Chapel:
High-Productivity Parallel Computing

Brad Chamberlain
Chapel Team

Google
September 19, 2007
Chapel

*Chapel*: a new parallel language being developed by Cray Inc.

**Themes:**

- **general parallelism**
  - data-, task-, nested parallelism using *global-view* abstractions
  - general parallel architectures

- **locality control**
  - data distribution
  - task placement (typically data-driven)

- **reduce gap between mainstream and parallel languages**
  - object-oriented programming (OOP)
  - type inference and generic programming
Chapel’s Setting: HPCS

**HPCS:** High *Productivity* Computing Systems (DARPA et al.)
- **Goal:** Raise HEC user productivity by $10 \times$ for the year 2010
- **Productivity** = Performance
  - Programmability
  - Portability
  - Robustness

- **Phase II:** Cray, IBM, Sun (July 2003 – June 2006)
  - Evaluated the entire system architecture’s impact on productivity…
    - processors, memory, network, I/O, OS, runtime, compilers, tools, …
  - …and new languages:
    - Cray: Chapel
    - IBM: X10
    - Sun: Fortress

- **Phase III:** Cray, IBM (July 2006 – 2010)
  - Implement the systems and technologies resulting from phase II
  - (Sun also continues work on Fortress, without HPCS funding)
Chapel and Productivity

Chapel’s Productivity Goals:

• vastly improve **programmability** over current languages/models
  ▪ writing parallel codes
  ▪ reading, modifying, porting, tuning, maintaining them

• support **performance** at least as good as MPI
  ▪ competitive with MPI on generic clusters
  ▪ better than MPI on more capable architectures

• improve **portability** compared to current languages/models
  ▪ as ubiquitous as MPI, but with fewer architectural assumptions
  ▪ more portable than OpenMP, UPC, CAF, …

• improve **code robustness** via improved semantics and concepts
  ▪ eliminate common error cases altogether
  ▪ better abstractions to help avoid other errors
Outline

✓ Introduction to Chapel

➢ Global-View Programming

☐ Data Parallel Examples: the Stencil Ramp

☐ Task Parallel Features & Examples

☐ Status & Summary
Parallel Programming Model Taxonomy

**programming model:** the mental model a programmer uses when coding using a language, library, or other notation

**fragmented models:** those in which the programmer writes code from the point-of-view of a single processor/thread

**global-view models:** those in which the programmer can write code that describes the computation as a whole
Global-view vs. Fragmented

- **Problem:** “Apply 3-pt stencil to vector”

\[
\begin{align*}
\text{global-view} & : ( \begin{array}{cccc}
\text{\textcolor{blue}{1}} & \text{\textcolor{blue}{2}} & \text{\textcolor{blue}{3}} & \text{\textcolor{blue}{4}} \\
\end{array} ) / 2 \\
\text{fragmented} & : \begin{array}{c}
\text{\textcolor{yellow}{1}} \\
\text{\textcolor{yellow}{2}} \\
\text{\textcolor{yellow}{3}} \\
\end{array}
\end{align*}
\]
Global-view vs. Fragmented

- **Problem:** “Apply 3-pt stencil to vector”

**Global-view**

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

**Fragmented**

\[
\begin{align*}
&= ( \quad + \quad ) / 2 \\
&= \quad + \quad \\
&= \quad + \\
&= \quad + \\
&= \quad + \\
&= \quad + \\
\end{align*}
\]
Parallel Programming Model Taxonomy

programming model: the mental model a programmer uses when coding using a language, library, or other notation

fragmented models: those in which the programmer writes code from the point-of-view of a single processor/thread

SPMD models: Single-Program, Multiple Data -- a common fragmented model in which the user writes one program & runs multiple copies of it, parameterized by a unique ID

global-view models: those in which the programmer can write code that describes the computation as a whole
Global-view vs. SPMD Code

- **Problem:** “Apply 3-pt stencil to vector”

  **global-view**

  ```chapel
  var n: int = 1000;
  var a, b: [1..n] real;

  forall i in 2..n-1 {
    b(i) = (a(i-1) + a(i+1))/2;
  }
  ```

