The Performance Model of ZPL

In the same way the von Neumann machine explains the performance of languages like C, the CTA explains the performance of ZPL. But to "understand" the explanation, programmers must be taught how the features of the language run on the abstract machine.

1

One Solution to Game of Life ...

```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];
direction  N = [-1, 0]; NE = [-1, 1];
           E = [ 0, 1]; SE = [ 1, 1];
           S = [ 1, 0]; SW = [ 1,-1];
           W = [ 0,-1]; NW = [-1,-1];
var   Ncount : [R] byte;
          TW : [R] boolean;
procedure Life();
      [R] begin
           TW := (Index1 * Index2) % 2; -- Make some data
           repeat
             Ncount := (TW@^N + TW@^NE + TW@^E + TW@^SE
                        + TW@^S + TW@^SW + TW@^W + TW@^NW);
                 TW := (Ncount=2 & TW) | (Ncount=3 & !TW);
           until false;
          end;
end;
```

2

Drainage

- Given an nxn array A, “plot” the drainage map from the highest to the lowest point, if it exists
- Declarations …

```
program drainage;
config var n : integer = 50;
region     R = [1..n,1..n];
direction no = [-1, 0]; ne = [-1, 1];
          ea = [ 0, 1]; se = [ 1, 1];
          so = [ 1, 0]; sw = [ 1,-1];
          we = [ 0,-1]; nw = [-1,-1];
var        A : [R] integer;
 Tu,Td,Up,Dn : [R] Boolean;
```

3

Drainage (continued)

Boundaries not included

```
procedure drainage();
[R] begin
    Up := 0; Dn := 0;
    read(A);
    Up := A=(min<<A);
    Dn := A=(max<<A);
    repeat
      Tu := Up; Td := Dn;
      Dn := Dn | (A<=A@no & Dn@no) | (A<=Ane & Dn@ne)
               | (A<=A@ea & Dn@ea) | (A<=Ase & Dn@se)
               | (A<=A@so & Dn@so) | (A<=Asw & Dn@sw)
               | (A<=A@we & Dn@we) | (A<=Anw & Dn@nw);
      Up := Up | (A>=A@no & Up@no) | (A>=Ane & Up@ne)
               | (A>=A@ea & Up@ea) | (A>=Ase & Up@se)
               | (A>=A@so & Up@so) | (A>=Asw & Up@sw)
               | (A>=A@we & Up@we) | (A>=Anw & Up@nw);
    until ! |<<((Tu != Up) | (Td != Dn));
    write(Up&Dn);
   end;
```

Moving down

Moving up

4

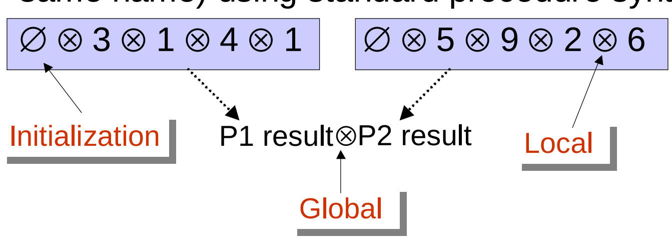
User Defined Reductions

Reduction -- combining elements with an associative operator in a tree computation -- is a powerful paradigm, but the operators (+, *, max, min, &, |) are limited, so allow users to define their own operators

- Examples --
  - Find the largest n elements
  - Find the largest element and its index (maxi, mini...)
  - Find the value closest to 4
  - ...

Three Components

- Reduction **`op<<A`** has three parts:
  - Initialization … first step is **`initial_val op A[1]`**
  - Local Combine … **`intermed_val op A[i]`**
  - Global Combine … **`intermed_val op intermed_val`**
- Users define all three (overloaded with the same name) using standard procedure syntax



Largest Two Elements

max<<A finds the largest … find the largest two

```
type bestn = array[1..2] of double;
```

```
procedure max2(var big2 : bestn);
  var i : integer;
  begin   for i := 1 to 2 do
            big2[i] := MINDOUBLE;
          end;
  end;
```

Initialization

```
procedure max2(nextval : double; var big2 : bestn);
  var i : integer, temp : double;
  begin   for i := 1 to 2 do
            if nextval > big2[i]then
              temp    := big2[i];
              big2[i] := nextval;
              nextval := temp;
            end;
          end;
  end;
```

Local Combining

```
procedure max2(var big2l, big2r : bestn);
  var i : integer, temp : double;
  begin   for i := 1 to 2 do
            max2(big2l[i],big2r);
          end;
  end;
```

Global Combining

… max2<< A ...

