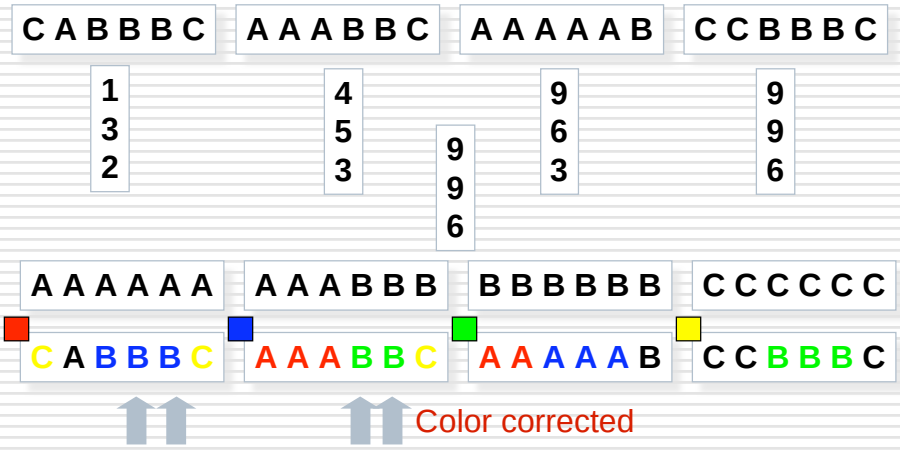
# NII Lectures:Next Languages -- Access To Parallelism For All

Lawrence Snyder
www.cs.washington.edu/homes/snyder

15 October 2008

## Example

- Assume 4 processors



http://phase.hpcc.jp/Omni/benchmarks/NPB/

# OpenMP Compiler

- 4 Processor Sun Enterprise running the NAS PB written in C with OpenMP

Block Tridiagonal
Conjugate Gradient
Embarrassingly ||
Fast Fourier Trans
Integer Sort
LU Decomposition
Multigrid Iteration
Sparse Matrix-Vector

| Program | Class | 1 thread | 2 threads | 4 threads |
|---|---|---|---|---|
| BT | W | 119.19 (1.00) | 61.28 (1.95) | 36.65 (3.25) |
| | A | 2900.02 (1.00) | 1546.70 (1.87) | 1024.93 (2.83) |
| CG | W | 14.61 (1.00) | 6.05 (2.41) | 3.12 (4.68) |
| | A | 49.65 (1.00) | 26.01 (1.91) | 15.14 (3.28) |
| EP | W | 33.36 (1.00) | 16.74 (1.99) | 8.45 (3.95) |
| | A | 267.39 (1.00) | 133.73 (2.00) | 67.98 (3.93) |
| FT | W | 6.07 (1.00) | 3.20 (1.90) | 1.85 (3.28) |
| | A | 113.96 (1.00) | 60.55 (1.88) | 34.73 (3.28) |
| IS | W | 0.76 (1.00) | 0.47 (1.62) | 0.38 (2.00) |
| | A(*1) | 17.05 (1.00) | 9.25 (1.84) | 5.81 (2.93) |
| LU | W | 194.90 (1.00) | 101.42 (1.92) | 54.43 (3.58) |
| | A | 1810.94 (1.00) | 775.63 (2.33) | 411.07 (4.41) |
| MG | W | 13.56 (1.00) | 6.58 (2.06) | 3.34 (4.06) |
| | A | 101.29 (1.00) | 50.68 (2.00) | 26.67 (3.80) |
| SP | W | 329.05 (1.00) | 175.04 (1.88) | 110.83 (2.97) |
| | A | 2127.84 (1.00) | 1157.58 (1.84) | 762.07 (2.79) |

# Outline

- Context
- Global View Languages
  - NESL
  - ZPL
- ZPL details including SUMMA
- Transparent Performance Model

## The Present Situation

- Previous lectures have argued that
  - We have tremendous opportunities to use ||ism
  - Much is known about parallel computation
  - There is (at least) one “right” machine model
  - Today’s programming facilities are weak, having few abstractions
- What do we want? What can we practically expect in the future?
- Consider a few basic goals …

## Using Parallelism

- “Parallelism is best when it is unseen” -- Calvin Lin, UT Austin
  - Processors are amazingly parallel
  - Parallel techniques are applied widely in OSs
  - Web search (Google) and other Internet services apply parallelism that we all use
- But writing fast || programs requires using the CTA machine model and knowing how a || computer will run the program … how can ||ism be transparent?