  **SPMD**

  ```chapel
  var n: int = 1000;
  var locN: int = n/numProcs;
  var a, b: [0..locN+1] real;

  if (iHaveRightNeighbor) {
    send(right, a(locN));
    recv(right, a(locN+1));
  }

  if (iHaveLeftNeighbor) {
    send(left, a(1));
    recv(left, a(0));
  }

  forall i in 1..locN {
    b(i) = (a(i-1) + a(i+1))/2;
  }
  ```
Global-view vs. SPMD Code

**Problem:** “Apply 3-pt stencil to vector”

**Global-view**

```chapel
var n: int = 1000;
var a, b: [1..n] real;

forall i in 2..n-1 {
    b(i) = (a(i-1) + a(i+1))/2;
}
```

**SPMD**

```chapel
var n: int = 1000;
var locN: int = n/numProcs;
var a, b: [0..locN+1] real;
var innerLo: int = 1;
var innerHi: int = locN;

if (iHaveRightNeighbor) {
    send(right, a(locN));
    recv(right, a(locN+1));
} else {
    innerHi = locN-1;
}

if (iHaveLeftNeighbor) {
    send(left, a(1));
    recv(left, a(0));
} else {
    innerLo = 2;
}

forall i in innerLo..innerHi {
    b(i) = (a(i-1) + a(i+1))/2;
}
```

Assumes `numProcs` divides `n`; a more general version would require additional effort.
Current HPC Programming Notations

- **communication libraries:**
  - MPI, MPI-2
  - SHMEM, ARMCI, GASNet
    (fragmented, typically SPMD)

- **shared memory models:**
  - OpenMP
    (global-view, trivially)

- **PGAS languages:**
  - Co-Array Fortran
    (SPMD)
  - UPC
    (SPMD)
  - Titanium
    (SPMD)
MPI SPMD pseudo-code

Problem: “Apply 3-pt stencil to vector”

