

# Chapel: Background

---

Sung-Eun Choi and Steve Deitz  
Cray Inc.

# Chapel Settings

- **HPCS: High Productivity Computing Systems (DARPA)**
  - Goal: Raise HEC user productivity by 10x
    - Productivity = Performance + Programmability + Portability + Robustness*
- Phase II: Cray, IBM, Sun (July 2003 – June 2006)
  - Evaluated entire system architecture
  - Three new languages (Chapel, X10, Fortress)
- Phase III: Cray, IBM (July 2006 – )
  - Implement phase II systems
  - Work continues on all three languages

# Chapel Productivity Goals

- Improve programmability over current languages
  - Writing parallel codes
  - Reading, changing, porting, tuning, maintaining, ...
- Support performance at least as good as MPI
  - Competitive with MPI on generic clusters
  - Better than MPI on more capable architectures
- Improve portability over current languages
  - As ubiquitous as MPI
  - More portable than OpenMP, UPC, CAF, ...
- Improve robustness via improved semantics
  - Eliminate common error cases
  - Provide better abstractions to help avoid other errors

# Outline

- Chapel's Settings and Goals
- Chapel's Themes
  - Global-view abstractions
  - General parallel programming
  - Multiple levels of design
  - Control of locality
  - Mainstream language features

# Global-View Abstractions

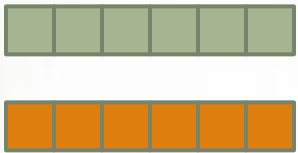
## Definitions

- **Programming model**  
*The mental model of a programmer*
- **Fragmented model**  
*Programmer takes point-of-view of a single processor/thread*
- **SPMD models** (Single Program, Multiple Data)  
*Fragmented models with multiple copies of one program*
- **Global-view model**  
*Programmer writes code to describe computation as a whole*