## Making Parallelism Transparent

- Higher level abstractions --
  - Write less, but get more done
  - Rely on compilers to generate parallel code

**Contrast: Low level programming forms with compiler optimizations**

- Expose the machine's behavior by implicit means
  - Sound foundations for abstractions & compiler
  - Visible performance model

## Outline

- Context
- Global View Languages
  - NESL
  - ZPL
- ZPL details including SUMMA
- Transparent Performance Model

## Global View Languages

- There is a (pleasant) alternative to shared memory parallel programming -- global view languages
  - Global view means “seeing” all of the data
  - Opposite is local view: MPI, threads, PGAS,…
  - Technically, Global View means *P-independent*, all program executions produce same result regardless of the number or arrangement of processors
  - Higher abstractions available to simplify programming by eliminating nuisance details
  - Higher abstractions put premium on thinking rather than slamming out lines of code

## NESL

- NESL was developed by Guy Belloch at Carnegie Mellon (CMU)
- Key structure is a sequence
  - `[2 14 -5 0 7]`
  - `"sequences can be composed of characters"`
  - `["sequence" "elements" "can be sequences"]`
    provided all are composed of the same atomic type
- Basic operation is *apply to each*, written with set notation
  `{a+1: a in [2 13 0 4 8]}` produces `[3 14 1 5 9]`
  `{a+b: a in [1 2 3]; b in [8 7 6]}` produces `[9 9 9]`

## More on NESL

- Compare NESL dot product with UPC

```
function dotprod(a,b) = sum({x*y: x in a; y in b});
   dotprod([2, 3, 1], [6, 1, 4]);
```

producing `[19]`

- "Nested" in NESL refers to nested parallelism:
  - Applying parallelism and within each parallel operation, applying more parallelism
  - In NESL, *apply to each* ops in *apply to each*
  - Consider NESL's matrix multiplication algorithm

## MM in NESL

- The function is defined

```
function matrix_multiply(A,B)=
      {{sum({x*y : x in rowA; y in columnB})
                 :   columnB in transpose(B)}
                 :   rowA in A}
```

- Three *apply to each* braces
  - Outer brace applied to rowA, in ||
  - Next brace applied to columnB, transposed, in ||
  - Inner brace applied to each of $n^2$ row/col pairs

## NESL Complexity Model

- NESL researchers identify two types of complexity in a program:
  - *Work*, which is the number of basic operations
    - MM has O($n^3$) work; dotproduct has O($n$) work
  - *Depth*, which is the longest chain of dependences; e.g. sum has O($\log_2 n$) depth
    - Both MM and dotproduct have O($\log_2 n$) depth
- Like the PRAM, these metrics do not yield a performance model because they ignore $P$, $\lambda$, locality, etc.

## Outline

- Context
- Global View Languages
  - NESL
  - ZPL
- ZPL details including SUMMA
- Transparent Performance Model

## ZPL

- ZPL, a || language developed University of Washington: www.cs.washington.edu/research/zpl
- ZPL, a research parallel language w/ 3 goals
  - Performance == platform-specific custom code
  - Portability == runs well on all platforms
  - Convenience == clean, easy-to-understand programs; no parallel grunge
- ZPL is
  - A global view array language (ZPL != APL)
  - Implicitly parallel, so it is easy to use
  - Many new parallel abstractions

## ZPL Is Important To Us

- ZPL is a representative of a high-level parallel language … few competitors because achieving those goals is tough
- To realize a solution …
  - ZPL is designed and built on the CTA
  - ZPL is the first high-level language to achieve "performance portability"
  - ZPL presents programmers with a visually-cued performance model: WYSIWYG
  - ZPL is insensitive to shared or message passing architectures, making it universal

