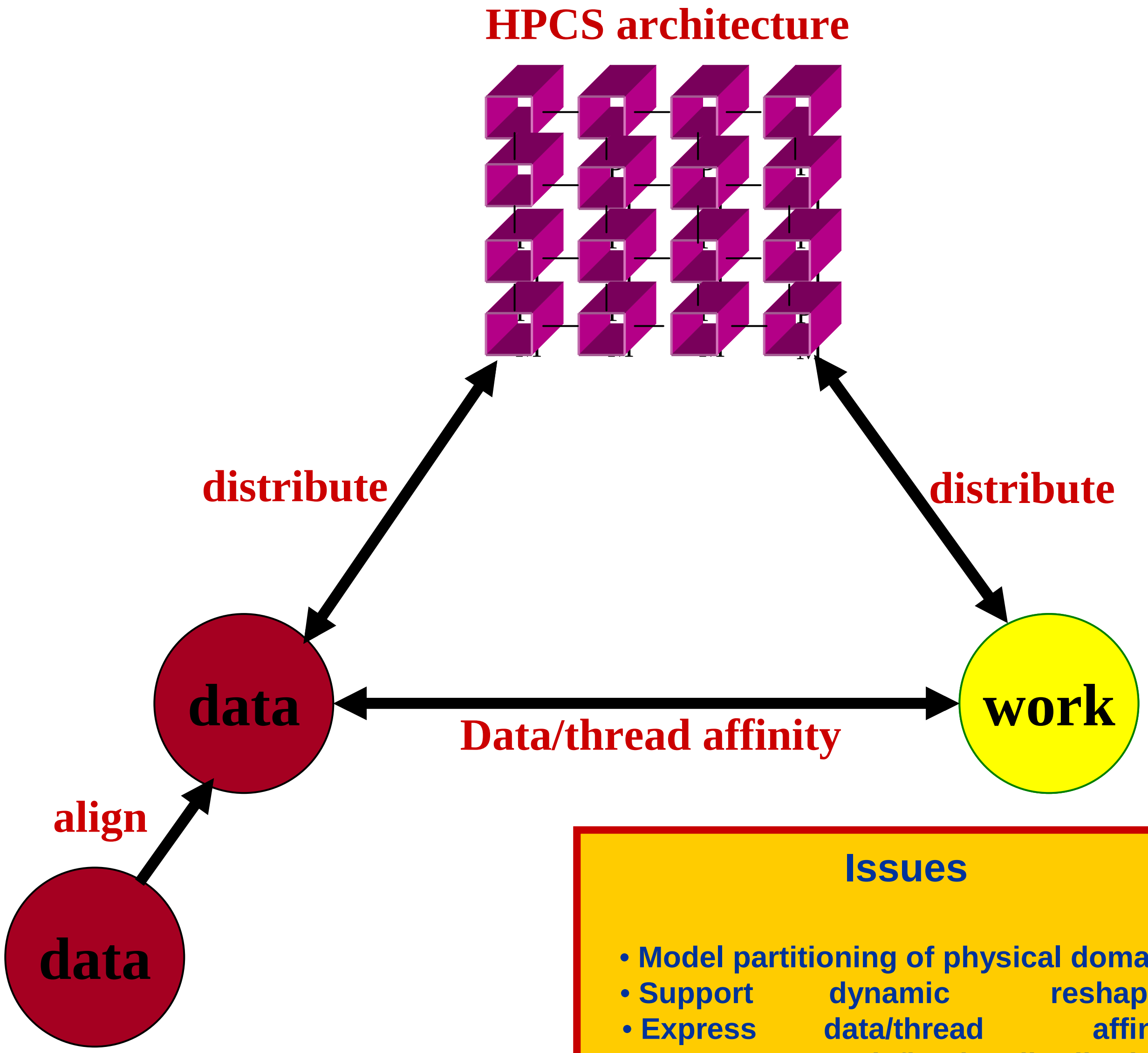
Critical Functionality

- High-Level Features for Explicit Concurrency
- High-Level Features for Locality Control
- High-Level Support for Distributed Collections
- High-Level Support for Programming-In-the-Large

