

Today's Plan

Higher level languages

- HPF
- ZPL

ZPL

Philosophy

- Provide performance portability for data-parallel programs
- Allow users to reason about performance
- Start from scratch
 - Parallelism is fundamentally different from sequential computing
 - Be willing to **throw out** conveniences familiar to sequential programmers

Basic idea

- An array language
- Implicitly parallel

ZPL History

The beginning

- Designed by a small team beginning in 1993
- Compiler and runtime released in 1997

Claims

- Portable to any MIMD parallel computer
- Performance comparable to C with message passing
- Generally outperforms HPF
- Convenient and intuitive

CS380P Lecture 15

Introduction to ZPL

3

Recall Our Example Computation

Jacobi Iteration

- The elements of an array, initialized to 0.0 except for 1.0's along its southern border, are iteratively replaced with the average of their 4 nearest neighbors until the greatest change between two iterations is less than some epsilon.

| | | | | | |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 1 | |

CS380P Lecture 15

Introduction to ZPL

4

Jacobi Iteration– The Main Loop

```
program Jacobi;
config    var n : integer = 512;
          epsilon : float = 0.00001;
region
    R      [1..n, 1..n];
var       A, Temp : [R] float;
          err : float;

direction north = [-1, 0];           south = [ 1,  0];
        east  = [ 0,  1];           west   = [ 0, -1];

prv
```

Naming Convention:

Arrays begin with upper case letters

Scalars begin with lower case letters

R

max<< returns
of an array

```
[north of R] A := 0.0; [south of R] A := 0.0;
[east of R]  A := 0.0; [west of R]  A := 0.0;
```

Naming Convention:
Arrays begin with upper case letters
Scalars begin with lower case letters

Reductions:
`max<<` returns the maximum
of an array expression

```
repeat
  Temp := (A@north + A@east + A@west + A@south)/4.0;
  err := max<< abs(Temp - A);
  A := Temp;
until err < epsilon;
```

end;
end;
 $\text{grid} := (\text{blue_grid} + \text{green_grid} + \text{purple_grid} + \text{red_grid}) / 4.0$

CS380P Lecture 20

Introduction to ZPL

5

Jacobi Iteration– The Region

```

program Jacobi;
config    var n : integer = 512;
          epsilon : float = 0.00001;

region
var      R = [1..n, 1..n];
var      A, Temp : [R] float;
          err : float;

direction north = [-1, 0];      south = [ 1,  0];
          east  = [ 0, 1];      west  = [ 0, -1];

procedure Jacobi();
[R] begin
    A := 0.0;
[north of R] A := 0.0; [west of R]  A := 1.0;
[east of R]  A := 0.0; [south of R] A := 0.0;

    repeat
        Temp := (A@north + A@east + A@west + A@south)/4.0;
        err  := max<< abs(Temp - A);
        A    := Temp;
    until err < epsilon;
end;
end;

```

The diagram illustrates the Jacobi iteration formula for a 4x4 grid. The formula is: $A := (A_{\text{north}} + A_{\text{east}} + A_{\text{west}} + A_{\text{south}}) / 4.0$. The grid is divided into four quadrants, each representing a neighbor's value: North (blue), East (green), West (purple), and South (red).

CS380P Lecture 15

Introduction to ZPL

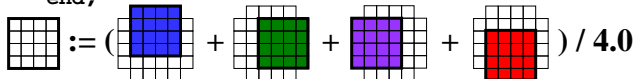
6

Jacobi Iteration– The Direction

```

program Jacobi;
config  var n : integer = 512;
        epsilon : float = 0.00001;
region  R = [1..n, 1..n];
var     A, Temp : [R] float;
        err : float;
direction north = [-1, 0];    south = [ 1,  0];
        east  = [ 0, 1];    west  = [ 0, -1];
procedure Jacobi();
[R] begin
    A := 0.0;
[north of R] A := 0.0; [west of R] A := 1.0;
[east of R] A := 0.0; [south of R] A := 0.0;
    repeat
        Temp := (A@north + A@east + A@west + A@south)/4.0;
        err := max<< abs(Temp - A);
        A := Temp;
    until err < epsilon;
end;
end;