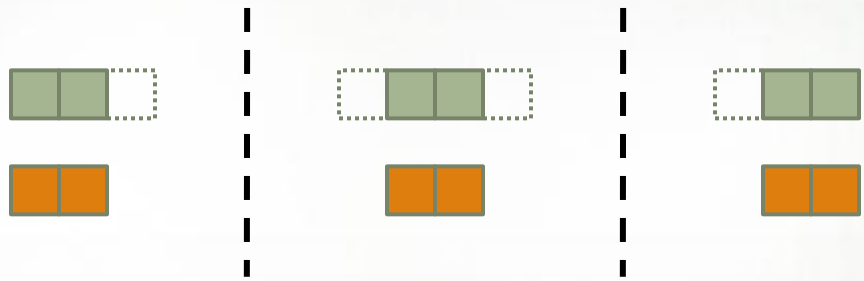
# Global-View Abstractions

## Example: 3-Point Stencil (Data Declarations)

*Global-View*

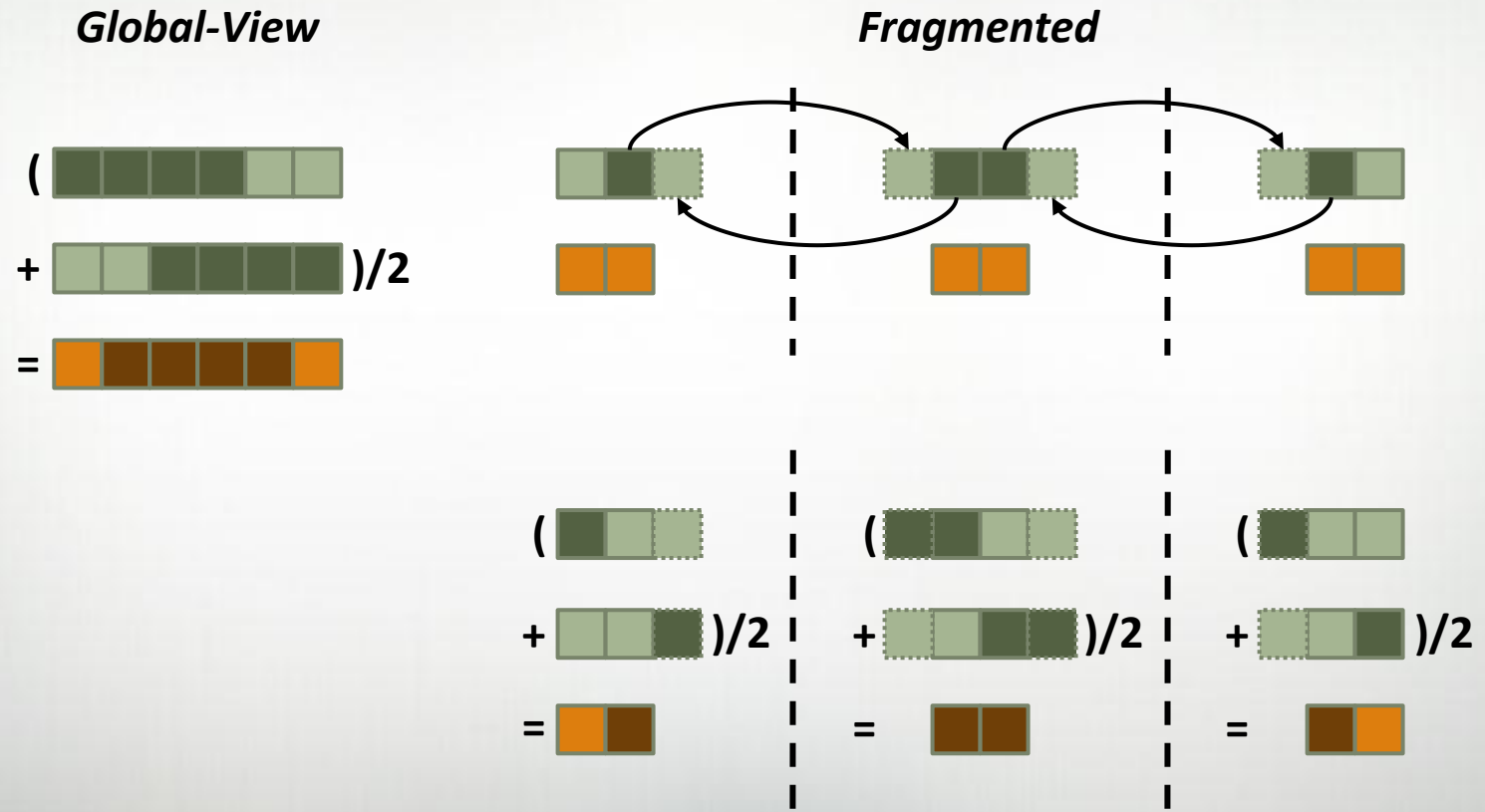


*Fragmented*



# Global-View Abstractions

## Example: 3-Point Stencil (Computation)




# Global-View Abstractions

## Example: 3-Point Stencil (Code)

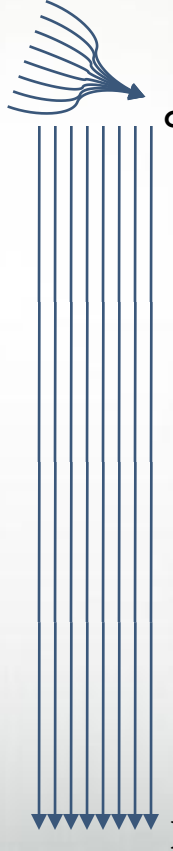
### Global-View

```
def main() {
  var n = 1000;
  var A, B: [1..n] real;

  forall i in 2..n-1 do
    B(i) = (A(i-1)+A(i+1))/2;
  }
```



