Chapel

Cascade High Productivity Language - chapel-lang.org

Brad Chamberlain (Employee at HPE, Seattle, was Cray) started working on ZPL at UW

Z level Programming Language

University of Washington, Larry Snyder
http://cs.washington.edu/research/zpl

Poker

- GUI IDE for parallel programs
- CTA machine ... Chip
- Phase ...
  - connection network (x)
  - process code (y)
  - export/import interphase
- Switching between phases (z)
- very simple scripting language

Program at the Z level ...

- Results are another language: ZPL
- Doesn’t have a GUI
- Doesn’t specify computing elements
- Very high level ...
- parallel array processing language
- syntax similar to Pascal
- a research language, no wide adoption

ZPL Region

first class concept

- region R = [1 .. n, 1 .. n];
- InR = [2 .. n-1, 2 .. n-1];
- TopRow = [1 .. n, 1];
- LeftC = [1, 1 .. n];

Direction -- used in regions

- direction west = [-1, 0];
- east = [1, 0]; NE = [1, 1]; SW = [-1, -1];

- region modifiers
- in: region left = west in R; // left column
- of: region lleft = west of InR;
- at: region lInR = InR at west;
- by: direction step = [1, 2];
  - region strip = R by step;

Parallel Arrays

Region of data

- var a, b, c : [R] double;
  - d : [InR] double;

Assignment & ops

- [R] a := b;
- [InR] a := b;
- [R] a := b + c;
- [InR] d := a + b;
- var sum : double;
  - [R] sum := +<< a; // reduction
- var f, g : [1 .. n] integer;
- [1 .. n] f := +|| g; // prefix sum
- reduction/prefix ops: +<< *<< min<< max<< |<< (or) &<< (and) bor<< band<< userfunc<< (assoc & comm)

Program examples

hello.z

- basic outline

jacobi.z

- config variables
  - -p, -help, -svarname=value
- boundary conditions
- Communication: @ operator
  - @^ operator -- wrap
- ResetTimer(), CheckTimer()
- write() on a region

jacobi2.z

- write() to a file

Zpl has a lot more ... but old ... gives a good intro to why Chapel

Chapel -- Cascade High Productivity Language

- Done by Cray (Now HP Enterprise) in Seattle, now at version 2.4.0
- Brad Chamberlain -- Ph.D. at UW on ZPL
- DARPA-funded HPCS (High Productivity Computing systems)
- old paper: on Linux lab machines as ~phil/public/csci515/final-chamberlain.pdf
- Web chapel-lang.org
  - open source -- get your own copy at the web site
  - video intro by Brad

Basic ideas of Chapel/ZPL .... global view language

- MPI -- fragmented view
  - SPMD
  - Data spread across processors
  - communication is explicit
- Global view
  - single program
  - Data in single program
  - Compiler figures out communication

Fragmented -- 3 point stencil (MPI style)

```
var n: int = 1000;
var locN: int = n/numTasks;
var A, B: [0..locN+1] float;
var myItLo: int = 1;
var myItHi: int = locN;
if (iHaveLeftNeighbor) then
   send(left, (A[1])
else
   myItLo = 2;
if (iHaveRightNeighbor) {
   send(right, A[locN]);
   recv(right,A[locN+1]);
} else
  myItHi = locN-1;
if (iHaveLeftNeighbor) then
  recv(left,A[0]);
for i := myItLo to myItHi do
  B[i] = (A[i-1] + A[i+1]) / 2
```

Global view 3 point stencil (Chapel)

```
var n: int = 1000;
var A, B: [1..n] float;
forall i in 2..n-1 do
  B[i] = (A[i-1] + A[i+1]) / 2;
```

Points of interest:

- forall i in ...
- No worries about numTasks
- No sequential dependencies
- Allows compiler and runtime system to partition A and B
- Allows for algorithm/implementation separation
- Does not show underlying system, shared vs. non-shared
- Portability -- recompile and reuse

Many globlal view languages appear to be "array processing languages"

Chapel’s goals

- Be a global view language
- Have constructs that allow for large amounts of parallelism
- Support both shared-memory and distributed-memory models
- Support both CPU and GPU computation
- Support for "data parallelism" and "task parallelism"
  - Beware ... these can conflict!
- Block-imperative programming style
- Allow object orient programming -- in parallel!
- Be a "useful" language, not a research language.
- Be efficient in implementation
- Run on a wide variety of current parallel machines
  - small n SMP machines, small clusters
  - All of the top 500 machines!
  - GPU work without having to use CUDA

Simple programs

- /home/phil/chapel-2.4.0
- examples/hello.chpl
- examples/hello2-module.chpl
  - /home/phil/setchpl-gasnet.bash
    - chpl file.chpl
  - /home/phil/setchpl-seq.bash

Block structured language similar to Ada, Pascal, Modula, C ...
Basic types and variables

var <name> [: <definition>] [= <initializer>] ;

Examples:

- examples/primers/variables.chpl

Locales: a unit of a parallel architecture, e.g. cluster node.