```



CS380P Lecture 13

introduction to ZPL

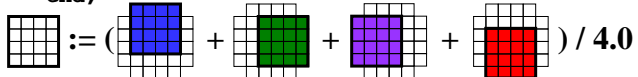
7

Jacobi Iteration– The Border

```

program Jacobi;
config  var n : integer = 512;
        epsilon : float = 0.00001;
region  R = [1..n, 1..n];
var     A, Temp : [R] float;
        err : float;
direction north = [-1, 0];    south = [ 1,  0];
        east  = [ 0, 1];    west  = [ 0, -1];
procedure Jacobi();
[R] begin
    A := 0.0;
[north of R] A := 0.0; [west of R] A := 1.0;
[east of R] A := 0.0; [south of R] A := 0.0;
    repeat
        Temp := (A@north + A@east + A@west + A@south)/4.0;
        err := max<< abs(Temp - A);
        A := Temp;
    until err < epsilon;
end;
end;

```



CS380P Lecture 15

introduction to ZPL

8

Jacobi Iteration– Remaining Details

```

program Jacobi;
config  var n : integer = 512;
        epsilon : float = 0.00001;
region  R = [1..n, 1..n];
var     A, Temp : [R] float;
        err : float;
direction north = [-1, 0];    south = [ 1,  0];
        east  = [ 0, 1];    west  = [ 0, -1];
procedure Jacobi();
[R] begin
    A := 0.0;
[north of R] A := 0.0; [west of R] A := 1.0;
[east of R] A := 0.0; [south of R] A := 0.0;
    repeat
        Temp := (A@north + A@east + A@west + A@south)/4.0;
        err := max<< abs(Temp - A);
        A := Temp;
    until err < epsilon;
end;
end;

```

CS380P Lecture 15 Introduction to ZPL 9

Arrays and Scalars

Scalar promotion

- Arrays can be promoted to scalars

```
Temp := (A@north + A@east + A@west + A@south) / 4.0;
```

- Scalars assume the shape of the arrays that they're promoted to

- Functions can also be promoted

```
abs(Temp - A)
```

- The **abs** function is applied to each element of the array expression
- Function promotion provides some amount of code reuse

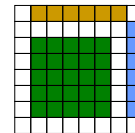
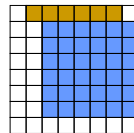
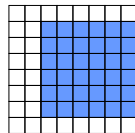
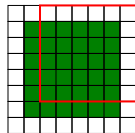
Regions Are Declarative

Regions state *what*, not *how*

- Regions are index sets
- Regions and region operators (**of**, **at**, **in**, ...) replace indexing and looping to simplify programming

```
region R = [1..8, 1..8];
region C = [2..7, 2..7];
var X,Y = [R] integer;
```

```
e = [ 0, 1];
n = [-1, 0];
ne = [-1, 1];
```



[C] X := ■

[C] Y@e := ■

[n of C] Y := ■

[C] Y := X@ne ■

CS380P Lecture 15

Introduction to ZPL

11

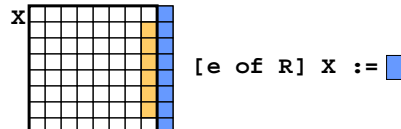
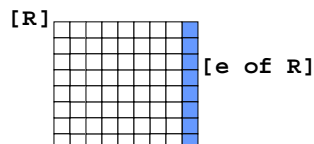
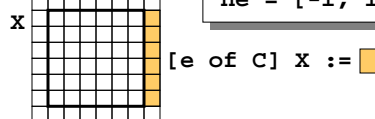
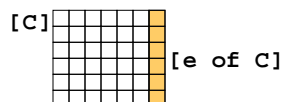
The of Operator

Defining adjacent regions

- The of operator defines a region adjacent to the given region in the specified direction

```
region R = [1..8, 1..8];
region C = [2..7, 2..7];
var X,Y = [R] integer;
```

```
e = [ 0, 1];
n = [-1, 0];
ne = [-1, 1];
```