Orthogonal Language Issues

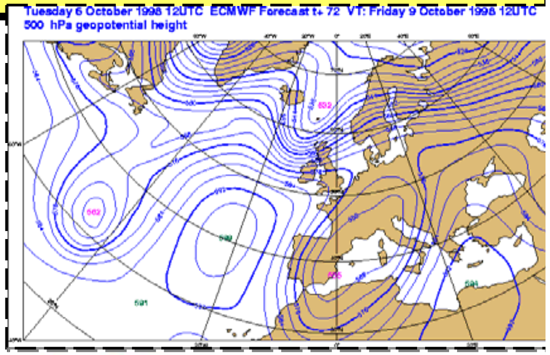
- global address space
- object orientation
- generic programming
- Extensibility
- Safety
- performance transparency

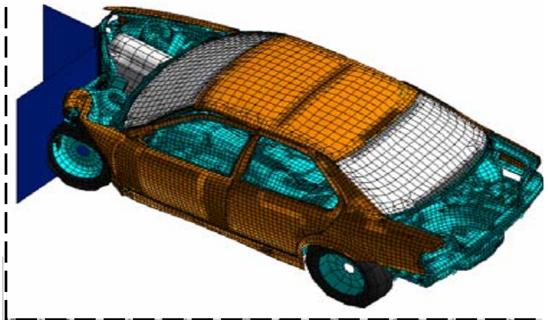
Locality Awareness: Distribution, Alignment, Affinity

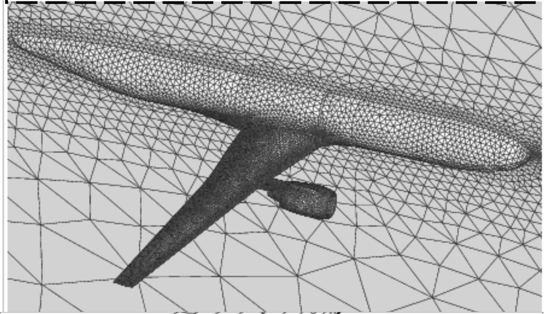


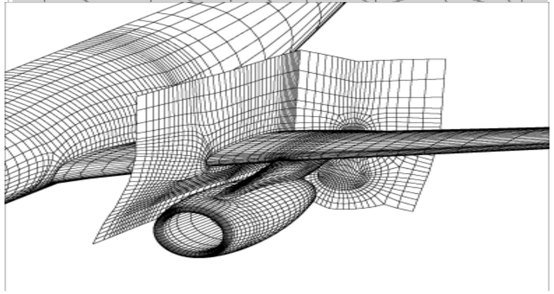
Issues

- Model partitioning of physical domains
- Support dynamic reshaping
- Express data/thread affinity
- Support user-defined distributions









Design Criteria of the "Chapel" Programming Language

- **Global name space**
  - *even in the context of a NUMA model*
  - *avoid "local view" programming model of MPI, Coarray Fortran, UPC*

