Parallel & Concurrent Programming: ZPL

Emery Berger

CMPSCI 691W

Spring 2006

Outline

- Previously:
  - **MPI** – point-to-point & collective
    - Complicated, far from problem abstraction
  - **OpenMP** - parallel directives
    - Language extensions to Fortran/C/C++
    - Questionable semantics, error-prone
- Today:
  - Something **way** better: ZPL

lecture material from ZPL project, UW

ZPL

- **Parallel array** language
  - Implicitly parallel
    - No parallel constructs *per se*
  - Very high level
    - Assignments work at array level, as in A := B + C
  - Machine independent
    - Compiles to ANSI C + communication library calls (e.g., MPI)
  - Efficient

Comparison

- Matrix-multiplication:
  - C
    - triply-nested loop

```
for (i=0;i<n;i++){
  for (j=0;j<n;j++){
    for (k=0;k<n;k++){
      c[i][j]=c[i][j]
            +a[i][k]*b[k][j];
    }
  }
}
```

```
[1..n,1..n] for k := 1 to n do
                C +=  (>>[,k] A) * (>>[k,] B);
            end;
```

  - ZPL
    - dot-product of rows & columns
    - efficiently implemented on parallel machines

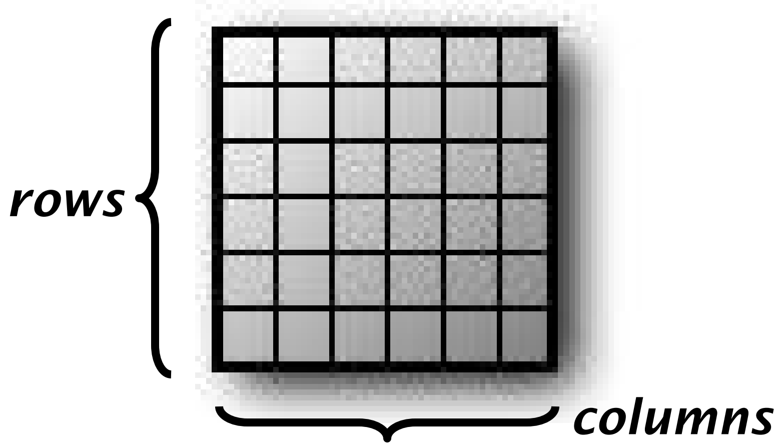
ZPL Outline

- Language overview
  - Regions
  - Directions
  - Parallel array operations
  - Handling boundary conditions
- ZPL programs & performance

Regions

- Key abstraction in ZPL: **regions**
  - **Index sets** (*rows,cols*) partition matrices
  - Operate on regions, not indexed items!

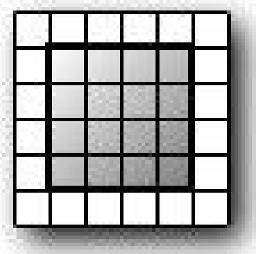
```
region
  R = [1..n,1..n];
```



Region Examples

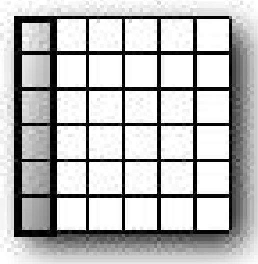
- **Interior** of matrix

```
region
  IntR =
    [2..n-1,2..n-1];
```



- **Left-most column**

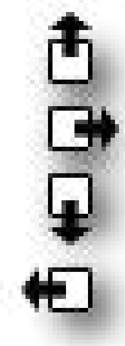
```
region
  Left = [1..n,1];
```

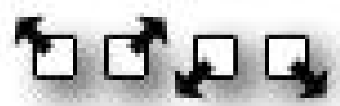


Directions

- **Directions**:
  - Offset vectors used to manipulate regions & array data

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

  nw = [-1,-1];
  ne = [-1, 1];
  sw = [ 1,-1];
  se = [ 1, 1];
```





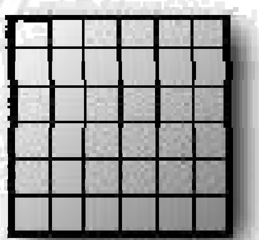
Creating New Regions

- **Prepositions** create new regions:
  - **in**
    - Applies direction to select part of region
  - **of**
    - Creates new region outside existing region
  - **at**
    - Shifts a region by a direction
  - **by**
    - Creates new region **strided** by direction

Applying Directions

- Use **“in”** to apply direction to region

