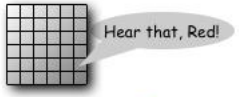
Tired of MPI?

The Pocket Guide to ZPL

ZPL is a parallel array programming language.

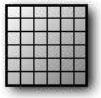
The region is the key abstraction underlying ZPL arrays.



6 See back for footnotes.

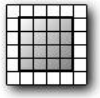
Regions are index sets. Region `R` is an n x n index set.

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



Region `IntR` specifies `R`'s interior indices.

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



Region `Left` is the first column of `R`.

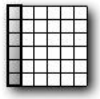
```
region
  Left = [1..n,1];
```

Directions are offset vectors used to manipulate regions and 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];
```

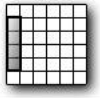
The preposition "in" creates a new region using a direction.

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



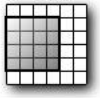
The "of" preposition creates a new region on the outside.

```
region
  SmallLeft =
    west of IntR;
```



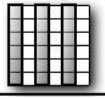
The "at" preposition creates a new region shifted by a direction.

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



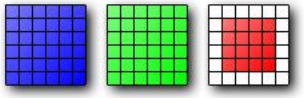
The "by" preposition creates a new region strided by a direction.

```
direction
  step = [1,2];
region
  SR = R by step;
```



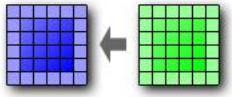
Parallel arrays are declared over regions.

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



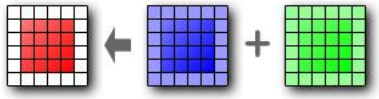
Regions control computation on parallel arrays.

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



All communication is explicit. There is no communication here.

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

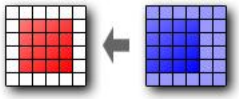


Only a small set of parallel operators induce communication. This provides a powerful performance model.

7

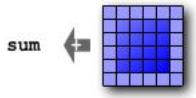
The at operator (@) shifts data in a direction, inducing point-to-point communication.

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



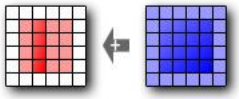
The reduce operator (op<<) computes reductions using a combining operator and induces reduction communication..

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



A partial reduction collapses dimensions of an array.

```
[2..n-1,i] C := +<<[IntR] A;
```



Reductions can be computed with many built-in operators. User-defined reductions are supported on associative and commutative functions.

```
+<< (sum)    min<< (min) |<< (or)
*<< (times) max<< (max) &<< (and)
bor<<  (bitwise or)
band<< (bitwise and)
myreduce<< (user-defined)
```

2

For parallel-prefix scans, change "<<" to "||."

Biological computer simulation codes, such as Conway's Game of Life, are easy to write in ZPL.

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.

```
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;
```

It's a Complete ZPL Program!

A configuration variable can be set on the command line.

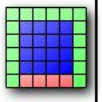
Count the live neighbors.

Update the world.

Is this a bleak metaphor or what?

Boundary conditions are easy to initialize and manipulate in ZPL.

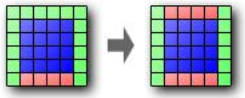
```
[north in R]     A := 0.0;
[east in R]      A := 0.0;
[west in R]      A := 0.0;
[south of IntR] A := 1.0;
```



This is usually a tedious and error-prone part of parallel scientific computing.

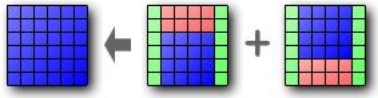
Even periodic boundary conditions are supported. Just use the **wrap** operator.

```
[north of IntR] wrap A;
```



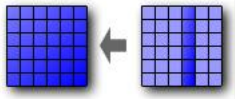
The wrap-at operator (@^) also considers periodic boundaries.

```
[R] A := A@^north + A@^south;
```



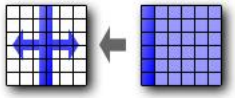
The flood operator (>>) replicates data across one or more dimensions of an array.

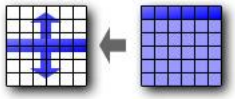
```
[R] A := >>[1..n,i] A;
```



Flood dimensions (*) efficiently store replicated data. We can take the cross product of the first column and row of an array easily.

```
var
  Col : [1..n,*] double;
  Row : [*,1..n] double;

[1..n,*] Col := >>[1..n,1] A;
[*,1..n] Row := >>[1,1..n] A;
[R] B := Col * Row;
```





The **Indexi** arrays are built-in constants which conform to the compute region..