Permute

ZPL's permute operation allows data to be moved around arbitrarily

Old syntax	New syntax	
`A:=<##[I1, I2] B;`	`A:=B#[I1,I2];`	Gather
`A:=>##[I1, I2] B;`	`A#[I1,I2]:=B;`	Scatter

The lists in brackets are arrays of indices s.t. the i,jth element comes from /goes to I1[i,j],I2[i,j]

Referring to the new syntax, the intuitive idea is that the simple array is enumerated, and the #-array is referenced from the given indices

Permute (continued)

- To transpose an array, simply write

```
[1..n,1..n] A:=A#[Index2, Index1];
```

```
                    1 2 3   1 1 1
[1..3,1..3] A := A#[1 2 3 , 2 2 2]
                    1 2 3   3 3 3
```

- To reverse … `[1..n] V := V#[n-Index1+1]`
- The use of "computable" subscripts is common and leads to optimizations

Permute (continued)

- 1-to-1 applications of # (permutations) can be programmed using either gather or scatter
- When the indices are not all different, the behavior differs
  - [1..n] V := V#[n]; -- assign every element V[n]
  - [1..n] V#[n] := V; -- unpredictable

In recent months permute has been upgraded

- Syntax
- Combining
- Rank change
- Serious Optimizations

ZPL's got lots more ...

"Core ZPL" has been presented, but there is more

- Hierarchical arrays
- Pipelining (scan)
- Sparse Arrays
- Local processing & control (unrestricted)
- (Will have) Task parallelism
- (Will have) Irregular data structures
- (Will have) User controlled load balancing

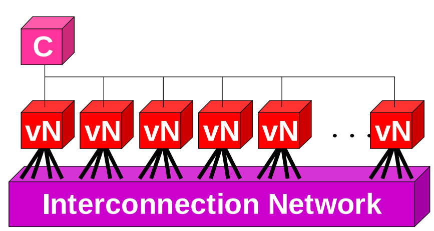
… and greater unification

ZPL code is generally faster than the custom message-passing programs written by professionals

11

ZPL Built On The CTA

- Recall that ZPL uses the CTA as its underlying machine model … like C uses vN
  - CTA explains cost of the language's operations (How?)
  - CTA provides a model for designing the compiler and the run-time system
- The CTA
  - P processors
  - λ >>1 latency to nonlocal memory
  - Unspecified interconnect



12

The Task ...

How is ZPL's performance model given?

- Specify how
  - processors are allocated to computation
  - regions (and arrays) are allocated in memory
  - rules of operation for primitive ZPL facilities including costs for computation and communication
- Assure that all of the source language features are explained
- Explain the interactions with optimizations

13

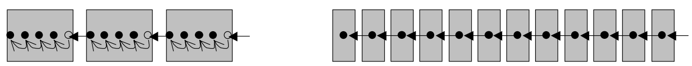
Assigning Work To Processors

- Two views for data parallel computation, .e.g. `A := B+C@east` as found in ZPL's dense arrays
  - Virtual Processors: Each processor is allocated the work of scalar data values, that is, think of a situation where there are as many processors as array elements
  - Variable Points Per Processor: Each processor is allocated from 1 to n values, but usually many
    - 1 element is virtual processor case
    - n elements is the sequential case
    - Algorithms with this feature are scalable

14

Assigning Work (continued)

- Many claim the two models are equivalent, since virtual can be emulated
- They're not equivalent because ...
  - Emulation misses the advantages of long instruction sequences
    - pipelining
    - caching
    - prefetch
  - Virtual misses significant costs of local shifting: **`V:=V@right`**



15

ZPL Assumes Many Points Per Processor

ZPL allocates regions (and therefore arrays) to processors so many contiguous elements are assigned to each

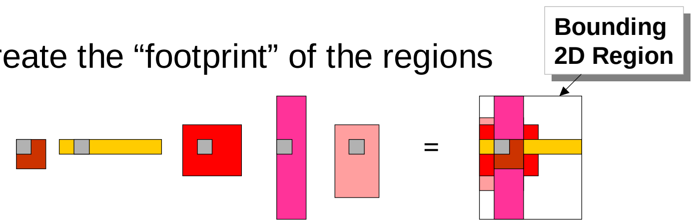
- Array Allocation Rules
  - Union the regions together computing the *bounding region*
  - Get processor number and arrangement from command line
  - Allocate the bounding region to the processors