```
region
  R = [1..n,1..n];
```

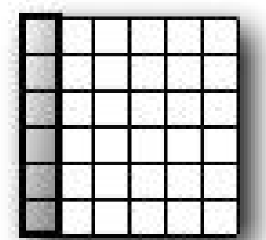


+

```
direction

  west  = [ 0,-1];
```

=

```
region
  Left = west in R;
```



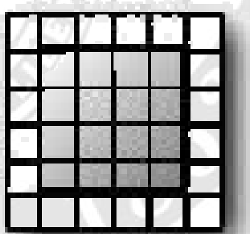
Create Region Outside

- Use **“of”** to create region **outside** existing region
  - Extends region

Shifting Regions

- Use **"at"** to create new region **shifted** by a direction

```
region
  IntR =
    [2..n-1,2..n-1];
```

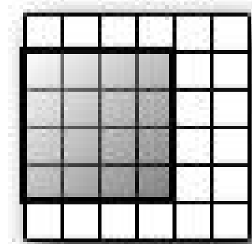


+

```
direction

  west  = [ 0,-1];
```

=

```
region
  IntRLeft =
    IntR at west;
```

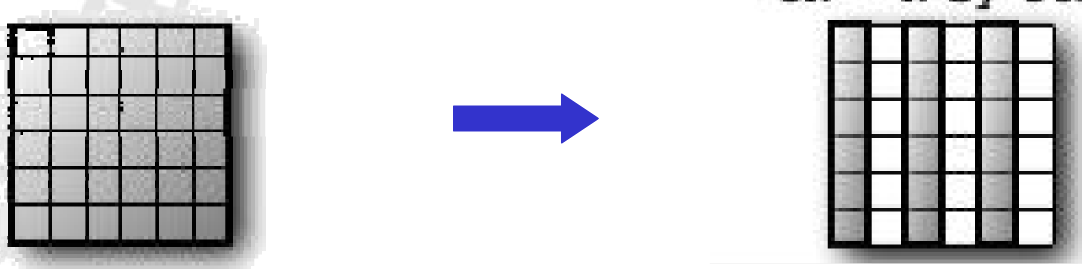


Striding Regions

- Use **“by”** to create new region **strided** by a direction

```
region
  R = [1..n,1..n];
```

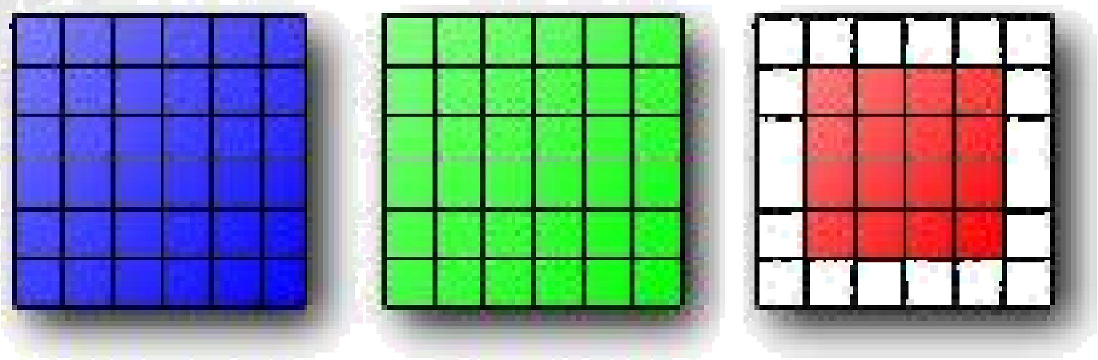
```
direction
  step = [1,2];