### Fragmented



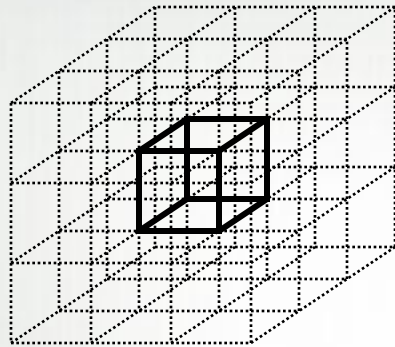
```
def main() {
  var n = 1000;
  var me = commRank(), p = commSize(),
      myN = n/p, myLo = 1, myHi = myN;
  var A, B: [0..myN+1] real;

  if me < p {
    send(me+1, A(myN));
    recv(me+1, A(myN+1));
  } else myHi = myN-1;
  if me > 1 {
    send(me-1, A(1));
    recv(me-1, A(0));
  } else myLo = 2;
  for i in myLo..myHi do
    B(i) = (A(i-1)+A(i+1))/2;
  }
```

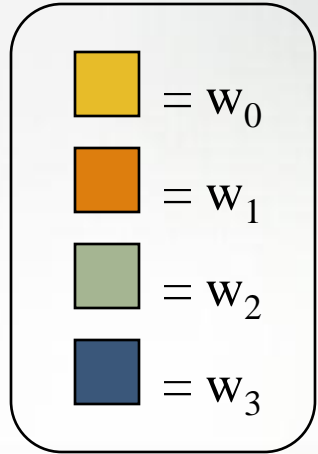
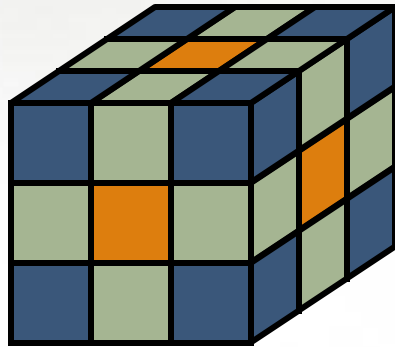
Assumes p divides n



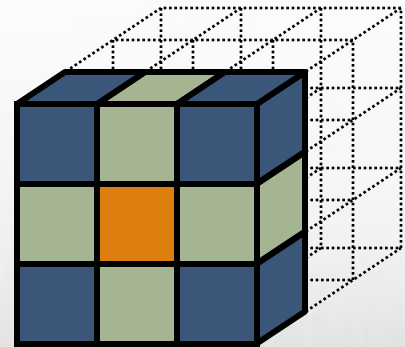
# NAS MG Stencil



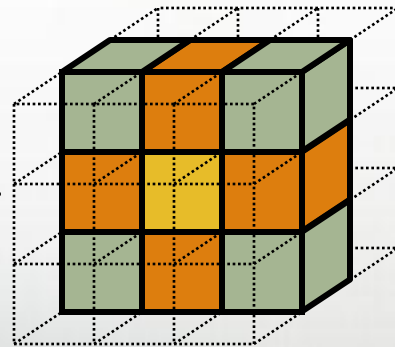
=



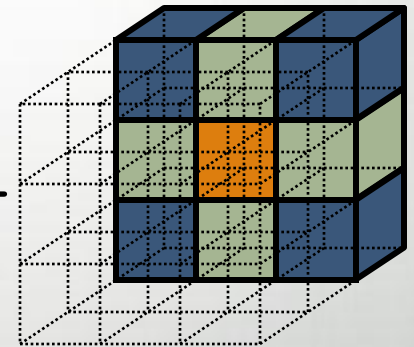
=



+



+





# NAS MG Stencil in Chapel

```

def rprj3(S, R) {
  const Stencil = [-1..1, -1..1, -1..1],
    W: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
    W3D = [(i,j,k) in Stencil] W((i!=0)+(j!=0)+(k!=0));

  forall inds in S.domain do
    S(inds) =
      + reduce [offset in Stencil] (W3D(offset) *
                                     R(inds + offset*R.stride));
}

```

Our previous work in ZPL has shown that such compact codes can result in better performance than the Fortran + MPI.