Consider a walk-through of the process

16

Union The Regions Together

Create the “footprint” of the regions



Technical point: Only interacting regions are “unioned,” e.g. if region R is used to declare an array which is manipulated in the scope of region S, R and S are said to *interact*

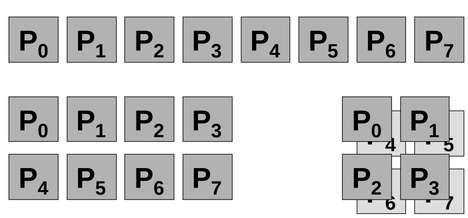
The bounding region is allocated to processors

Get Processor Number + Arrangement

The number of processors and their arrangement is given by the programmer on the command line

- For the purpose of [understanding] allocation, processors are viewed as being arranged in grids … this is simply an abstraction:

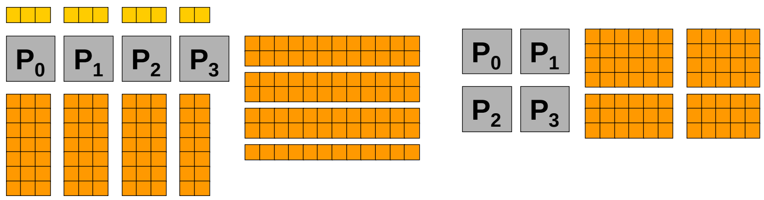
The CTA does not favor any arrangement, so use a generic one



Allocate the Bounding Region to the Grid

The bounding region is allocated to the grid in the "most balanced" arrangement possible

- Regions inherit their position from the bounding region
- Array elements inherit their positions from their index's position in the region
- 1D is segmented; 2D is panels, strips or blocks; 3D ...



ZPL allocates multiple elements per processor

19

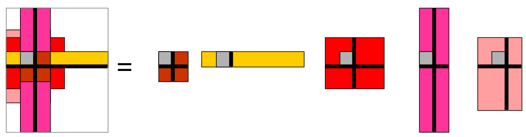
Break

20

Fundamental Fact of ZPL

Such allocations are mostly standard, but one fact makes ZPL performance clear:

ZPL has the property that for any arrays **`A, B`** of same rank and having an element **`[i, …, k]`**, that element of each will be stored on the same processor



Corollary: Element-wise operations do not require any communication: **`[R] … A+B …`**

Performance Model (WYSIWYG)

To state how ZPL performs operations, each operator's work and communication needs are given … producing a performance model

- Performance is given in terms of the CTA
- Performance is relative, e.g. *x* is more expensive in communication than *y*

- Rules…

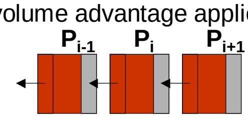
`A + B` -- Element-size array operations

- No communication
- Work comparable to C
- Work fully parallelizable, i.e. time = work/P

Rules Of Operation (continued)

`A@east` -- @ references including **`wrap`**

- Nearest neighbor point-to-point communication of edge elements, i.e. the surface-to-volume advantage applies
- Local data motion, possibly

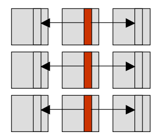


`+<<` -- Reduce and scan

- Accumulate local elements
- Ladner/Fischer O(log P) tree accumulation
- Broadcast, which is worst case O(log P), but usu. less

`>> [1..n,k] A` -- Flood

- Multicast array segments
- Represent data non-redundantly



Rules of Operation (continued)

`<##[I1, I2]` -- Permutation

- (Potential) all-to-all processors communication to distribute routing information implied by **`I1, I2`**
- (Potential) all-to-all processors communication to route the elements of A

- Full information on all of ZPL in Chapter 8 of *ZPL Programmer's Guide*
- "What you see is what you get" performance model … large performance features visible