- **Multiple models of parallelism**

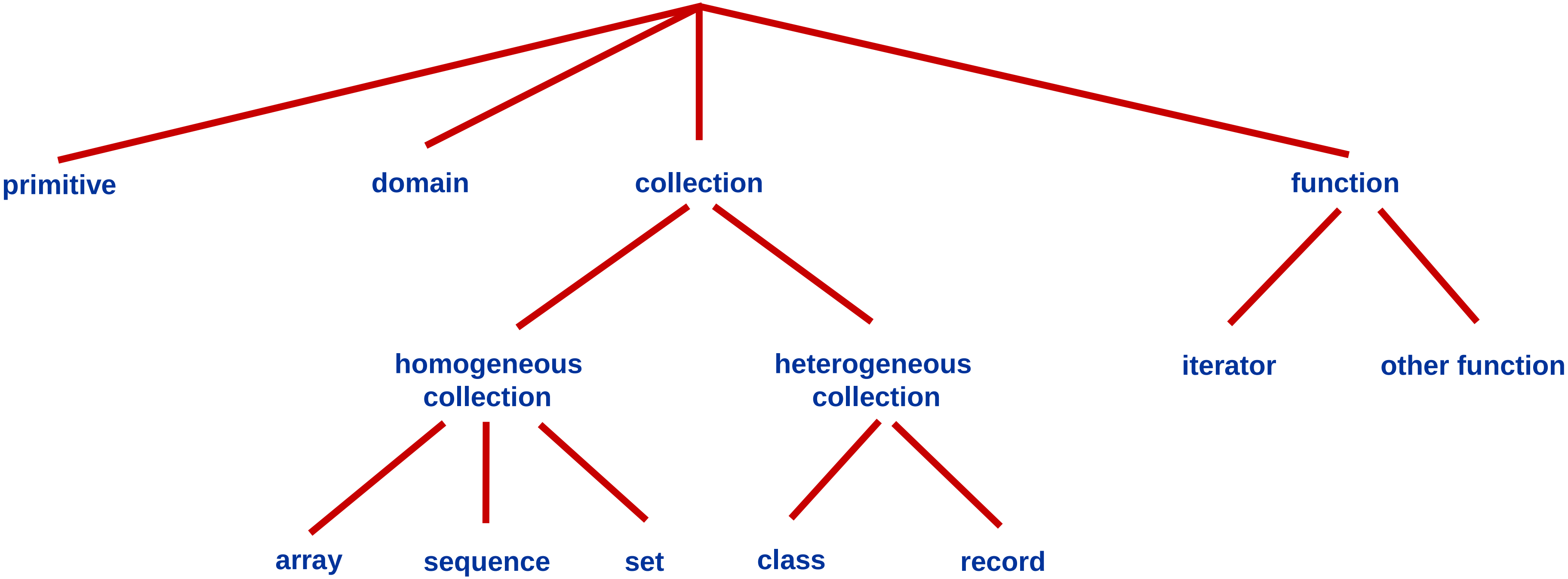
- **Provide support for:**
  - *explicit parallel programming*
  - *locality-aware programming*
  - *interoperability with legacy codes (MPI, Coarray Fortran, UPC, etc.)*
  - *generic programming*

Chapel Basics

- A modern base language
  - *Strongly typed*
  - *Fortran-like array features*
  - *Objected-oriented*
  - *Module structure for name space management*
  - *Optional automatic storage management*

- High performance features
  - *Abstractions for parallelism*
    - *data parallelism (domains, forall)*
    - *task parallelism (cobegin)*
  - *Locality management via data distributions and affinity*

Type Tree



primitive

domain

collection

function

homogeneous collection

heterogeneous collection

iterator

other function

array

sequence

set

class

record

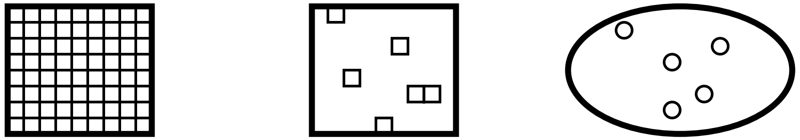
Language Design Highlights

- **The “Concrete Language” enhances HPF and ZPL**
  - *domains as first-class objects: index space, distribution, and associated set of arrays*
  - *generalized arrays and HPF-type data distributions*
  - *support for automatic partitioning of dynamic graph-based data structures*
  - *high-level control for communication (halos,…)*
  - *abstraction of iteration: generalization of the CLU iterator*

- **The “Abstract Language” supports generic programming**
  - *abstraction of types: type inference from context*
  - *data structure inference: system-selected implementation for programmer-specified object categories*
  - *specialization using cloning*

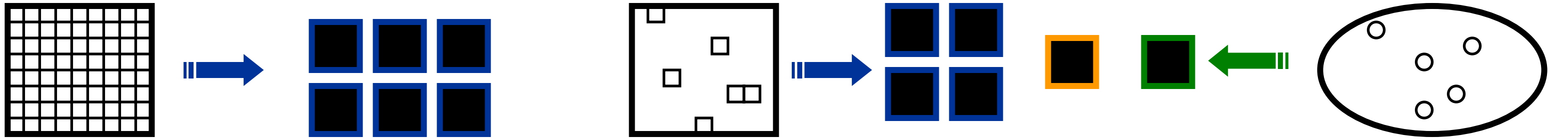
Domains

- *index sets: Cartesian products, sparse, opaque*

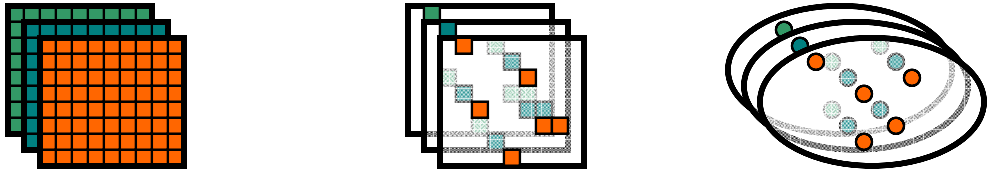


- *locale view*: a logical view for a set of locales

- *distribution*: a mapping of an index set to a locale view



- *array*: a map from an index set to a collection of variables



Source: Brad Chamberlain, Cray Inc.