- predefined variables:
  - const numLocales: int = ...;
  - const Locales: [1 .. numLocales] Locale = ...;
- examples/hello6-taskpar-dist.chpl
  - here.id entry
- examples/hello5-taskpar.chpl
  - here.numCores;
  - Notice: -nl 2 does nothing here!

Other Elements

Basic types

- bool, int, uint, real, imag, complex, string
- enum id { enum-constand-list }; (C style with name=value)

Structured types

- record types, union types, class types
- tuple (lightweight record, all fields of same type)
- domains (sets of indices ... e.g. region)
- arrays, index types ... more later.

Flow Control

- if/then/else
- while, do while
- for, forall, coforall
  - Be careful! coforall can cause slowdowns!
- select X { when ... do ... when ... do ... }
- type select
- label, break, continue
- use statement (for records)
- begin (start a task, don’t wait)
- cobegin (parallel statements)

Ranges

first class concept: ranges (examples/primer/ranges.chpl)

- const r = 1..10;
- const low = 5, high = 15;
- const r2 = low .. high;
- const r3 = low .. <high; // 5 ... 14
- Empty: const e1 = 1 .. 0;
- const countDown = 1..10 by -1; // 10, 9, 8, ..., 1
  - runtime does not like arrays with these ranges
- Basis for iteration

```
var sum = 0;
for i in 1 .. 10 do  sum += i;
```

- Array declarations

```
const r = 1 .. 10;
var A: [r] int;
var B: [r,r] float;
var C: [-5 .. 5] complex;
```

- Unbounded!

```
const ur = 1.. ;
```

- ranges of enumerated:

```
enum dir {north, east, south, west};
const ne = dir.north .. dir.east;  // enum name required
```

- more: #, align, +, -, ==, slicing, variables, parameters ... (see program)

Domains (examples/primer/domains.chpl)

Domains ... needed for arrays

- Index set ... specified by ranges.

  var RD: domain(3) = {1..10, 1..10, 1..10}
- :proc:~ChapelDomain.expand -- expands or contracts domains, equally on both ends

  Var RDexp = RD.expand((1,2,-2)); // {0..11, -1..12, 3..8}
- :proc:~ChapelDomain.exterior -- new domain, 0, same, non 0 - outside.

  Var RDext1 = RD.exterior((1,4,-4)); // {11..11, 11..14, -3..0}

  Var RDext2 = RD.exterion((0,4,0)); // {1..10, 11..14, 1..10}
- :proc:~ChapelDomain.interior -- new domain, + from bottom, - from top

  Var RDint = RD.interior((2,-5,1)); // {9..10, 1..5, 10..10}
- :proc:~ChapelDomain.translate -- moves indecies up or down

  Var RDtrans = RD.translate((0,4,-4)); // {1..10, 5..14, -3..6}

  Var RDtrans0 = RD.translage(-1); // {0..9, 0..9, 0..9}
- Sparse domains are definded, but not well supported at the time
- can also define as consts

Tuples (examples/primer/tuples.chpl)

- grouping of several values together in a list

```
var myTuple = (1, "Two");
myTuple(2) is "two"  (assert(myTuple1(1) == "two"))
```

- Sometimes feels like a struct/record but is used in different ways

```
var tupleDef1: (int, string);
var tupleDef2: 3*int;
tupleDef2 += 1; // can do math on tuples with single math type
```

- Various manipulations on tuples

```
var (myInt, myString) = myTuple;
for t in myTuple { writeln(t); }
proc magnitude( (x,y,z):3*real ) {
 return sqrt(x*x + y*y + z*z);
}
var threeReals = (1.0, 2.0, 2.0);
var m = magnitude(threeReals);
writeln(m);
```

- One element tuple ?

```
var one = (1,);
writeln(one);
```

Arrays

Simple array work: (examples/primer/arrays.chpl)