```
[1..3, 1..3] begin
                ... Index1 ...;
(1,1) (1,2) (1,3)
(2,1) (2,2) (2,3)
(3,1) (3,2) (3,3)
... Index2 ...;
end;
```

```
1 1 1
2 2 2
3 3 3
```

```
1 2 3
1 2 3
1 2 3
```

A one line solution: "[R] B := >>[1..n,1] A * >>[1,1..n] A;"

Unlike APL, ZPL doesn't have hundreds of operators, though there are a few more.

Phew!

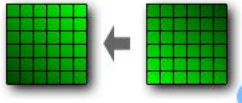
Each parallel operator has different communication requirements.

Review	
@, @^	point-to-point
<<	reduction
||	parallel prefix
>>	broadcast
Next Concept	
#	all-to-all

One more!

The remap operator (#) moves data between arrays as specified by map arrays. Arrays can be transposed with the **Indexi** constants.

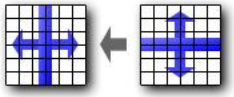
```
[R] B := B#[Index2,Index1];
```



1

Flooded arrays can also be transposed. Conveniently, the **Indexi** arrays are flooded in all but the **i**th dimension.

```
[1..n,*] Col := Row#[Index2,Index1];
```



Note that **Index2**'s use is legal since **Row** is flooded in the first dimension.

Parallel Matrix Multiplication is a well-studied problem. Three well-known algorithms are easy to write in ZPL. 1. Cannon's Algorithm.

Given parallel arrays A, B, and C, matrix multiply A and B to get C.

```
region                    var
  R = [1..n, 1..n];         A, B, C : [R] double;

direction
  right = [0, 1];
  below = [1, 0];

var
  i : integer;

[R] begin
  A#[Index1,((Index2-Index1+n)%n)+1] := A;
  B#[((Index1-Index2+n)%n)+1,Index2] := B;
  C := A * B;
  for i := 2 to n do
    A := A@^right;
    B := B@^below;
    C += A * B;
  end;
end;
```

Cannon's Algorithm

Skew first.

Then, repetitively shift and multiply.

2. SUMMA Algorithm. 3. PSP or Bulk Communication Algorithm.

```
[R] begin
  C := 0.0;
  for i := 1 to n do
    C += >>[ ,i] A * >>[i, ] B;
  end;
end;
```

SUMMA Algorithm

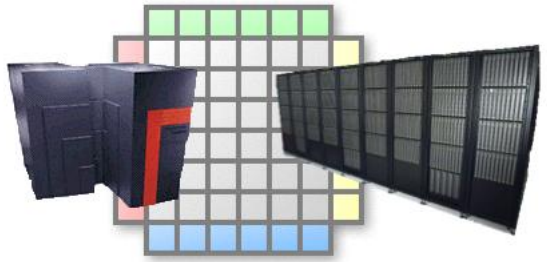
```
region
  IJ  = [1..n, 1..n, *   ];
  JK  = [*,    1..n, 1..n];
  IK  = [1..n, 1,    1..n];
  IJK = [1..n, 1..n, 1..n];

var
  A3 : [IJ] double;
  B3 : [JK] double;
  C3 : [IK] double;

[IJ] A3 := A#[Index1, Index2];
[JK] B3 := B#[Index2, Index3];
[IK] C3 := +<< [IJK] (A3 * B3);
[R]  C := C3#[Index1, 1, Index2];
```

5

PSP or Bulk Algorithm

ZPL is optimized for high performance and runs on clusters as well as custom-designed supercomputers.

3 4



Don't wait! Download your copy today!
http://www.cs.washington.edu/research/zpl

References

1. The Design and Implementation of a Parallel Array Operator for the Arbitrary Remapping of Data.
   S. J. Deitz, B. L. Chamberlain, S.-E. Choi and L. Snyder.
   In ACM Conference on Principles and Practice of Parallel Programming, 2003.

2. High-level Language Support for User-defined Reductions.
   S. J. Deitz, B. L. Chamberlain and L. Snyder.
   In The Journal of Supercomputing, 23(1), 2002.

3. A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures.
   B. L. Chamberlain, S. J. Deitz and L. Snyder.
   In ACM Conference on Supercomputing, 2000.

4. ZPL: A Machine Independent Programming Language for Parallel Computers.
   B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby.
   In IEEE Transactions on Software Engineering, 26(3), 2000.

5. Problem Space Promotion and Its Evaluation as a Technique for Efficient Parallel Computation.
   B. L. Chamberlain, E. C. Lewis and L. Snyder.
   In ACM International Conference on Supercomputing, 1999.

6. Regions: An Abstraction for Expressing Array Computation.
   B. L. Chamberlain, E. C. Lewis, C. Lin and L. Snyder.
   In ACM International Conference on Array Programming Languages, 1999.

7. ZPL's WYSIWYG Performance Model.
   B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder and W. D. Weathersby.
   In IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments, 1998.

ZPL Research Group
Department of Computer Science
University of Washington
Box 352350
Seattle, WA 98195-2350
zpl-info@cs.washington.edu
http://www.cs.washington.edu/research/zpl