Example: Matrix Vector Multiplication V1

```
var Mat:    domain(2) = [1..m, 1..n];
var MatCol: domain(1) = Mat(2);
var MatRow: domain(1) = Mat(1);

var A: array [Mat]    of float;
var v: array [MatCol] of float;
var s: array [MatRow] of float;

s = sum(dim=2) [i,j:Mat] A(i,j)*v(j);
```

Example: Matrix Vector Multiplication V2

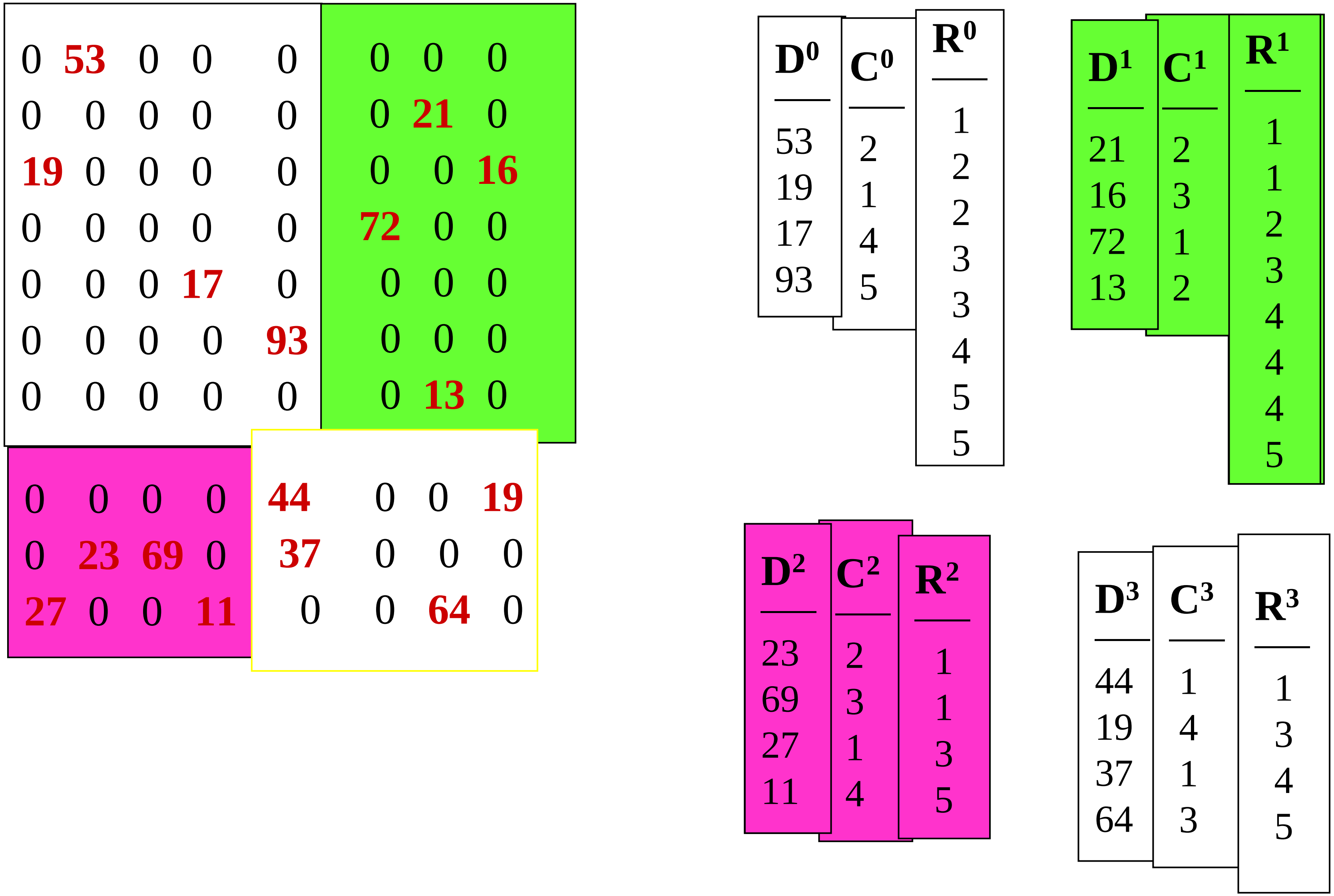
```
var L: array[1..p1,1..p2] of locale;

var Mat:    domain(2) dist(block,block) to L = [1..m,1..n];
var MatCol: domain(1) align(*,Mat(2)) = Mat(2);
var MatRow: domain(1) align(Mat(1),*) = Mat(1);

var A: array [Mat]    of float;
var v: array [MatCol] of float;
var s: array [MatRow] of float;

s = sum(dim=2) [i,j:Mat] A(i,j)*v(j);
```