SPMD (pseudocode + MPI)

```chapel
var n: int = 1000, locN: int = n/numProcs;
var a, b: [0..locN+1] real;
var innerLo: int = 1, innerHi: int = locN;
var numProcs, myPE: int;
var retval: int;
var status: MPI_Status;

MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
MPI_Comm_rank(MPI_COMM_WORLD, &myPE);
if (myPE < numProcs-1) {
    retval = MPI_Send(&a(locN), 1, MPI_FLOAT, myPE+1, 0, MPI_COMM_WORLD);
    if (retval != MPI_SUCCESS) { handleError(retval); }
    retval = MPI_Recv(&a(locN+1), 1, MPI_FLOAT, myPE+1, 1, MPI_COMM_WORLD, &status);
    if (retval != MPI_SUCCESS) { handleErrorWithStatus(retval, status); }
} else
    innerHi = locN-1;
if (myPE > 0) {
    retval = MPI_Send(&a(1), 1, MPI_FLOAT, myPE-1, 1, MPI_COMM_WORLD);
    if (retval != MPI_SUCCESS) { handleError(retval); }
    retval = MPI_Recv(&a(0), 1, MPI_FLOAT, myPE-1, 0, MPI_COMM_WORLD, &status);
    if (retval != MPI_SUCCESS) { handleErrorWithStatus(retval, status); }
} else
    innerLo = 2;
forall i in (innerLo..innerHi) {
    b(i) = (a(i-1) + a(i+1))/2;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays
subroutine comm3(u,n1,n2,n3,kk)
    integer axis, i1, i2, i3, buff_len, indx, buff_id, d1, d2, d3
    double precision u(n1,n2,n3)
    include 'cafnpb.h'
    include 'globals.h'
    implicit none
    buff_len = 0
    buff_id = 2 + dir
    if( axis .eq.  1 )then
        do i=1,nm2
            u(1,i2,i3) = buff(i,3)
            indx = indx + 1
            buff_len = buff_len + 1
        enddo
        if( dir .eq. -1 )then
            do i=1,nm2
                u(1,i2,i3) = buff(i,1)
                indx = indx + 1
                buff_len = buff_len + 1
            enddo
        endif
        buff(1:buff_len,buff_id) = buff(len, buff_id+1)
    endif
    else if( axis .eq.  2 )then
        do i1=1,n1
            do i3=2,n3-1
                buff(i1,4) = buff(i1,3)
                indx = indx + 1
                buff_len = buff_len + 1
            enddo
            if( dir .eq. -1 )then
                do i1=1,n1
                    buff(i1,2) = buff(i1,1)
                    indx = indx + 1
                    buff_len = buff_len + 1
                enddo
                buff(i1:buff_len,buff_id) = buff(len, buff_id+1)
            endif
            u(i1,i2,1) = buff(indx, buff_id)
            indx = indx + 1
            buff_len = buff_len + 1
        enddo
        if( dir .eq. -1 )then
            do i1=1,n1
                buff(i1,2) = buff(i1,1)
                indx = indx + 1
                buff_len = buff_len + 1
            enddo
            buff(i1:buff_len,buff_id) = buff(len, buff_id+1)
        endif
    else if( axis .eq.  3 )then
        do i=1,nm2
            u(n1,1,i3) = buff(i,4)
            indx = indx + 1
            buff_len = buff_len + 1
        enddo
        do i=1,nm2
            u(n1,i2,1) = buff(i,2)
            indx = indx + 1
            buff_len = buff_len + 1
        enddo
        if( dir .eq. -1 )then
            do i=1,nm2
                u(n1,i2,i3) = buff(i,3)
                indx = indx + 1
                buff_len = buff_len + 1
            enddo
            do i=1,nm2
                u(n1,1,i3) = buff(i,1)
                indx = indx + 1
                buff_len = buff_len + 1
            enddo
            buff(i:buff_len,buff_id) = buff(len, buff_id+1)
        endif
    endif
end subroutine comm3
Summarizing Fragmented/SPMD Models

- **Advantages:**
  - fairly straightforward model of execution
  - relatively easy to comprehend, learn, reason about
  - relatively easy to implement
  - reasonable performance on commodity architectures
  - portable/ubiquitous
  - lots of important scientific work has been accomplished using them

- **Disadvantages:**
  - blunt means of expressing parallelism: cooperating executables
  - fails to abstract away architecture / implementing mechanisms
  - obfuscates algorithms with many low-level details
    - error-prone
    - brittle code: difficult to read, maintain, modify, *experiment*
    - “MPI: the assembly language of parallel computing”
Outline

✓ Introduction to Chapel
✓ Global-View Programming
➢ Data Parallel Examples: the Stencil Ramp
☐ Task Parallel Features & Examples
☐ Status & Summary
Example 1: Jacobi Iteration

\[ \sum \left( \begin{array}{c} 
\text{+} \\
\text{+} \\
\text{+} \\
\end{array} \right) \div 4 \]

repeat until max change < \( \varepsilon \)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
        + A(i,j-1) + A(i,j+1)) / 4.0;

    var delta = max reduce abs(A(D) - Temp(D));
    A(D) = Temp(D);
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
do {
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4.0;
    var delta = max reduce abs(A(D) - Temp(D));
    A(D) = Temp(D);
} while (delta > epsilon);
writeln(A);
```

Declare program parameters

- `config` ⇒ can’t change values after initialization
- `const` ⇒ can be set on executable command-line

Prompt:
```
jacobi --n=10000 --epsilon=0.0001
```

Note that no types are given; inferred from initializer

- `n` ⇒ `integer` (current default, 32 bits)
- `epsilon` ⇒ `floating-point` (current default, 64 bits)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
D: subdomain(BigD) = [1..n, 1..n],
LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
do {
[(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4.0;
var delta = max reduce abs(A(D) - Temp(D));
A(D) = Temp(D);
} while (delta > epsilon);
writeln(A);
```

Declare domains (first class index sets)

- `domain(2)` ⇒ 2D arithmetic domain, indices are integer 2-tuples
- `subdomain(P)` ⇒ a domain of the same type as `P` whose indices are guaranteed to be a subset of `P`'s
- `exterior` ⇒ one of several built-in domain generators
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1, 0);

var A, Temp : [BigD] real;
```