CS380P Lecture 15

Introduction to ZPL

12

Region Operators

Simple region calculus

- A dense r -dimensional region can be specified by pairs of upper and lower limits: $\langle L_1, U_1 \rangle, \langle L_2, U_2 \rangle, \dots, \langle L_r, U_r \rangle$
- If direction $d = (d_1, d_2, \dots, d_r)$ and $R = \langle L_1, U_1 \rangle, \langle L_2, U_2 \rangle, \dots, \langle L_r, U_r \rangle$
- Then we have the following:

$$R \text{ at } d = \langle L_1 + d_1, U_1 + d_1 \rangle, \langle L_2 + d_2, U_2 + d_2 \rangle, \dots, \langle L_r + d_r, U_r + d_r \rangle$$

d of R satisfies

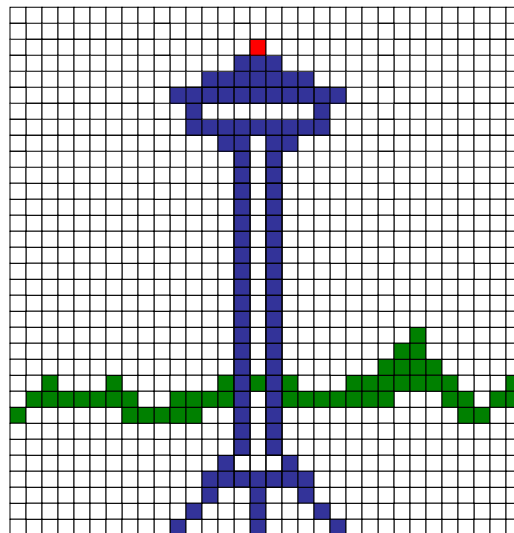
$$\langle L'_i, U'_i \rangle = \begin{cases} \langle U_i + 1, U_i + d_i \rangle & \text{if } d_i > 0 \\ \langle L_i, U_i \rangle & \text{if } d_i = 0 \\ \langle L_i + d_i, L_i - 1 \rangle & \text{if } d_i < 0 \end{cases}$$

CS380P Lecture 15

Introduction to ZPL

13

Class Exercise



CS380P Lecture 15

Introduction to ZPL

14

Regions as Indices

Applied to statements

- The region r prepended to a statement gives the indices over which all computation on rank r arrays is applied
 $[Rr] \dots Ar + Br \dots$
- By applying regions to entire statements, array references are identical by default
- Regions are scoped, i.e., a region on an inner statement overrides a region on an outer statement

```
[1..n] begin . . .
    [2..n-1] . . . A + B . . .
end;
```
- Regions can be **dynamic**, i.e., bounds are evaluated at runtime
 $[i..j] \dots A + B \dots$

CS380P Lecture 15

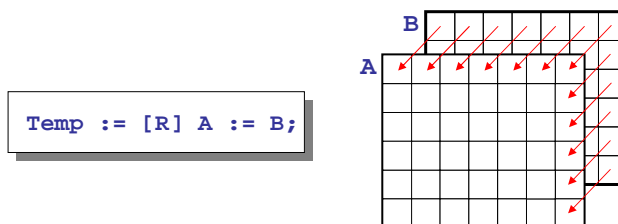
Introduction to ZPL

15

How Do We Get Parallelism?

Array language semantics

- Assign the rhs elements to the lhs elements in any order



- There are implied barrier between statements and between the evaluation of the rhs and the lhs

CS380P Lecture 15

Introduction to ZPL

16

Global Operators

Reductions and scans

- Reductions (<<) and scans (||) are array functionals that perform global operations

+<< *A* reduces an array expression *A* to its sum, producing a scalar

```
+<< 2 4 6 8 ≡ 20
```

+|| *A* computes the parallel prefix of *A*, producing an array of the same size as *A*

```
+|| 2 4 6 8 ≡ 2 6 12 20
```

| Reduce | Scan |
|--------|------|
| +<< | + |
| *<< | * |
| max<< | max |
| min<< | min |
| &<< | & |
| << | |

Other Language Details

Data types

```
boolean
ubyte  sbyte  char
integer  uinteger
float  double  quad
```

Unary operators

```
+  -  !
```

Binary operators

```
+  -  *  /  ^  %  &
```

Relational operators

```
=  !=  <  >  <=  >=
```

Bitwise operators

```
. . .
```

Assignment operators

```
. . .
```

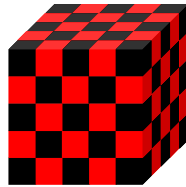
Control constructs

```
if-then-{elsif}-else
repeat-until
while-do
for-do
exit
return
continue
halt
begin-end
```

Consider an Example: Red/Black SOR

Compute partial differential equations

- Use successive over-relaxation
- Arrange 3D values into red and black cells
- Update in place by alternately computing values for red and black cells



CS380P Lecture 15

Introduction to ZPL

19

Two Implementations of Red/Black SOR

Can you spot the bugs?