ZPL is only parallel language with performance model

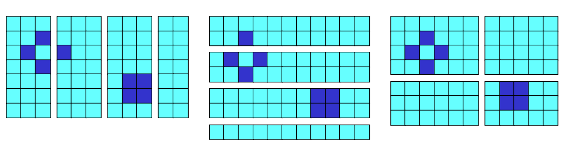
Applying The WYSIWYG Model To Life...

```
program Life;
config var n : integer = 512;
region      R = [1..n, 1..n];
         BigR = [0..n+1,0..n+1];
direction   N = [-1, 0]; NE = [-1, 1];
            E = [ 0, 1]; SE = [ 1, 1];
            S = [ 1, 0]; SW = [ 1,-1];
            W = [ 0,-1]; NW = [-1,-1];
var   Ncount : [R] byte; TW : [BigR] boolean;
procedure Life();
      [R] begin
            TW := (Index1 * Index2) % 2; -- Make some data
            repeat
          Ncount := (TW@N + TW@NE + TW@E + TW@SE
                   + TW@S + TW@SW + TW@W + TW@NW);
              TW := (Ncount=2 & TW) | (Ncount=3 & !TW);
            until false;
           end;
end;
```


Analyzing Life By Color

- Blue: Effectively no time … each processor does set-up and scalar computation locally
- Gold: Element-wise computation perfectly parallel … **`Index`***i* constants are generated

How is TW allocated on 4 procs? Three basic choices...

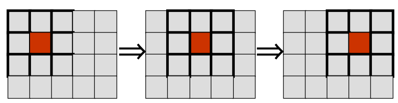


Delay is cλ

Analyzing By Color (continued)

- Red: Element-wise computation with @ operations … expect λ delay for @ (though it may be combined with wrap) and then full parallel speed-up for local operations

- That's the worst case … optimizations are possible … for example stencil optimizations



7 additions are used for each element, but fewer adds are sufficient

27

Summarizing WYSIWYG Model

- Data and processing allocations are given
- All constructs of the language are explained in terms of the allocations and the CTA
- Result: relative, worst-case statement of how the computation runs anywhere … rely on it
- Optimizations can improve on the times, but if they don't fire, nothing is lost

The best use of the WYSIWYG model is to make comparative programming decisions

28

Comparing Cannon's and SUMMA MM

- If you have to write MM, what algorithm do you want to use?
- Analyze the choices with WYSIWYG …
- Recall that the two algorithms have the structure:

```
Cannon's
Declare
Skew A
Skew B
Initialize
loop til n
C+=A*B
Rotate A,B
```

```
SUMMA
Declare
Initialize
loop til n
flood A
flood B
C+=A*B
```

29

Comparing Cannon's and SUMMA MM

- Step one is to cancel out the equivalent parts of the two solutions … they'll work the same
- For MM the comparison reduces to whether the initial skews and the iterated rotates are more/less expensive than iterated floods

<u>Cannon's</u>
~~Declare~~
Skew A
Skew B
~~Initialize~~
~~loop til n~~
~~C+=A*B~~
Rotate A,B

<u>SUMMA</u>
~~Declare~~
~~Initialize~~
~~loop til n~~
flood A
flood B
~~C+=A*B~~

30

Programming Cannon's In ZPL

- Handle the skewed arrays

```
c11 c12 c13              a11 a12 a13 a14
c21 c22 c23  <----   a21 a22 a23 a24
c31 c32 c33      a31 a32 a33 a34
c41 c42 c43  a41 a42 a43 a44
     ^
     |   b13   c11 c12 c13   a11 a12 a13 a14
     b12 b23   c21 c22 c23   a22 a23 a24 a21
b11 b22 b33    c31 c32 c33   a33 a34 a31 a32
b21 b32 b43    c41 c42 c43   a44 a41 a42 a43
b31 b42        b11 b22 b33
b41            b21 b32 b43
               b31 b42 b13
               b41 b12 b23
```

Pack skewed arrays into dense arrays by rotation; process all n^2 elements at once

Four Steps of Skewing A

```
          for i := 2 to m do
[i..m, 1..n] A := A@^right;   -- Shift last m-i rows left
          end;
```

… And Skew B vertically

```
a11 a12 a13 a14        a11 a12 a13 a14
a21 a22 a23 a24        a22 a23 a24 a21
a31 a32 a33 a34        a32 a33 a34 a31
a41 a42 a43 a44        a42 a43 a44 a41
```

Initial — i = 2 step

```
a11 a12 a13 a14        a11 a12 a13 a14
a22 a23 a24 a21        a22 a23 a24 a21
a33 a34 a31 a32        a33 a34 a31 a32
a43 a44 a41 a42        a44 a41 a42 a43
```

i = 3 step — i = 4 step

32

Cannon's Declarations

For completeness, when A is m×n and B is n×p, the declarations are …

```
region       Lop = [1..m, 1..n];
             Rop = [1..n, 1..p];
             Res = [1..m, 1..p];
direction right = [ 0, 1];
          below = [ 1, 0];
var            A : [Lop] double;
               B : [Rop] double;
               C : [Res] double;
```