Declare arrays

- **var** ⇒ can be modified throughout its lifetime
- : **T** ⇒ declares variable to be of type **T**
- : **[D] T** ⇒ array of size **D** with elements of type **T**
  - *(no initializer)* ⇒ values initialized to default value (0.0 for reals)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
config const epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
```

Set Explicit Boundary Condition

indexing by domain ⇒ slicing mechanism
array expressions ⇒ parallel evaluation
Jacobi Iteration in Chapel

Compute 5-point stencil

\[(i,j) \text{ in } D \] \Rightarrow parallel forall expression over \(D\)'s indices, binding them to new variables \(i\) and \(j\)

\textbf{Note:} since \((i,j) \in D\) and \(D \subseteq BigD\) and \(Temp: [BigD]
\Rightarrow no bounds check required for \(Temp(i,j)\)
with compiler analysis, same can be proven for \(A\)'s accesses

\[\sum \left( \begin{array}{c}
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\text{\textbullet} \\
\end{array} \right) \div 4 \]

\[ \forall (i, j) \text{ in } D \] \(Temp(i, j) = (A(i-1, j) + A(i+1, j) + A(i, j-1) + A(i, j+1)) / 4.0; \)

\textbf{var} delta = max \textbf{reduce} abs(A(D) - Temp(D));
A(D) = Temp(D);
\} \textbf{while} (delta > epsilon);

\textbf{writeln}(A);
Jacobi Iteration in Chapel

```chapel
config const n = 6,
   epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],

Compute maximum change

**op reduce** ⇒ collapse aggregate expression to scalar using **op**

**Promotion:** `abs()` and `–` are scalar operators, automatically promoted to work with array operands

do {
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 1.0;

    var delta = max reduce abs(A(D) - Temp(D));
    A(D) = Temp(D);
} while (delta > epsilon);

writeln(A);
```

Compute maximum change

*op reduce* ⇒ collapse aggregate expression to scalar using *op*

**Promotion:** `abs()` and – are scalar operators, automatically promoted to work with array operands
Jacobi Iteration in Chapel

```chapel
config const n = 6,
     epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
do
{
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4.0;

    var delta = max reduce abs(A(D) - Temp(D));
    A(D) = Temp(D);
} while (delta > epsilon);

writeln(A);
```

- **Copy data back & Repeat until done**
- uses slicing and whole array assignment
- standard `do...while` loop construct
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [(i,j) in D]
        Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4.0;

    var delta = max reduce abs(A(D) - Temp(D));
    A(D) = Temp(D);
}
while (delta > epsilon);

writeln(A);
```

Write array to console

If written to a file, parallel I/O would be used
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1] distributed (Block),
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
```

With this change, same code runs in a distributed manner
Domain distribution maps indices to locales
⇒ decomposition of arrays & default location of iterations over locales
Subdomains inherit parent domain’s distribution
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1] distributed (Block),
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
        + A(i,j-1) + A(i,j+1)) / 4.0;

    var delta = max reduce abs(A(D) - Temp(D));
    [ij in D] A(ij) = Temp(ij);
} while (delta > epsilon);

writeln(A);
```
Example 2: Multigrid

- **V**: input array
- **U**: hierarchical work arrays
- **R**: 

\[ n \]

\[ \text{numLevels} \]
Overview of NAS MG

initialize $V$

run timed portion

output norms & timings
MG’s projection/interpolation cycle

rprj3 → resid → interp → psinv

rprj3 → resid → interp → psinv

rprj3 → resid → interp → psinv

Google 2007: Chapel (31)
Multigrid: 27-Point Stencils

\[ w_0 + w_1 + w_2 + w_3 \]
Multigrid: Stencils in Chapel

- Can write them out explicitly, as in Jacobi...

```python
def rprj3(S, R) {
    param w: [0..3] real = (0.5, 0.25, 0.125, 0.0625);
    const Rstr = R.stride;

    forall ijk in S.domain do
        S(ijk) = w(0) * R(ijk)
            + w(1) * (R(ijk+Rstr*(1,0,0)) + R(ijk+Rstr*(-1,0,0))
                + R(ijk+Rstr*(0,1,0)) + R(ijk+Rstr*(0,-1,0))
                + R(ijk+Rstr*(0,0,1)) + R(ijk+Rstr*(0,0,-1)))
            + w(2) * (R(ijk+Rstr*(1,1,0)) + R(ijk+Rstr*(1,-1,0))
                + R(ijk+Rstr*(-1,1,0)) + R(ijk+Rstr*(-1,-1,0))
                + R(ijk+Rstr*(1,0,1)) + R(ijk+Rstr*(1,0,-1))
                + R(ijk+Rstr*(-1,0,1)) + R(ijk+Rstr*(-1,0,-1)))
            + w(3) * (R(ijk+Rstr*(1,1,1) + R(ijk+Rstr*(1,1,-1))
                + R(ijk+Rstr*(1,-1,1) + R(ijk+Rstr*(1,-1,-1))
                + R(ijk+Rstr*(-1,1,1) + R(ijk+Rstr*(-1,1,-1))
                + R(ijk+Rstr*(-1,-1,1) + R(ijk+Rstr*(-1,-1,-1)))
```