region
  SR = R by step;
```



Parallel Arrays

- Parallel arrays **declared over regions**

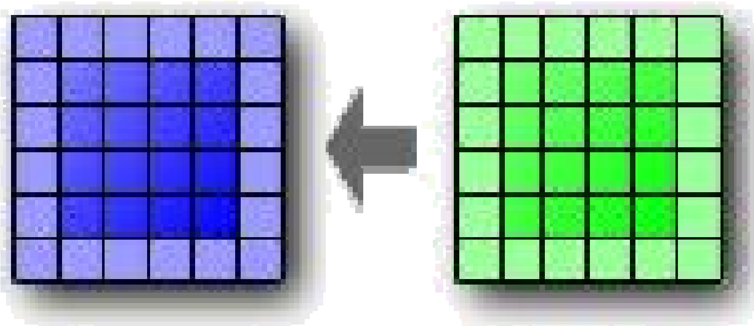
```
var
  A, B : [R] double;
  C    : [IntR] double;
```



Computing Over Arrays

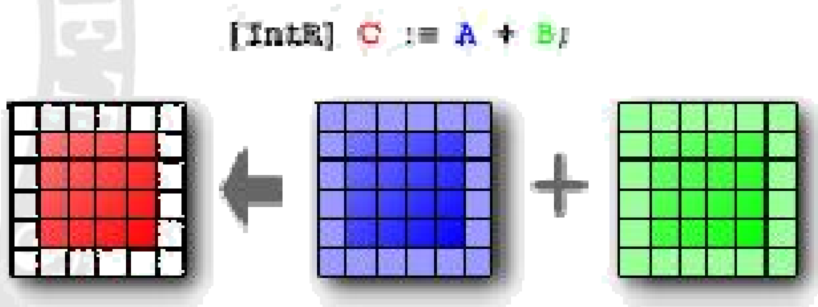
- Can use regions as **modifiers** that define computations over arrays:

