Parallel Programming in Chapel

LACSI 2006
October 18th, 2006

Steve Deitz
Chapel project, Cray Inc.
Why is Parallel Programming Hard?

- Partitioning of data across processors
- Partitioning of tasks across processors
- Inter-processor communication
- Load balancing
- Race conditions
- Deadlock
- And more ...
How is it getting easier?

- Support for new global address space languages
  - Co-Array Fortran
    - Modest extension to Fortran 95 for SPMD parallelism
    - Developed at Cray
    - Accepted as part of next Fortran
  - Unified Parallel C (UPC)
    - Small extension to C for SPMD parallelism

- Development of new high-level parallel languages
  - Chapel
    - Under development at Cray (part of HPCS)
    - Builds on MTA C, HPF, and ZPL
    - Provides modern language features (C#, Java, …)
  - Fortress and X10
HPCS = High Productivity Computing Systems (a DARPA program)

Overall Goal: Increase productivity for High-End Computing (HEC) users by 10× for the year 2010

Productivity = Programmability
+ Performance
+ Portability
+ Robustness

Result must be…
…revolutionary, not evolutionary
…marketable to users beyond the program sponsors

Phase II Competitors (July `03 – June`06): Cray, IBM, Sun
Outline

- Chapel Features
  - Domains and Arrays
  - Cobegin, Begin, and Synchronization Variables

- 3-Point Stencil Case Study
  - MPI
  - Co-Array Fortran
  - Unified Parallel C
  - Chapel
Domains and Arrays

- **Domain**: an index set
  - Specifies size and shape of arrays
  - Supports sequential and parallel iteration
  - Potentially decomposed across locales

- Three main classes:
  - **Arithmetic**: indices are Cartesian tuples
    - Rectilinear, multidimensional
    - Optionally strided and/or sparse
  - **Indefinite**: indices serve as hash keys
    - Supports hash tables, associative arrays, dictionaries
  - **Opaque**: indices are anonymous
    - Supports sets, graph-based computations

- Fundamental Chapel concept for data parallelism
- A generalization of ZPL’s *region* concept
A Simple Domain Declaration

```plaintext
var m: int = 4;
var n: int = 8;

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

\[ D \]
A Simple Domain Declaration

\[\begin{align*}
\text{var } m &: \text{ int } = 4; \\
\text{var } n &: \text{ int } = 8; \\
\text{var } D &: \text{ domain}(2) = [1..m, 1..n]; \\
\text{var } D\text{Inner} &: \text{ subdomain}(D) = [2..m-1, 2..n-1];
\end{align*}\]
Domain Usage

- **Declaring arrays:**
  \[ \text{var } A, B: [D] \text{ float; } \]

- **Sub-array references:**
  \[ A(D_{\text{Inner}}) = B(D_{\text{Inner}}); \]

- **Sequential iteration:**
  \[ \text{for } i, j \text{ in } D_{\text{Inner}} \{ \ldots A(i, j) \ldots \} \]
  \[ \text{or: for } i j \text{ in } D_{\text{Inner}} \{ \ldots A(i j) \ldots \} \]

- **Parallel iteration:**
  \[ \text{forall } i j \text{ in } D_{\text{Inner}} \{ \ldots A(i j) \ldots \} \]
  \[ \text{or: [ij in } D_{\text{Inner}} \] \ldots A(i j)\ldots \]

- **Array reallocation:**
  \[ D = [1..2m, 1..2n]; \]
\textbf{Other Arithmetic Domains}

\begin{verbatim}
\textbf{var} StridedD: \textit{subdomain}(D) = D \textbf{by} (2,3);
\end{verbatim}

\begin{verbatim}
\textbf{var} SparseD: \textit{sparse domain}(D) =
(/(1,1),(4,1),(2,3),(4,2), \ldots /);
\end{verbatim}
Cobegin

- Executes statements in parallel
- Synchronizes at join point

```cobegin
    { 
      leftSum = computeLeft();
      rightSum = computeRight();
    } sum = leftSum + rightSum;
```

- Composable with data parallel abstractions

```coq
  var D: domain(1) distributed(Block) = [1..n];
  var A: [D] float;

  ...