33

Cannon's Algorithm

Skew A, Skew B, {Multiply, Accumulate, Rotate}

```
           for i := 2 to m do -- Skew A
  [i..m, 1..n] A := A@^right;
           end;
           for i := 2 to p do -- Skew B
  [1..n, i..p] B := B@^below;
           end;

         [Res] C := 0.0;       -- Initialize C
           for i := 1 to n do -- For common dim
         [Res] C := C + A*B;  -- For product
         [Lop] A := A@^right; -- Rotate A
         [Rop] B := B@^below; -- Rotate B
           end;
```

34

Cannon's Algorithm

Skew A, Skew B, {Multiply, Accumulate, Rotate}

```
          for i := 2 to m do -- Skew A
[i..m, 1..n] A := A@^right;
          end;
          for i := 2 to p do -- Skew B
[1..n, i..p] B := B@^below;
          end;

        [Res] C := 0.0;        -- Initialize C
          for i := 1 to n do -- For common dim
        [Res] C := C + A*B;  -- For product
        [Lop] A := A@^right; -- Rotate A
        [Rop] B := B@^below; -- Rotate B
          end;
```

Comms have λ latency, but much data motion

SUMMA Algorithm Analysis

The flood is more expensive than λ time, but less that λ(log P) ... probably much less

```
[1..m,1..p]    C := 0.0;        -- Initialize C
          for k := 1 to n do
   [1..m,*]  Col := >>[ ,k] A; -- Flood kth col of A
   [*,1..p]  Row := >>[k, ] B; -- Flood kth row of B
[1..m,1..p]    C += Col*Row;    -- Combine elements
          end;
```

SUMMA does not require as much comm or data motion as Cannon's, nor does it "touch" the array as much

A Parallel Programming Technique

ZPL's approach admits new programming tricks

Problem Space Promotion (PSP) is a parallel programming technique in which d-dimension data is processed by solving the problem in a higher dimension d' > d

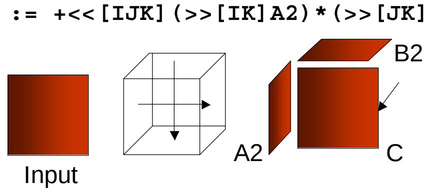
- Flooding (logically) replicates the data
- Intermediate data structures need not be built, i.e. PSP is space efficient
- Greater parallelism than the control flow solution
- Less synchronous solution

37

3D MM Is A Problem Space Promo Alg

The 2D arrays are multiplied in a logical 3D space … flooding creates the logical set-up

```
region IK = [1..n,*,1..n]
       JK = [*,1..n,1..n];
       IJ = [1..n,1..n,*];
      IJK = [1..n,1..n,1..n];
[IK] A2 := A#[Index1,Index3,Index2];
[JK] B2 := B#[Index3,Index2,Index1];
[IJ]  C := +<<[IJK](>>[IK]A2)*(>>[JK]B2);
```



38

Sorting By PSP

To sort, compute the position in the output by counting the number of elements smaller than

```
[1,1..n] begin
[1..n,1]    ST := S#[Index2,Index1]; -- Transpose input
             P := +<<[1..n,1..n](>>[*,1..n]S <= >>[1..n,*]ST);
                              -- Compare n^2 items, reduce
             S := S#[Index1,P]; -- Reorder by permutation
          end;
```

S

3	1	4	5	9	2

ST

3
1
4
5
9
2

	3	1	4	5	9	2
3	1	0	1	1	1	0
1	1	1	1	1	1	1
4	0	0	1	1	1	0
5	0	0	0	1	1	0
9	0	0	0	0	1	0
2	1	0	1	1	1	1

P

3	1	4	5	6	2

S

1	2	3	4	5	9

Summary

- The CTA explains how ZPL works when we explain how the operations of the language are implemented (by the language designers and compiler writers) on the CTA
- The WYSIWYG performance model is ZPL's way of giving that information
  - Allocation of data and processing are specified
  - Work is specified
  - Communication is specified
- We've shown how to apply WYSIWYG to assess alternative algorithms