- var A: [1..n] real; // no "array" key word
- A[1] = 1.1;
- A(2) = 2.2; // parens also work
- A[2..4] = 3.3; // slicing a sub array
- writeln(A); // writes entire array
- writeln(A[2..4]); // or A(2..4)
- var B: [1..n, 1..n] real; // multi-dimensional
- forall (i,j) in {1..n, 1..n} do
- B(i,j) = i + j/10.0;
- forall b in B do b += 1; // Do something to every element
- forall (i,j) in B.domain do B(i,j) -= 1; // another way

```
printArr(B);  // exact code
def printArr(X: [?D] real) {
  writeln("within printArr, D is: ", D, "\n");
  forall (i,j) in D do
    X(i,j) = -X(i,j);
  writeln("after negating X within printArr, X is:\n", X, "\n");
}
```

- var ProbSace: domain(2) = {1..n, 1..n};
- var C, D, E: [ProbSpace] bool;
- for (i,j) in ProbSpace do
- C(i,j) = (i+j) %3 == 0;
- for ij in ProbSpace do // ij is a tuple!
- D(ij) = ij(1) == ij(2);
- E = C; // array assignment
- same as:
- forall (e,c) in (E,C) do e = c;
- E = 1.0; // promotion of scalar to array
- D[2..n-1, 2..n-1] = C[1..n-2, 3..n]; // sub array assignment
- D[2.., ..] = C[..n-1, ..]; // less specification!
- A = B[n/2, ..]; // slicing and assignment
- const offset = (1,1); // tuple (no type given)
- [ij in ProbSpace[2..n-1, 2..n-1]] F(ij) = B(ij + offset); // ZPL style!
- [b in B] b = -b; // All the array!
- const Z: [ProbSpace] complex = [(i,j) in ProbSpace] i + j*1.0i;

```
var VarDom = {1..n};
var VarArr: [VarDom] real = [i in VarDom] i;
writeln("Initially, VarArr = ", VarArr, "\n");

// Now, if we reassign VarDom, VarArr will be reallocated with the
// old values preserved and the new values initialized to the element
// type's default value.
VarDom = {1..2*n};
writeln("After doubling VarDom, VarArr = ", VarArr, "\n");

// As mentioned before, this reallocation preserves values according
// to index, so if we extend the lower bound of the domain, the
// non-zero values will still logically be associated with indices
// 1..n:
VarDom = {-n+1..2*n};
writeln("After lowering VarDom's lower bound, VarArr = ", VarArr, "\n");

// If the domain shrinks, values will be thrown away
VarDom = {2..n-1};
writeln("After shrinking VarDom, VarArr = ", VarArr, "\n");
```

```
var Y: [ProbSpace] [1..3] real;

forall ((i,j), k) in [ProbSpace, 1..3] do
  Y(i,j)(k) = i*10 + j + k/10.0;

writeln("Y is:\n", Y);
```

Indefinite Domains ...

- var People: domain(string):
- var Age: [People] int;
- People += "John";
- Age("John") = 62;
- see primers/opaque.chpl

Data distribution control

${CHAPEL_HOME}/examples/primers/distributions.chpl

```
use BlockDist, CyclicDist, BlockCycDist, ReplicatedDist;
config const n = 8;
const Space = {1..n, 1..n};
const BlockSpace = Space dmapped Block(boundingBox=Space);
var BA: [BlockSpace] int;

forall ba in BA do
  ba = here.id;
writeln("Block Array Index Map");
writeln(BA);
writeln();
```

Other primers:

- atomic.chpl -- use of atomic variables
- domains.chpl -- more methods to deal with domains
- sparse.chpl -- sparce domains and arrays
- locales.chpl -- more local manipulations
- timers.chpl -- timing your code
- taskParallel.chpl -- cobegin, coforall features
- reductions.chpl -- more on reductions in chapel
- classes, fileIO, genericClasses, iterators, procedures, slices ...

chplvis

Tool to help visualize computation

- Can be turned on or off at run time
- Run results in data file that help you visualize computations
- Simple use: chplvis1.chpl
  - use VisualDebug;
  - startVdebug("filename");
  - stopVdebug();
- chplvis2.chpl
  - tagVdebug("name")
  - pauseVdebug()
- chplvis3.chpl -- Jacobi implementation
- chplvis4.chpl -- tasks
- Controling from command line
  - config constant VisualDebugOn
  - compile constant DefaultVisualDebugOn (default true)

Chapel procs

Procedures declarations:

proc name ([formal arguments]) [: return type] { .... }

- Formal arguments: name : type
  - call by value
  - arguments may be given a default value

```
class Circle {  var center : Point;  var radius : real; }
proc create_circle(x = 0.0, y = 0.0, diameter = 0.0)
{
  var result = new Circle();
  result.radius = diameter / 2;
  result.center.x = x;
  result.center.y = y;
  return result;
}
```