```
[IntR] A := B;
```



Arrays & Communication

- Most computations in ZPL
  do not involve communication

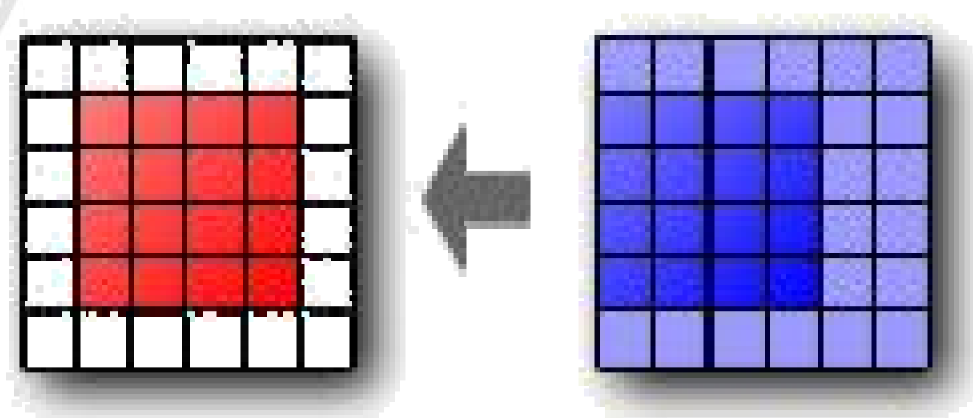


- Exceptions include:
  - **Shifting**
  - **Reduction**
  - **Broadcast**
  - **All-to-all**

Shifting Arrays

- **@** operator **shifts** data in direction
  - This translation induces **point-to-point** communication

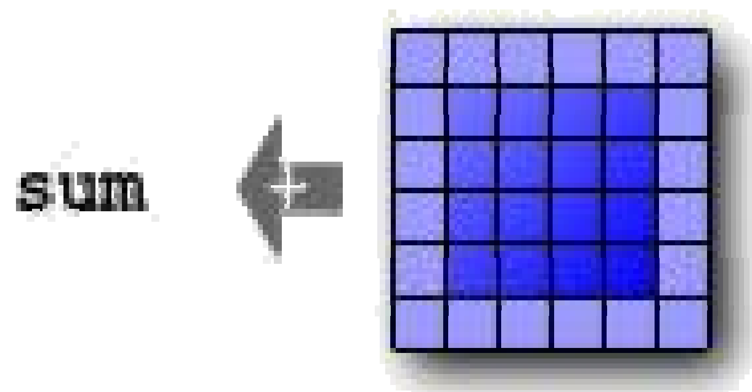
```
[IntR] C := A@west;
```



Reduction

- **Op<<** computes reductions
  - Reduction (tree-style) communication
    - +<< (sum), *<< (times), min<<, max<<…
  - For prefix (scan), use **op||**

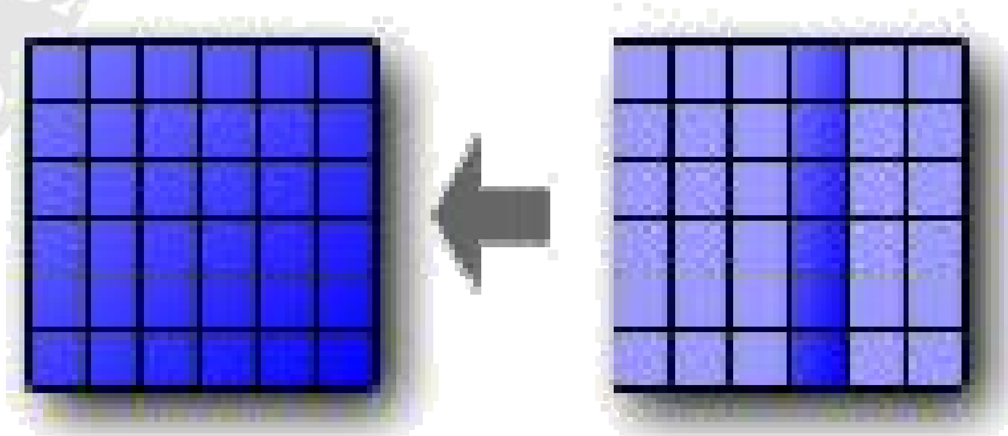
```
[IntR] sum := +<< A;
```



Broadcast (Flooding)

- **>> (flood)** replicates data across dimensions of array
  - Triggers broadcast operation

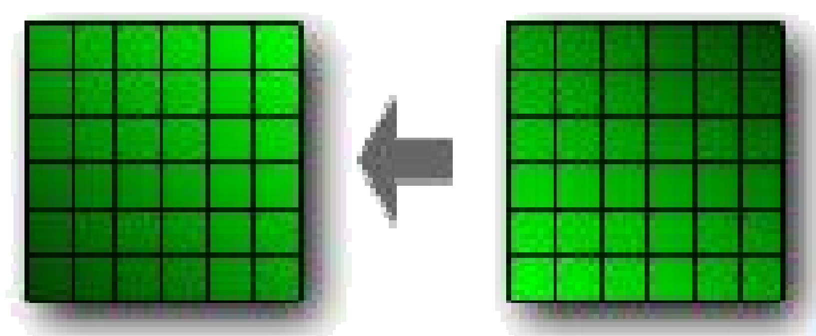
```
[R] A := >>[1..n,i] A;
```



Mapping

- **Remap (#)** moves data between arrays
  - Specified by “map” arrays
  - Built-in **Index1, Index2**
    - **Index1** = row indices, **Index2 =** col indices

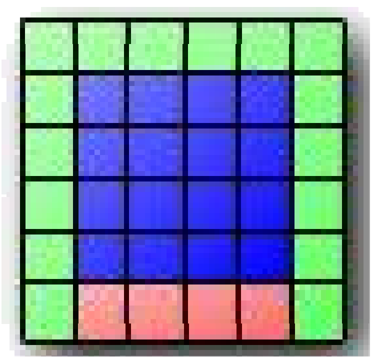
```
[R] B := B#[Index2,Index1];
```



Boundary Conditions

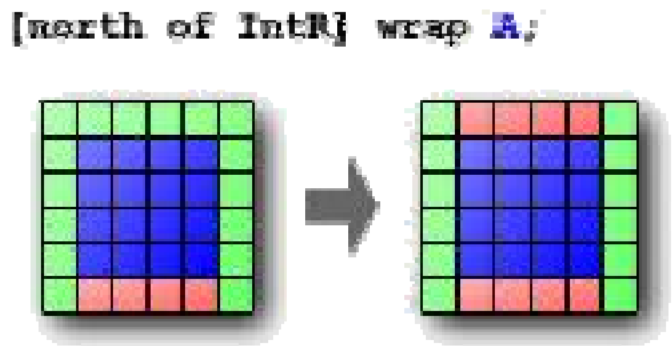
- Boundary conditions ("corner cases")
  - Usually tedious, error-prone
  - **Very** simple in ZPL