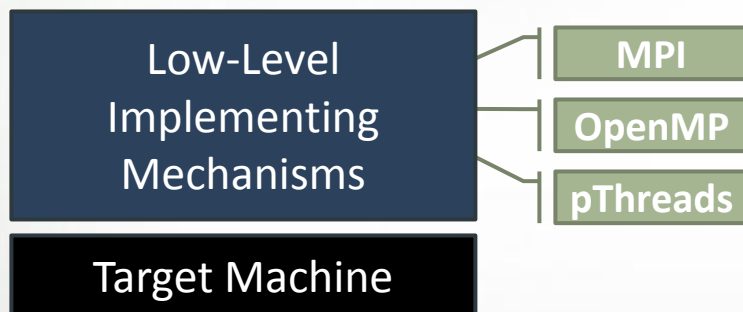
# Summary of Current Programming Systems

	System	Data Model	Compute Model
Communication Libraries	MPI/MPI-2	Fragmented	Fragmented
	SHMEM	Fragmented	Fragmented
	ARMCI	Fragmented	Fragmented
	GASNet	Fragmented	Fragmented
Shared Memory	OpenMP, pThreads	Global-View (trivially)	Global-View (trivially)
PGAS Languages	Co-Array Fortran	Fragmented	Fragmented
	UPC	Global-View	Fragmented
	Titanium	Fragmented	Fragmented
HPCS Languages	Chapel	Global-View	Global-View
	X10 (IBM)	Global-View	Global-View
	Fortress (Sun)	Global-View	Global-View

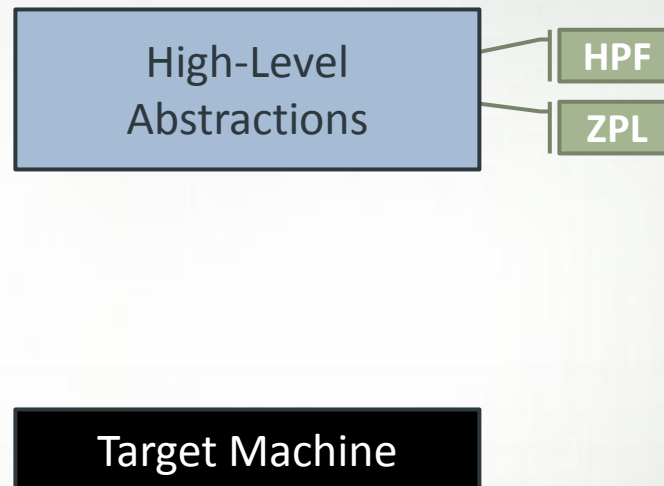
# General Parallel Programming

- Express all parallelism in the software
  - Forms: data, task, nested (arbitrary composition thereof)
  - Levels: module, function, loop, statement
  
- Target all parallelism in the hardware
  - Systems: multicore desktops, clusters, HPC systems
  - Types: multithreading, vector
  - Levels: across cores, across nodes, across systems