  cobegin {
    [i in 1..n/2] computeLeft(A(i));
    [i in n/2+1..n] computeRight(A(i));
  }
```
Synchronization Variables

- Implements produce/consumer idioms

```haskell
cobegin {
    var s: sync int = computeProducer();
    computeConsumer(s);
}
```

- Subsumes locks

```haskell
var lock: sync bool;

lock = true;
    compute();
lock;
```

- Implements futures
◆ Spawns a concurrent computation
◆ No synchronization

```haskell
var leftSum: sync int;
begin leftSum = computeLeft();
rightSum = computeRight();
...
sum = leftSum + rightSum;
```

◆ Subsumes SPMD model

```haskell
for locale in Locales do
begin myMain(locale);
```
More Chapel Features

- Locales and user-defined distributions
- Objects (classes, records, and unions)
- Generics and local type inference
- Tuples
- Sequences and iterators
- Reductions and scans (parallel prefix)
- Default arguments, name-based argument passing
- Function and operator overloading
- Modules (namespace management)
- Atomic transactions
- Automatic garbage collection
Outline

- Chapel Features
  - Domains and Arrays
  - Cobegin, Begin, and Synchronization Variables

- 3-Point Stencil Case Study
  - MPI
  - Co-Array Fortran
  - Unified Parallel C
  - Chapel
Compute the counting numbers via iterative averaging

<table>
<thead>
<tr>
<th>Initial condition</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Iteration 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Iteration 3</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
<td>2.5</td>
<td>4</td>
</tr>
<tr>
<td>Iteration 4</td>
<td>0</td>
<td>0.5</td>
<td>1.5</td>
<td>2.5</td>
<td>4</td>
</tr>
<tr>
<td>Iteration 5</td>
<td>0</td>
<td>0.75</td>
<td>1.5</td>
<td>2.75</td>
<td>4</td>
</tr>
<tr>
<td>Iteration 6</td>
<td>0</td>
<td>0.75</td>
<td>1.75</td>
<td>2.75</td>
<td>4</td>
</tr>
<tr>
<td>\vdots</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration K</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Array Allocation

C + MPI

\[
\begin{align*}
\text{my}_n &= n \times (\text{me}+1)/p - n\times\text{me}/p; \\
A &= (\text{double } \*)\text{malloc}((\text{my}_n+2)\times\text{sizeof(double)});
\end{align*}
\]

Co-Array Fortran

\[
\begin{align*}
\text{MY}_N &= N\times\text{ME}/P - N\times(\text{ME}-1)/P \\
\text{ALLOCATE} & (A(0:\text{MY}_N+1)[*])
\end{align*}
\]

Unified Parallel C

\[
\begin{align*}
A &= (\text{shared double } \*)\text{upc_all_alloc}((n+2), \text{sizeof(double)});
\end{align*}
\]

Chapel

\[
\begin{align*}
\text{var } D & : \text{domain}(1) \text{ distributed(Block) } = [0..n+1]; \\
\text{var } \text{InnerD} & : \text{subdomain}(D) = [1..n]; \\
\text{var } A & : [D] \text{ float};
\end{align*}
\]
Array Initialization

**C + MPI**

```c
for (i = 0; i <= my_n; i++)
    A[i] = 0.0;
if (me == p - 1)
    A[my_n+1] = n + 1.0;
```

**Co-Array Fortran**

```fortran
A(:) = 0.0
IF (ME .EQ. P) THEN
    A(MY_N+1) = N + 1.0
ENDIF
```

**Unified Parallel C**

```c
upc_forall (i = 0; i <= n; i++; &A[i])
    A[i] = 0.0;
if (MYTHREAD == THREADS-1)
    A[n+1] = n + 1.0;
```

**Chapel**

```chapel
A(0..n) = 0.0;
A(n+1) = n + 1.0;
```
### Stencil Computation

#### C + MPI