Google 2007: Chapel (33)
Multigrid: Stencils in Chapel

- ...or, note that a stencil is simply a reduction over a small subarray expression
- Thus, stencils can be written in a “syntactically scalable” way using reductions:

```chapel
def rprj3(S, R) {
    const Stencil: domain(3) = [-1..1, -1..1, -1..1], // 27-points
    w: [0..3] real = (0.5, 0.25, 0.125, 0.0625), // 4 wgts
    w3d = [(i,j,k) in Stencil] w((i!=0) + (j!=0) + (k!=0));

    forall ijk in S.domain do
        S(ijk) = + reduce [off in Stencil]
            (w3d(off) * R(ijk + R.stride*off));
}
```
subroutine comm3(u,n1,n2,n3,kk)
    implicit none
    integer n1, n2, n3, kk
    double precision u(n1,n2,n3)
    integer axis
    if( .not. dead(kk) )then
        if( axis .eq.  2 )then
            do  i1=1,n1
                do  i2=1,n2
                    do  i3=2,n3-1
                        buff_len = buff_len + 1
                        indx = indx + 1
                        u(i1,i2,i3) = buff(indx, buff_id )
                        indx = indx + 1
                    enddo
                enddo
            enddo
            else if( axis .eq.  3 )then
                do  i2=1,n2
                    do  i3=2,n3-1
                        buff_len = buff_len + 1
                        indx = indx + 1
                        u(1,i2,i3) = buff(indx, buff_id )
                        indx = indx + 1
                    enddo
                enddo
            endif
        endif
    endif
end subroutine comm3
Example 3: Fast Multipole Method (FMM)

```chapel
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

1D array over levels of the hierarchy
Example 3: Fast Multipole Method (FMM)

```chapel
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

- 1D array over levels of the hierarchy
- ...of 3D sparse arrays of cubes (per level)
- ...of 1D vectors
- ...of 2D discretizations of spherical functions, (sized by level)
- ...of complex values

Example:

- `x + y \cdot i`
FMM: Supporting Declarations

\[
\text{var } \text{OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;}
\]

\textit{previous definitions:}

\[
\text{var } \text{n: int } = \ldots;
\]
\[
\text{var } \text{numLevels: int } = \ldots;
\]
\[
\text{var } \text{Levels: domain}(1) = [1..\text{numLevels}];
\]
\[
\text{var } \text{scale: [lvl in Levels] int } = 2^{*(\text{lvl}-1)};
\]
\[
\text{var } \text{SgFnSize: [lvl in Levels] int } = \text{computeSgFnSize}(\text{lvl});
\]
\[
\text{var } \text{LevelBox: [lvl in Levels] domain}(3) = [(1,1,1)..(n,n,n)] \text{ by } \text{scale}(\text{lvl});
\]
\[
\text{var } \text{SpsCubes: [lvl in Levels] sparse subdomain}(\text{LevelBox}) = \ldots;
\]
\[
\text{var } \text{Sgfns: [lvl in Levels] domain}(2) = [1..\text{SgFnSize}(\text{lvl}), 1..2*\text{SgFnSize}(\text{lvl})];
\]
FMM: Computation

\[
\text{var } \text{OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;} \\
\]

\textit{outer-to-inner translation:}

\[
\text{for lvl in [1..numLevels) by -1 { } }
\]
\[
\ldots
\]
\[
\text{for all cube in SpsCubes(lvl) { } }
\]
\[
\text{for all sib in out2inSiblings(lvl, cube) { } }
\]
\[
\text{const Trans = lookupXlateTab(cube, sib); } \\
\text{atomic ISgfn(lvl)(cube) += OSgfn(lvl)(sib) * Trans; } \\
\]
\[
\ldots
\]
Fast Multipole Method: Summary

- Chapel code captures structure of data and computation far better than sequential Fortran/C versions (let alone MPI versions of them)
  - cleaner, more succinct, more informative
  - rich domain/array support plays a big role in this