Sparse Matrix Distribution



0	**53**	0	0	0	0	0	0
0	0	0	0	0	0	**21**	0
19	0	0	0	0	0	0	**16**
0	0	0	0	0	**72**	0	0
0	0	0	**17**	0	0	0	0
0	0	0	0	**93**	0	0	0
0	0	0	0	0	0	**13**	0
0	0	0	0	**44**	0	0	**19**
0	**23**	**69**	0	**37**	0	0	0
27	0	0	**11**	0	0	**64**	0

D^0	C^0	R^0
53	2	1
19	1	2
17	4	2
93	5	3
		3
		4
		5
		5

D^1	C^1	R^1
21	2	1
16	3	1
72	1	2
13	2	3
		4
		4
		4
		5

D^2	C^2	R^2
23	2	1
69	3	1
27	1	3
11	4	5

D^3	C^3	R^3
44	1	1
19	4	3
37	1	4
64	3	5

Example: Matrix Vector Multiplication V3

```
var L: array[1..p1,1..p2] of locale;

var Mat: domain(sparse2)
                dist(myspdist) to L
                layout(mysplay)
                = [1..m,1..n]
                where enumeratenonzeroes();

var MatCol: domain(1) align(*,Mat(2)) = Mat(2);
var MatRow: domain(1) align(Mat(1),*) = Mat(1);

var A: array [Mat]    of float;
var v: array [MatCol] of float;
var s: array [MatRow] of float;

s = sum(dim=2) [i,j:Mat] A(i,j)*v(j);
```

