Five Powerful Chapel Idioms

Steve Deitz
Cray Inc.
What is Chapel?

• A new parallel language
  • Under development at Cray Inc.
  • Supported through the DARPA HPCS program
  • Abstractions from ZPL, HPF, Cray XMT C, ...

• With many powerful idioms, features, and functions
  • Asynchronous and synchronous remote tasks
  • Data parallelism when applicable
  • User-defined distributions
  • Local and remote transactions
  • Arbitrarily nested parallelism
    ...

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
Chapel Design Concepts

- General parallel programming
  - Express all levels of software parallelism
  - Target all levels of hardware parallelism
- Partitioned Global Address Space (PGAS)
- Global-view abstractions
- Multiple levels of design
- Control of locality
- Mainstream language features
  - From scripting languages for fast prototyping
  - From object-oriented languages for robust designs
Chapel Parallel Programming Model

- Single task executes main() on Locale 0

- Advantages over SPMD
  - Single (global) flow of control
  - Fragmentation of problem is unnecessary (though possible)
Easy Invocation of Remote Asynchronous and Synchronous Tasks
Easy Invocation of Remote Tasks

• Syntax

```
on-statement:
on expression statement
```

• Semantics

• Evaluates expression to determine locale
• Executes statement on locale

• Example

```
on object { update(object); }
on A(i) { A(i) = B(i) + f(i); }
```
Easy Invocation of Asynchronous Tasks

• Syntax

```
begin-statement:
    begin statement
```

• Semantics

• Executes statement in a concurrent task
• Control continues immediately to next statement

• Example

```
sync {
    begin f1();
    f2();
}
```
Easy Invocation of Remote Asynchronous Tasks

```c
begin on A(i) {
    A(i) += f(i);
}
```
Contrasted depictions of a 3-point stencil

**Global-view**

\[
\begin{align*}
( & + & )/2 \\
\end{align*}
\]

**Fragmented**

\[
\begin{align*}
( & + & )/2 \\
\end{align*}
\]
Global-View vs. Fragmented Code

Contrasted codes of a 3-point stencil

**Global-view**

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

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

**Fragmented**

```python
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

\[ w_0 = w_1 = w_2 = w_3 \]
NAS MG Stencil in Fortran + MPI

```fortran
use self_intronics
implicit none
include ' cafnpb.h '
integer n1, n2, n3
double precision u(n1,n2,n3)
integer axis

if( axis .eq. 1 ) then
  call comm1p(axis, u, n1, n2, n3, k)
  else if( axis .eq. 2 ) then
    buff(1:buff_len, buff_id+1)
    do i3=2,n3-1
      buff(1:buff_len, buff_id)
    enddo
    do i2=2,n2-1
      u(i1,i2,1) = buff(1:buff_len, buff_id+1)
      u(i1,i2,1) = buff(1:buff_len, buff_id)
    enddo
  else if( axis .eq. 3 ) then
    buff(1:buff_len, buff_id)
    do i3=2,n3-1
      buff(1:buff_len, buff_id+1)
    enddo
    do i2=2,n2-1
      u(i1,1,i3) = buff(1:buff_len, buff_id)
      u(i1,1,i3) = buff(1:buff_len, buff_id)
    enddo
  endif
endif

if( dir .eq. -1 ) then
  if( axis .eq. 1 ) then
    d1 = 1
    else if( axis .eq. 2 ) then
      d2 = 1
      else if( axis .eq. 3 ) then
        d3 = 1
        endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
end
```
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));
}

Previous work shows performance is still possible:

Support for User-Defined Distributions
A “recipe” for distributed arrays that...
Instructs the compiler how to Map the global view...

...to a fragmented, per-processor implementation
Domains are associated to a distribution

```javascript
const Dist = new Block(rank=2, bbox=[1..4, 1..8]);

var Dom: domain(2) distributed Dist = [1..4, 1..8];
```

The distribution defines:

- Ownership of domain indices and array elements
- Default distribution of work (task-to-locale map)

E.g., forall loops over distributed domains/arrays
Authoring Distributions

- (Advanced) programmers can write distributions
- Built-in library of distributions
  - No extra compiler support for built-in distributions
  - Compiler uses structural interface:
    - Create domains and arrays
    - Map indices to locales
    - Access array elements
    - Iterate over indices/elements sequentially, in parallel, zippered
    - ...
- Distributions are built using language-level concepts
  - On for data and task locality
  - Begin, cobegin, and coforall for data parallelism
All domain types can be distributed. Semantics are independent of distribution. (Though performance and parallelism will vary...)
2009 Summer Internship: Albert Sidelnik from UIUC

- Added a distribution that maps data to GPUs
- Changed distribution of domain for HPCC STREAM
- Minor compiler changes (to generate CUDA, etc.)

```plaintext
const Dist = new GPUDist(rank=1, tpb=256);

const Dom: domain(1) distributed Dist = [1..m];

var A, B, C: [Dom] real;

forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```
Support for Local and Remote Transactions
Atomic Transactions (Work in Progress)

- **Syntax**

  ```
  atomic-statement:
    atomic statement
  ```

- **Semantics**

  - Executes statement as if it is a single operation
  - No other task sees a partial result

- **Example**

  ```java
  atomic A(i) = A(i) + 1;
  ```

```java
atomic {
    newNode.next = node;
    newNode.prev = node.prev;
    node.prev.next = newNode;
    node.prev = newNode;
}
```
Support for Arbitrarily Nested Parallelism
Example of invoking two data-parallel tasks

```plaintext
sync {
    begin A = B + alpha * C;
    begin D = E + alpha * F;
}
```
More Chapel...

- Full day tutorial
  Upcoming joint tutorial with X10 and UPC at SC ’09
- Download the release
  http://sourceforge.net/projects/chapel/
- Contact us
  Send us mail at chapel_info@cray.com
  Visit our web page at http://chapel.cray.com/
  View archives of chapel-users@lists.sourceforge.net
- Position paper
  http://chapel.cray.com/LACSS09_DEITZ.pdf