- Parallelism shifts at different levels of hierarchy
  - Global view and syntactic separation of concerns helps here
  - Imagine writing in a fragmented language

- Code very clear to Boeing engineer familiar with FMM

- Yet, I’ve elided some non-trivial code (data distribution)
Example 4: Stencils on Unstructured Grids
Key Data Parallel Challenge: Distributions

A domain’s indices may be distributed…
…implying a distribution for its arrays
…and a default work assignment for iteration & slicing using the domain
Key Data Parallel Challenge: Distributions

A domain’s indices may be distributed…

…implying a distribution for its arrays

…and a default work assignment for iteration & slicing using the domain
Distributions Overview

**Distributions**: “recipes for distributed arrays”

- Intuitively, distributions implement the lowering…
  - **from**: the user’s global view of distributed data aggregates
  - **to**: the fragmented implementation for distributed memory machines

- Define two primary things:
  - mapping of indices to locales
  - implementation of domain indices/array elements within a locale

- Author must implement an interface which supports:
  - allocation/reallocation of indices and elements
  - mapping functions (e.g., index-to-locale, index-to-value)
  - iterators: parallel/serial; global/local
  - communication idioms

- Chapel provides a standard library of distributions…
  ...written using the same mechanism as user-defined distributions
  ...tuned for different platforms to maximize performance
Outline

✓ Introduction to Chapel
✓ Global-View Programming
✓ Data Parallel Examples: the Stencil Ramp
➢ Task Parallel Features & Examples
☐ Status & Summary
Task Parallelism: Task Creation

- **begin**: creates a task for future evaluation

  ```
  begin DoThisTask();
  WhileContinuing();
  TheOriginalThread();
  ```

- **co-begin**: supports a structured list of sibling tasks:

  ```
  cobegin { 
  computePivot(lo, hi, data);
  cobegin {
  Quicksort(lo, pivot, data);
  Quicksort(pivot, hi, data);
  } // implicit synchronization here
  cobegin {
  ComputeTaskA(...);
  ComputeTaskB(...);
  ComputeTaskC(...);
  } // implicit synch
  ```

- **co-forall**: loop-style construct for generating sibling tasks

  ```
  coforall e in Edges {
  exploreEdge(e);
  } // implicit synchronization here
  ```
Task Parallelism: Task Coordination

- **sync variables**: store full/empty state along with value
  
  ```chapel
  var result: sync real; // result is initially empty
  cobegin {
      ... = result; // block until full, leave empty
      result = ...; // block until empty, leave full
  }
  result.readFF(); // read when full, leave full;
  // other variations also supported
  ```

- **single-assignment variables**: writable once only
  
  ```chapel
  var result: single real = begin f(); // result initially empty
  ...
  // do some other things
  total += result; // block until result has been filled
  ```

- **atomic sections**: support transactions against memory
  
  ```chapel
  atomic {
      newnode.next = insertpt;
      newnode.prev = insertpt.prev;
      insertpt.prev.next = newnode;
      insertpt.prev.prev = newnode;
  }
  ```
Task Parallelism: Task Placement

- **on clauses**: indicate where tasks should execute
  
  • in a data-driven manner...
    
    ```chapel
    computePivot(lo, hi, data);
    cobegin {
      on A(lo) do Quicksort(lo, pivot, data);
      on A(pivot) do Quicksort(pivot, hi, data);
    }
    ```
  
  • ...or by naming machine resources explicitly
    
    ```chapel
    // Chapel provides: const Locale: [0..numLocales-1] locale;
    on Locale(0) begin gatherResults();
    coforall loc in 1..numLocales-1 {
      on Locale(loc) do compute();
    }
    ```
Example 1: UTS

**UTS:** Unbalanced Tree Search Benchmark
- performs exhaustive search on a variety of random, unbalanced trees
- requires dynamic load balancing to achieve good speedup
- employs cryptographically strong tree generation using SHA1
- joint effort between Ohio State, U. of North Carolina, U. of Maryland
Consider a balanced Tree ... 

Process each sub-tree in parallel
What if the tree is unbalanced?

This strategy results in severe load imbalance!