```
[north in R]    A := 0.0;
[east in R]     A := 0.0;
[west in R]     A := 0.0;
[south of IntR] A := 1.0;
```



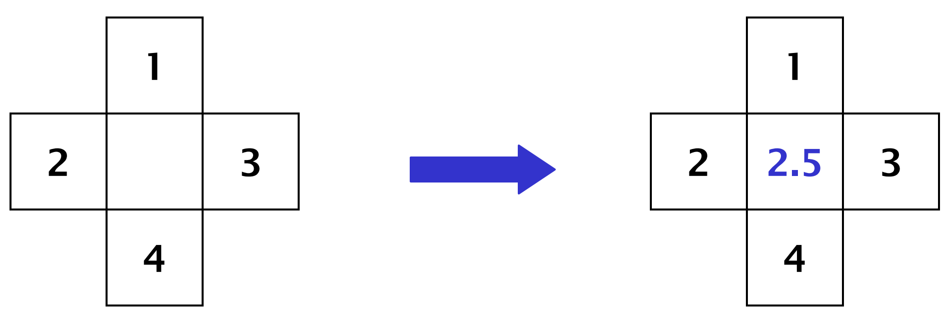
Boundary Conditions

- **Periodic** boundary conditions with **wrap**



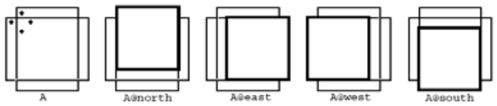
ZPL Example

- **Jacobi iteration** – replace elements in array with average of four nearest neighbors, until largest change < δ
  - Consider difficulty of parallelizing with MPI/OpenMP
    - boundary conditions, etc.

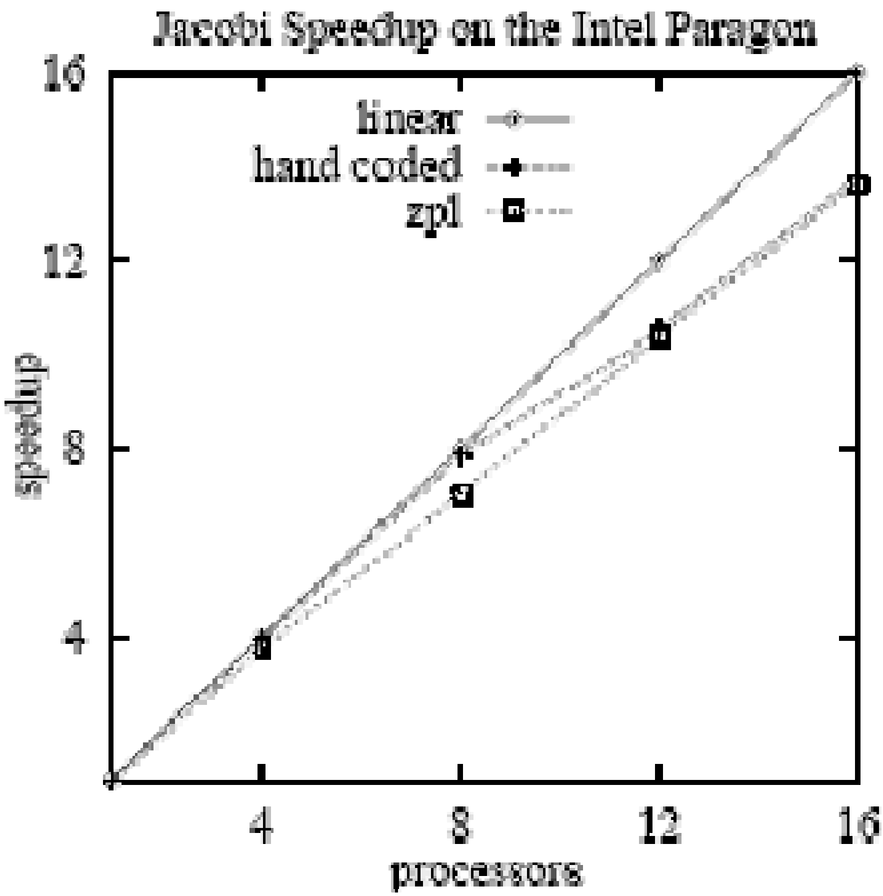


ZPL Example