```c
if (me < p-1)
    MPI_Send(&(A[my_n]), 1, MPI_DOUBLE, me+1, 1, MPI_COMM_WORLD);
if (me > 0)
    MPI_Recv(&(A[0]), 1, MPI_DOUBLE, me-1, 1, MPI_COMM_WORLD, &status);
if (me > 0)
    MPI_Send(&(A[1]), 1, MPI_DOUBLE, me-1, 1, MPI_COMM_WORLD);
if (me < p-1)
    MPI_Recv(&(A[my_n+1]), 1, MPI_DOUBLE, me+1, 1, MPI_COMM_WORLD, &status);
for (i = 1; i <= my_n; i++)
    Tmp[i] = (A[i-1]+A[i+1])/2.0;
```

#### Co-Array Fortran

```fortran
CALL SYNC_ALL()
IF (ME .LT. P) THEN
    A(MY_N+1) = A(1)[ME+1]
    A(0)[ME+1] = A(MY_N)
ENDIF
CALL SYNC_ALL()
DO I = 1,MY_N
    TMP(I) = (A(I-1)+A(I+1))/2.0
ENDDO
```

#### Unified Parallel C

```c
upc_barrier;
upc_forall (i = 1; i <= n; i++; &A[i])
    Tmp[i] = (A[i-1]+A[i+1])/2.0;
```

#### Chapel

```chapel
forall i in InnerD do
    Tmp(i) = (A(i-1)+A(i+1))/2.0;
```
C + MPI

```c
my_delta = 0.0;
for (i = 1; i <= my_n; i++)
    my_delta += fabs(A[i]-Tmp[i]);
MPI_Allreduce(&my_delta, &delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Co-Array Fortran

```fortran
MY_DELTA = 0.0
DO I = 1,MY_N
    MY_DELTA = MY_DELTA + ABS(A(I)-TMP(I))
ENDDO
CALL SYNC_ALL()
DELTA = 0.0
DO I = 1,P
    DELTA = DELTA + MY_DELTA[I]
ENDDO
```

Unified Parallel C

```c
my_delta = 0.0;
upc_forall (i = 1; i <= n; i++; &A[i])
    my_delta += fabs(A[i]-Tmp[i]);
upc_lock(lock);
delta = delta + my_delta;
upc_unlock(lock);
```

Chapel

```chapel
delta = + reduce abs(A(InnerD)-Tmp(InnerD));
```
Reduction

C + MPI

```c
my_delta = 0.0;
for (i = 1; i <= my_n; i++)
    my_delta += fabs(A[i]-Tmp[i]);
MPI_Allreduce(&my_delta, &delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Co-Array Fortran

```fortran
MY_DELTA = 0.0
DO I = 1,MY_N
    MY_DELTA = MY_DELTA + ABS(A(I)-TMP(I))
ENDDO
CALL SYNC_ALL()
DELTA = 0.0
DO I = 1,P
    DELTA = DELTA + MY_DELTA[I]
ENDDO
```

Unified Parallel C

```c
my_delta = 0.0;
upc_forall (i = 1; i <= n; i++; &A[i])
    my_delta += fabs(A[i]-Tmp[i]);
upc_lock(lock);
delta = delta + my_delta;
upc_unlock(lock);
```

Chapel

```chapel
delta = 0.0;
forall i in InnerD do
    atomic delta += abs(A(i)-Tmp(i));
```
C + MPI

```c
if (me == 0)
    printf("Iterations: %d\n", iters);
```

Co-Array Fortran

```fortran
IF (ME .EQ. 1) THEN
    WRITE(*,*) 'Iterations: ', ITERS
ENDIF
```

Unified Parallel C

```c
if (MYTHREAD == 0)
    printf("Iterations: %d\n", iters);
```

Chapel

```chapel
writeln("Iterations: ", iters);
```
### Case Study Summary

<table>
<thead>
<tr>
<th>desirable features</th>
<th>C+MPI</th>
<th>CAF</th>
<th>UPC</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td>global view of computation</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>global view of data</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>data distributions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>arbitrary data distributions</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>multi-dimensional arrays</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>syntactic execution model</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>one-sided communication</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>support for reductions</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>simple mechanisms</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Implementation Status

- Serial implementation
  - Covers over 95% of features in Chapel
  - Prototype quality
- Non-distributed threaded implementation
  - Implements sync variables, single variables, begin, and cobegin
http://chapel.cs.washington.edu/

chapel_info@cray.com
deitz@cray.com