UTS is designed to require dynamic balancing of the workload

contents adapted from James Dinan, the Ohio State University
exploreTree(rand(SEED));

def exploreTree(id) {
    const numChildren = computeNumChildren(rand(id));
    coforall c in 1..numChildren do
        exploreTree(rand(id,c));
}

**The Big Question**

**Q:** How will this *coforall* be implemented?
- spawn a thread per task?
- use a fixed number of threads and a task pool?
- use some sort of peer-based work-sharing or stealing algorithm a la Cilk or the Cray MTA?

**A:** Could be any of these; semantics are separate from implementing mechanisms
exploreTree(rand(SEED));

```python
def exploreTree(id) {
    const numChildren = computeNumChildren(rand(id));
    coforall c in 1..numChildren do
        exploreTree(rand(id,c));
}
```

**The Big Question**

**Q:** How will this `coforall` be implemented?

- spawn a thread per task?
- use a fixed number of threads and a task pool?
- use some sort of peer-based work sharing or stealing algorithm a la Cilk/Cray MTA?

**A:** Could be any of these; semantics are separated from implementing mechanisms.

**An Open Question**

**Q:** For the user who wants more control over their program’s implementation, what is the equivalent of a distribution for task-parallel codes? (i.e., How should the recipe for throttling parallelism be expressed and specified?)
Chapel Load Balancing Code

```chapel
var numTasks: sync int = 0;
var terminated: single bool;

def balance_load(inout q: DeQueue(TreeNode)) {
    if (q.size > 2*chunkSize && numTasks.readXX() < MAX_TASKS) {
        // Split chunkSize nodes into a new queue
        var work = q.split(chunkSize);
    }
}

def create_tree(inout q: DeQueue(TreeNode)) {
    ...
    // Update task counts; detect termination
    var numTasks_l = numTasks;
    numTasks_l -= 1;
    if numTasks_l == 0 then
        terminated = true;
    numTasks = numTasks_l;
}
```

Declare synchronized variables to throttle number of live tasks

Check if we have surplus work and whether the runtime is running low on available tasks

Spawn off surplus work and rely on the runtime to balance the load.

When a task completes it updates the shared counter and checks to see if it is the last one.

When a task completes it updates the shared counter and checks to see if it is the last one.
UTS MPI Load Balancing Code

Google 2007: Chapel (55)

contents adapted from James Dinan, the Ohio State University
Example 2: MADNESS

**MADNESS:**
- Multiresolution ADaptive NumErical Scientific Simulation
- a framework for scientific simulation in many dimensions using adaptive multiresolution methods in multiwavelet bases

**People:**
- Gregory Beylkin (University of Colorado), George Fann (Oak Ridge National Laboratory), Zhenting Gan (CCSG), Robert Harrison (CCSG), Martin Mohlenkamp (Ohio University), Fernando Perez (University of Colorado), P. Sadayappan (The Ohio State University), Takeshi Yanai (CCSG)
What does Madness do?

- Think of Madness as a math library
- Numerical representations for analytic functions
  - Stored in the scaling function (Gauss Legendre Polynomial) and Multiwavelet bases
  - Operations on functions become fast with guaranteed precision
  - Differential and Integral operators become $O(n)$ in numerical representation

- Applications that can benefit from Madness include:
  - Density Functional Theory (DFT) (Quantum chemistry domain)
    - Explore electronic structure of many-body systems
  - Fluid dynamics
  - Climate modeling
  - Etc …
Numerical Representation for Functions

- Analytic function is *projected* into the numerical representation
- Approximate the function using basis functions
  - Similar to Fourier, but basis functions have compact support
  - Approximation is over a closed interval of interest
- Recursively subdivide the analytic function spatially to achieve desired accuracy
- Avoid extra computation in uninteresting areas
- Store the result in a *Function Tree*
  - 1d: Binary Tree
  - 2d: Quad Tree
  - 3d: Oct Tree
The 1d Function Tree of a Gaussian

contents adapted from James Dinan, the Ohio State University
Function Evaluation in the Numerical Representation

\[ k = 8 \]
Core Algorithm: Differentiation

- **Perform:** \( df = f \cdot \text{diff()} \)
- Walk down the tree and everywhere that we have coefficients, perform differentiation
- Performing differentiation involves getting our left and right neighbors and applying the derivative operator
Differentiation: I have neighbors

contents adapted from James Dinan, the Ohio State University
Differentiation: I’m too fine

contents adapted from James Dinan, the Ohio State University
Differentiation: I’m too coarse