Language Summary

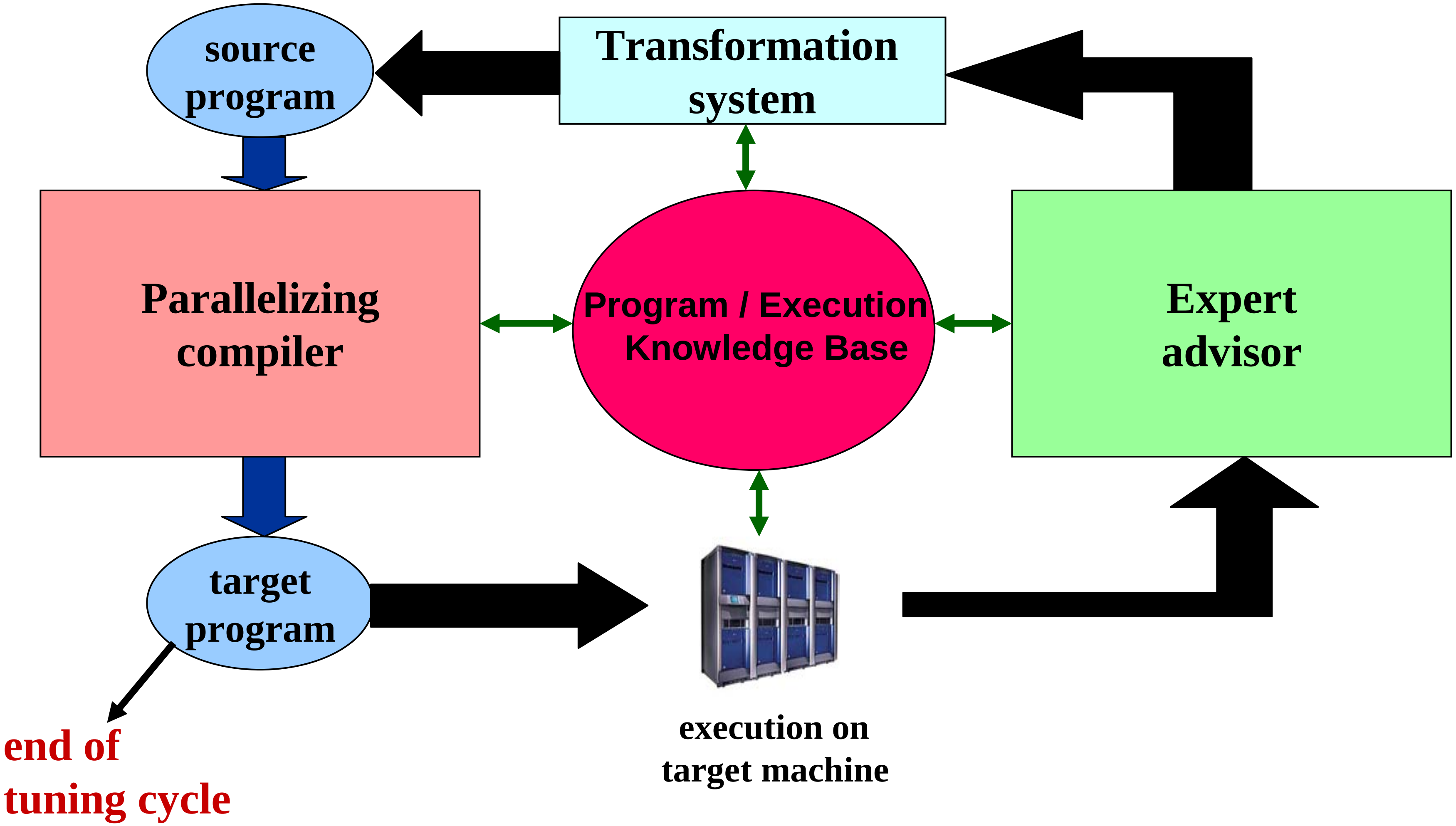
- Global name space
- High level control features supporting explicit parallelism
- High level locality management
- High level support for collections
- Static typing
- Support for generic programming

Contents

Issues in Programming Environments and Tools

- **Reliability Challenges**
  - *massive parallelism poses new problems*
  - *fault prognostics, detection, recovery*
  - *data distribution may cause vital data to be spread across all nodes*
- **(Semi) Automatic Tuning**
  - *closed loop adaptive control: measurement, decision-making, actuation*
  - *information exposure: users, compilers, runtime systems*
  - *learning from experience: databases, data mining, reasoning systems*
- **Introspection**
  - *a technology for support of validation, fault detection, performance tuning*