# Multiple Levels of Design

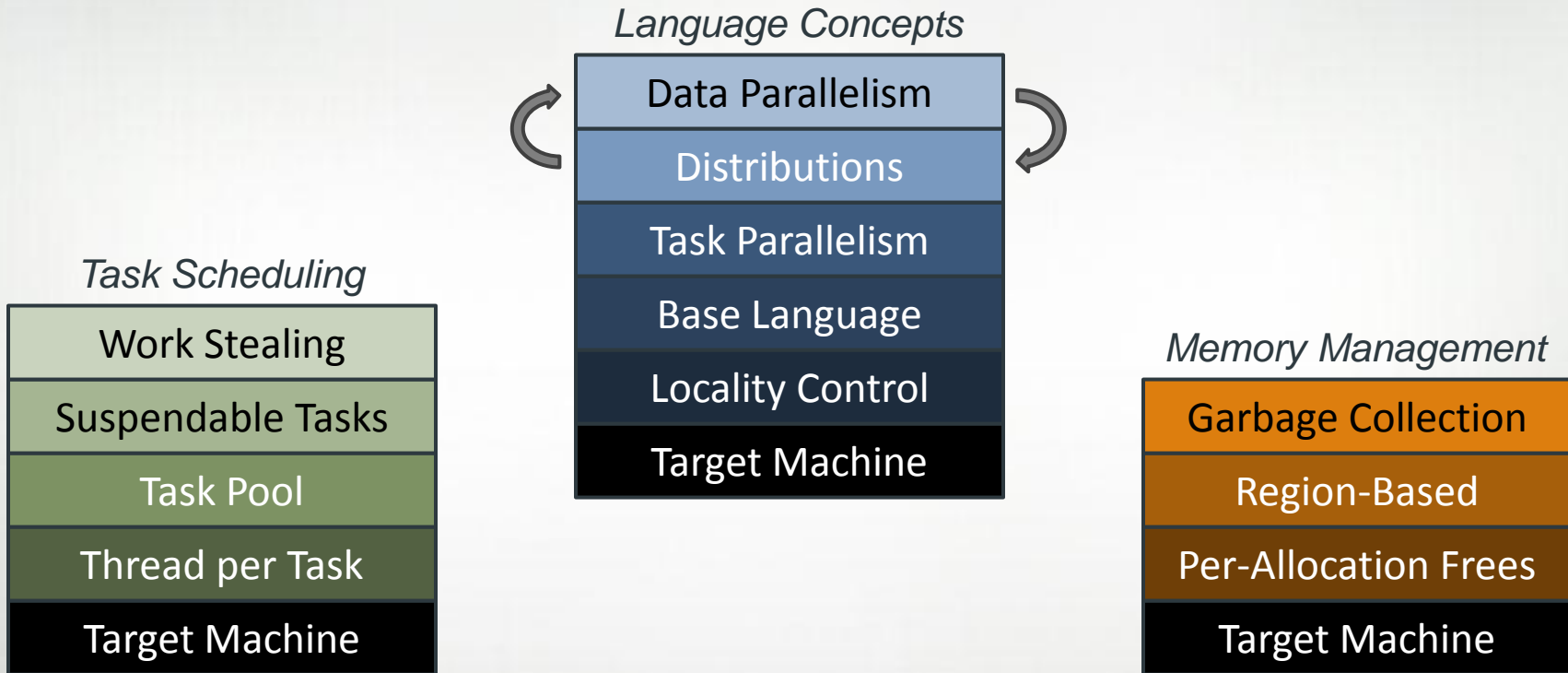


*“Why is everything so difficult?”*



*“Why can’t I optimize this?”*

# Multiple Levels of Design



# Control of Locality

## Given

- Scalable systems tend to store memory with processors
- Remote accesses tend to take longer than local accesses

## Therefore

- Placement of data relative to computation matters
- Programmers need control over data placement

## Note

- As multi-core chips grow, locality may matter on desktops
- GPUs/accelerators expose node-level locality



# Mainstream Language Features

- Object-oriented programming with value and reference classes
- Generic programming with types and compile-time constants
- Latent typing and a rich set of primitive types
- Modules for libraries and code organization
- Functions with nesting, overloading, and named arguments
- Multi-dimensional and associative arrays with slicing, etc.
- Classes, records, and unions
- Tuples, ranges, and domains
- Standard modules (*e.g.*, Math, Random, Time, BitOps, Norm)

# Questions?

- Chapel's Settings and Goals
- Chapel's Design
  - Global-view abstractions
  - General parallel programming
  - Multiple levels of design
  - Control of locality
  - Mainstream language features