Google 2007: Chapel (64)

contents adapted from James Dinan, the Ohio State University
def diff (n = 0, l = 0, result) {
    if !s.has_coeffs(n, l) {
        // Run down tree until we hit scaling function coefficients
        diff(n+1, 2*l, result);
        diff(n+1, 2*l+1, result);
    } else {

        var sm = get_coeffs(n, l-1);
        var sp = get_coeffs(n, l+1);
        var s0 = s[n, l];

        // We have s0, check if we found sm and sp at this level
        if !isNone(sm) && !isNone(sp) {
            var r = rp*sm + r0*s0 + rm*sp;
            result.s[n, l] = r * 2.0**n;
        } else {
            recur_down(n, l);
            diff(n+1, 2*l, result);
            diff(n+1, 2*l+1, result);
        }
    }
}
def diff (n = 0, l = 0, result) {
    if !s.has_coeffs(n, l) {
        // Run down tree until we hit scaling function coefficients
        cobegin {
            diff(n+1, 2*l  , result);
            diff(n+1, 2*l+1, result);
        }
    } else {
        cobegin {
            var sm = get_coeffs(n, l-1);
            var sp = get_coeffs(n, l+1);
            var s0 = s[n, l];
        }
        // We have s0, check if we found sm and sp at this level
        if !isNone(sm) && !isNone(sp) {
            var r = rp*sm + r0*s0 + rm*sp;
            result.s[n, l] = r * 2.0**n;
        } else {
            recur_down(n, l);
            cobegin {
                diff(n+1, 2*l  , result);
                diff(n+1, 2*l+1, result);
            }
        }
    }
}
Outline

- Introduction to Chapel
- Global-View Programming
- Data Parallel Examples: the Stencil Ramp
- Task Parallel Features & Examples

- Status & Summary
Chapel Work

- Chapel Team’s Focus:
  - specify Chapel syntax and semantics
  - implement prototype Chapel compiler
  - code studies of benchmarks, applications, and libraries in Chapel
  - community outreach to inform and learn from users, colleagues
  - support users evaluating our preliminary releases
  - refine language based on these activities
Chapel Status

- Draft language specification publicly available
- Portable prototype compiler development progressing
  - Current State:
    - most base language features implemented
    - task parallelism supported within one locale
    - data parallel features implemented, but single-threaded
    - performance competitive with hand-coded C for key 1D kernels
  - Current work/Next steps:
    - support for multi-locale task parallelism
    - parallel implementation of data parallel features
    - continue improving performance for broader set of idioms
- June 2007 release made to ~12 early users at ~8 sites
  - to provide early evaluation and feedback...
    ...on language
    ...on implementation – usefulness and portability
  - initial response positive, (cautiously) optimistic
Academic Collaborations

- **Vikram Adve & Robert Bocchino (UIUC):** software transactional memory for distributed memory computation

- **Franz Franchetti (CMU):** SPIRAL back-end targeting Chapel to leverage its portability

- **Dan Grossman, Larry Snyder (UW):** co-sponsoring a seminar studying and evaluating Chapel, Fall `07
Chapel Contributors

- **Current:**
  - Brad Chamberlain
  - Steven Deitz
  - Samuel Figueroa
  - Mary Beth Hribar
  - David Iten

- **Alumni:**
  - David Callahan
  - Hans Zima (CalTech/JPL)
  - John Plevyak
  - Wayne Wong
  - Shannon Hoffswell
  - Roxana Diaconescu (CalTech)
  - Mark James (JPL)
  - Mackale Joyner (2005 intern, Rice University)
  - Robert Bocchino (2006 intern, UIUC)
  - James Dinan (2007 intern, OSU)
Summary

- Chapel’s goal: “solve the parallel programming problem”
  - general parallel programming
    - data-, task-, nested parallelism
    - general architectures
    - interoperability with other parallel models
  - able to achieve high performance
    - user can tune data and computation for locality
    - user-defined distributions for data aggregates
    - explicit control for task parallelism
    - ability to work at multiple levels of detail within one program

- We have our work cut out for us… …but what enjoyable work it is!
For More Information…

http://chapel.cs.washington.edu

chapel_info@cray.com
bradc@cray.com