Example: Offline Performance Tuning



Legacy Code Migration

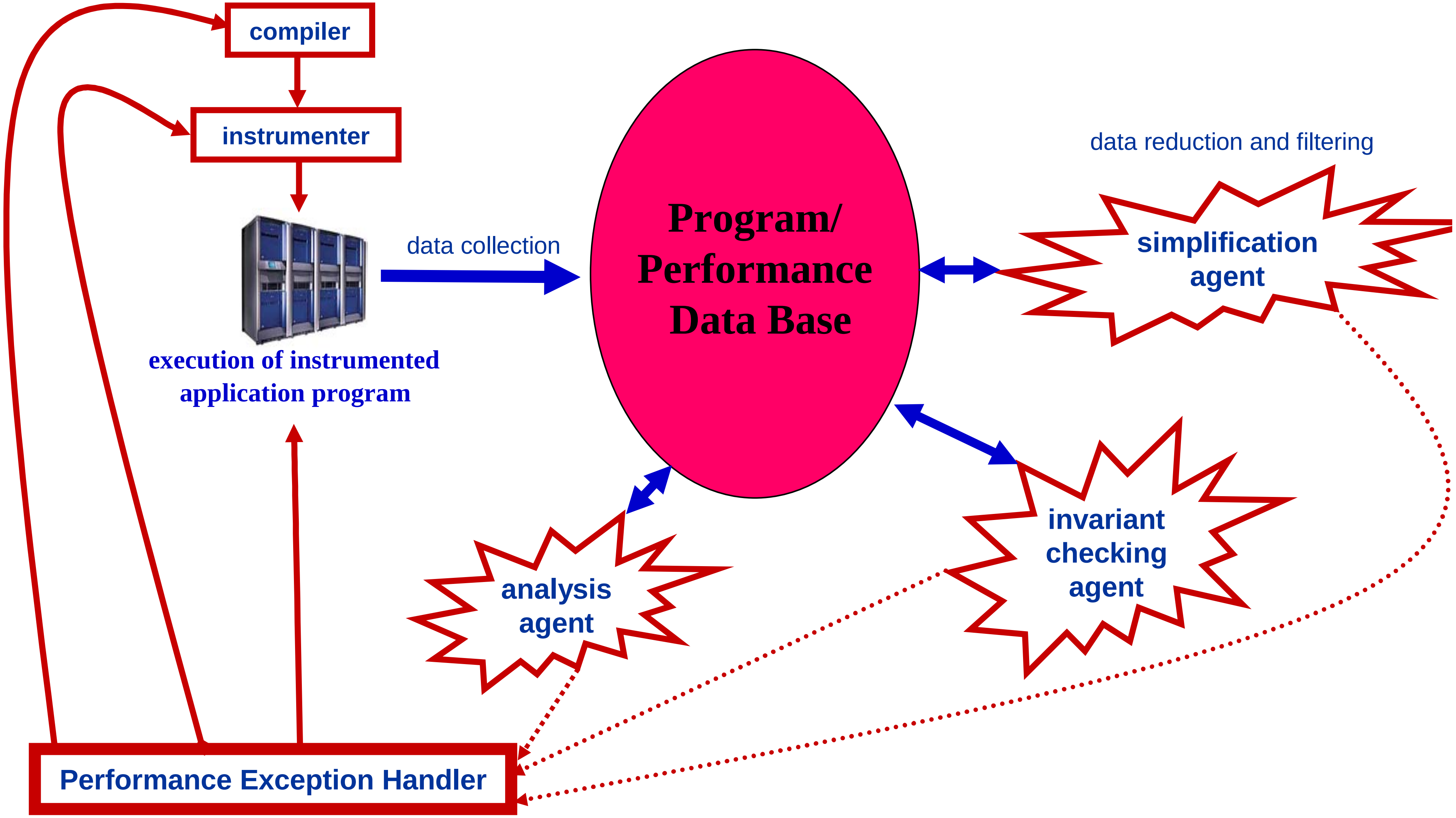
- **Rewriting Legacy Codes**
  - *preservation of intellectual content*
  - *opportunity for exploiting new hardware, including new algorithms*
  - *code size may preclude practicality of rewrite*

- **Language, compiler, tool, and runtime support**
  - *(Semi) automatic tools for migrating code*
  - *incremental porting*
  - *transition of performance-critical sections requires highly-sophisticated software for automatic adaptation*
    - *high-level analysis*
    - *pattern matching and concept comprehension*
    - *optimization and specialization*

Potential Uses of Lightweight Threads

- **Fine grain *application parallelism***
- **Implementation of a *service layer***
- **Components of agent systems that asynchronously *monitor the computation* performing introspection, and dealing with:**
  - ***dynamic program validation***
  - ***fault tolerance***
  - ***intrusion prevention and detection***
  - ***performance analysis and tuning***
  - ***support of feedback-oriented compilation***
- ***Introspection* can be defined as a system’s ability to:**
  - ***explore its own properties***
  - ***reason about its internal state***
  - ***make decisions about appropriate state changes where necessary***

Example: A Society of Agents for Performance Analysis and Feedback-Oriented Tuning



compiler

instrumenter

execution of instrumented application program

data collection

Program/ Performance Data Base

data reduction and filtering

simplification agent

invariant checking agent

analysis agent

Performance Exception Handler

Conclusion

- **Today’s programming languages, models, and tools cannot deal with 2010 architectures and application requirements**
- **Peta scale architectures will pose new challenges but may provide enhanced support for high level languages and compilers**
- **The Cascade programming language “Chapel” targets the creation of a viable language system together with a programming environment for economically feasible and robust high productivity computing of the future**

D.Callahan, B.Chamberlain, H.P.Zima: The Cascade High Productivity Language

Proceedings of the HIPS2004 Workshop, Santa Fe, New Mexico, April 2004