- Calls: args can be named

```
var c = create_circle(diameter=3.0,2.0);
```

- Return type not required to be declared

- Arguments of unknown type

```
proc unknownArg(x)
{
  writeln(x);
  if x.type == int then
    writeln("I see you've passed me an integer!");
  else if x.type == string {
    writeln("I liked that last variable so much, I'll write it again!");
    writeln(x);
  }
}
var intArg = 5;
var strArg = "Greetings, procedure unknownArg!";
var boolArg = false;
writeln("Using generic arguments");
unknownArg(intArg);
unknownArg(strArg);
unknownArg(boolArg);
writeln();
```

- Query operator ... get the domain of something (Not always needed.)

```
proc invertArray(ref A: [?D] int): [D] int{
  for a in A do a = -a;
  return A;
}
```

Chapel procs (page 3)

- Query the type of something

```
proc genericProc(arg1 : ?valueType, arg2 : valueType): void {
  select(valueType) {
    when int do writeln(arg1, " and ", arg2, " are ints");
    when real do writeln(arg1, " and ", arg2, " are reals");
    otherwise writeln(arg1, " and ", arg2, " are somethings!");
  }
}
```

- Arguments by default are passed as "constants"

```
proc simple ( a : int ) { a = 10; } // Compile error
```

- Argument intents: in, inout, out, ref
  - proc simple (in a : int) { a = 10; } // local change
  - proc simple (inout a : int) { a = 10; } // copies back final value
  - proc simple (out a : int) { a = 10; } // a is 0, not value of arg, final value passed back
  - proc simple (ref a : int) { a = 10; } // assignment sets actual parameter, no waiting
- Can return a reference to a type:

```
proc refElement(ref array : [?D] ?T, idx) ref : T { return array[idx]; }
```

- Procedure loverloading is available
  - types of args must be different types in required parameters

- Operator overloading:

```
operator ^(left : bool, right : bool): bool {
  return (left || right) && !(left && right);
}
```

- Another one

```
record MyRecord { var value: int; }
operator *(left : MyRecord, right : int): int { return left.value*right; }
```

- Yet again another way

```
record Point { var x, y: real; }
operator Point.+(p1: Point, p2: Point) { return new Point(p1.x + p2.x, p1.y + p2.y); }
```

- Be very careful ... following destroys things!

```
operator +(left: int, right: int): int { return left - right; }
```

Chapel iterators

Related to a procedure ... but returns multiple values ... at multiple times

- runs for a while, returns a vaule, and then runs later, starting at same point!
- Helps complicated sequences ...
- Uses the keyword "yield" to return a value
- Regular "return" or off the end stops the iterator

```
iter oddsThenEvens(N: int): int {
  for i in 1..N by 2 do
    yield i; // yield values instead of returning.
  for i in 2..N by 2 do
    yield i;
}
```

- Lots of things are similar, args, intents, ...
- Use is most often used in loops

```
for i in oddsThenEvens(10) do write(i, ", "); writeln();
```

- It is possible to yield nothing
- State may be saved in variables

```
iter fibonacci(n: int) {
  var (current, next) = (0, 1);
  for 1..n {
    yield current;
    (current, next) = (next, current + next);
  }
}
```

Iterators (page 2)

- Zip operator (mainly used with ranges and iterators)

  for (i, j) in zip(-5..-1, 1..5) { writeln ("(",i,",",j,")"); }

  - Both ranges must be same length
- Iterator and an indifinite range

  for (i, j) in zip(fibonacci(10), 1..) { ....
- Yield tuples:

```
iter multiloop(n: int) {
  for i in 1..n do
    for j in 1..n do
      yield (i, j);
}
```

- Look at itr-tree.chpl here -- more complex example

Parallel iterators

- Iterator used in a forall loop!
- Two types ... "standalone" and "leader-follower"
- Also, parallel iterator requires a serial version that is overloaded
- parIters.chpl here (vdebug decorated)
- tree.chpl

Other random topics:

- timer: .start(), .stop(), .elapsed(), .clear()
- sync { begin {...}; begin {...}; begin {...} ... }
  - This is similar to a cobegin ... but slightly different.
- atomic variables:
  - var atom : atomic int;
  - var.write(), var.read(), var.waitFor(value)
- sync variables : two states, empty and full
  - Read an empty or write a full, wait for other one.
  - .readFE(), .writeEF(val), .writeXF(true) (full or empty matters)
- retuction: op reduce listOfValues (produces one value)
- scan: op scan listOfValues (produces array of values)
- here.gpus.size, on here.gpus[id] -- see example/gpus