```
program Jacobi;
/*                    Jacobi Iteration
               Written by L. Snyder, May 1994      */
config var     n     : integer    = 512; -- Declarations
               delta : float      = 0.000001;

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

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

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

 [R]           repeat                   -- Body
                 Temp := (A@north + A@east
                         + A@west + A@south)/4.0;
                 err  := max<< abs(A - Temp);
                 A    := Temp;
               until err < delta;
end;
```



ZPL Performance



ZPL Example: Life

Conway's Game of Life simulates cells which can live, die, and reproduce according to the following rules:

Rule I. (Survival) A cell survives only if it has two or three live neighbors.

Rule II. (Birth) A cell is born in any empty square with exactly three live neighbors.

ZPL Example: Life

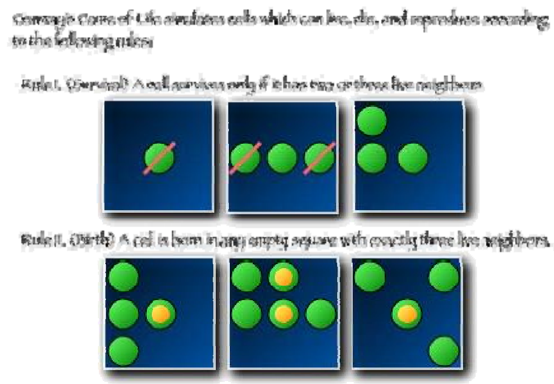
```
program Life;

config var
  n : integer = 100;

region
  BigR = [0..n+1, 0..n+1];
  R    = [1..n,   1..n  ];

direction
  nw   = [-1, -1]; north = [-1, 0]; ne   = [-1, 1];
  west = [ 0, -1];                  east = [ 0, 1];
  sw   = [ 1, -1]; south = [ 1, 0]; se   = [ 1, 1];

var
  TW : [BigR] boolean; -- The World
  NN : [R]    integer; -- Number of Neighbors

procedure Life();
begin
  -- Initialize the world
  [R] repeat
    NN := TW@nw   + TW@north + TW@ne   +
          TW@west +            TW@east +
          TW@sw   + TW@south + TW@se;
    TW := (TW & NN = 2) | (NN = 3);
  until !(|<< TW);
end;
```

A configuration variable can be set on the command line.

Count the live neighbors.

Update the world.

Is this a bleak metaphor or what?



The End

- Next time:
  - Your turn!
  - Occam & Multilisp