Regions and region operators raise the level of abstraction

```

for nrel := 1 to nITER do
  /* Red relaxation */
  [I with Red]    U := factor*(hsq*F + U@top  + U@bot  + U@left+
                                U@right + U@front + U@back);
  /* Black relaxation */
  [I without Red] U := factor*(hsq*F + U@top  + U@top  + U@left+
                                U@right + U@front + U@back);
end;
  
```

ZPL

```

DO nrel = 1, iter
  where (RED(2:NX-1, 2:NY-1, 2:NZ-1))
!   Relaxation of the Red points
    U(2:NX-1, 2:NY-1, 2:NZ-1) =
      & factor*(hsq*F(2:NX-1, 2:NY-1, 2:NZ-1)+
      & U(1:NX-2, 2:NY-1, 2:NZ-1) + U(3:NX, 2:NY-1, 2:NZ-1)+
      & U(2:NX-1, 1:NY-2, 2:NZ-1) + U(2:NX-1, 1:NY-2, 2:NZ-1)+
      & U(2:NX-1, 2:NY-1, 1:NZ-2) + U(2:NX-1, 2:NY-1, 3:NZ))
    elsewhere
!   Relaxation of the Black points
    U(2:NX-1, 2:NY-1, 2:NZ-1) =
      & factor*(hsq*F(2:NX-1, 2:NY-1, 2:NZ-1)+
      & U(1:NX-2, 2:NY-1, 2:NZ-1) + U(3:NX, 2:NY-1, 2:NZ-1)+
  
```

F90/HPF

Comparing HPF and ZPL

What's the difference in the two codes?

- We cheated by not showing the definition of the Red mask in ZPL

More fundamentally

- Indexing is error prone
- **Different things should look different**
 - With the explicit indices, everything looks similar
 - Why is this important?
- **Abstraction principle**
 - If something is important, then it should be given a name and reused
 - Regions and directions support provide abstraction for data-parallel computation

CS380P Lecture 15

Introduction to ZPL

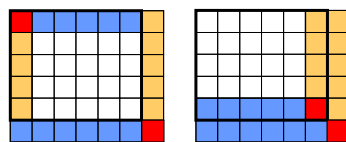
21

Boundary Conditions

Data parallelism

- Often quite regular except for the

ZPL elevates the concept of a bound



periodic

mirror

The shallow benchmark

```
/* Periodic boundary conditions
[ e of I ] wrap U, Uold, V, Vold
[ s of I ] wrap U, Uold, V, Vold
[ se of I ] wrap U, Uold, V, Vold
```

C Periodic boundary conditions

```
uold(m+1,:n) = uold(1,:n)
vold(m+1,:n) = vold(1,:n)
pold(m+1,:n) = pold(1,:n)
u(m+1,:n) = u(1,:n)
v(m+1,:n) = v(1,:n)
p(m+1,:n) = p(1,:n)
CAPR$ DO PAR on POLD<:,1>
  uold(:m,n+1) = uold(:m,1)
  vold(:m,n+1) = vold(:m,1)
  pold(:m,n+1) = pold(:m,1)
  u(:m,n+1) = u(:m,1)
  v(:m,n+1) = v(:m,1)
  p(:m,n+1) = p(:m,1)
  uold(m+1,n+1) = uold(1,1)
  vold(m+1,n+1) = vold(1,1)
  pold(m+1,n+1) = pold(1,1)
  u(m+1,n+1) = u(1,1)
  v(m+1,n+1) = v(1,1)
  p(m+1,n+1) = p(1,1)
```

HPF

CS380P Lecture 15

Introduction to ZPL

22

What's Different About ZPL?

Concise

- High level array language
- Sequential semantics

The Simple benchmark

| ZPL | Fortran77 | C+MPI |
|-----------|------------|------------|
| 500 lines | 2400 lines | 5000 lines |

Clean

- Eliminates array indexing
- Support for boundary conditions

Efficient

- Provides special support for structured communication
 - “Optimize the common case”
- More on this next class

Next Time

Lecture

- Using ZPL
- We'll start at 2:30 for March 20 only