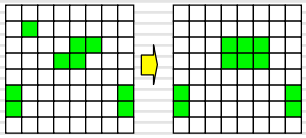
ZPL is "designed from first principles"

# A ZPL Example: Conway's Life

```
program Life;
config const n : integer = 10;
region R = [1..n, 1..n];
direction nw = [-1, -1]; no = [-1, 0]; ne = [-1, 1];
          w  = [ 0, -1];               e  = [ 0, 1];
          sw = [ 1, -1]; so = [ 1, 0]; se = [ 1, 1];
var  TW : [R] boolean;
     NN : [R] sbyte;
procedure Life();
begin -- Initialize the world
[R] repeat
      NN := TW@^nw + TW@^no + TW@^ne
          + TW@^w  +          TW@^e
          + TW@^sw + TW@^so + TW@^se;
      TW := (TW & NN = 2) | (NN = 3);
    until !(|<< TW);
end;
```

# Conway's Game of Life

- Life: organisms w/2,3 neighbors live, birth occurs w/ 3 neighbors; death otherwise; world is a torus

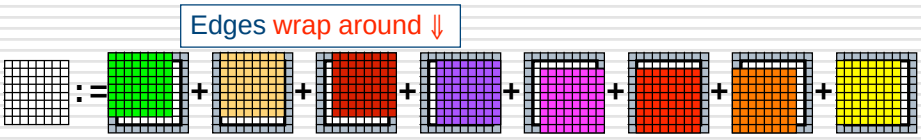


- TheWorld[i,j] == 1 in generation n+1
  - if [i,j] is 1 in generation n and has 2 neighbors or
  - if [i,j] in generation n has 3 neighbors
- Or: (thisGen && neighbors== 2) || (neighbors==3)

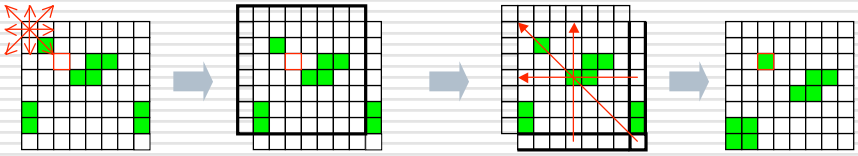
See Life As An Array Computation

# Global View of Life

- Count neighbors by adding organisms (bits)




- Closer look at World@^NW



**TW@^nw** is the array of Northwest neighbors

# Express Array Computation in ZPL

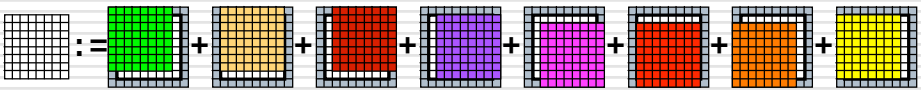
## Conway's Life: The World is bits

```
[1..n,1..n] repeat
              NN := TW@^NW + TW@^N  + TW@^NE
                  + TW@^W    +          TW@^E
                  + TW@^SW  + TW@^S  + TW@^SE;
             TW := (TW & NN = 2) | (NN = 3);
      until ! (|<< TW);
```

Add up neighbor bits

Apply rules to live by

"Or" bits in world to see if any alive



# Life In ZPL

```
program Life;                          Conway's Life
config const n : integer = 10;         The world is n × n; default to 10
region R = [1..n, 1..n];               Index set of computation
direction nw = [-1, -1]; no = [-1, 0]; ne = [-1, 1];
          w  = [ 0, -1];                    e  = [ 0, 1];
          sw = [ 1, -1]; so = [ 1, 0]; se = [ 1, 1];
var  TW : [R] boolean;                 Problem state, The World
     NN : [R] sbyte;                   Work array, Number of Neighbors
procedure Life();                      Entry point procedure
begin -- Initialize the world          I/O or other data specification
[R] repeat                             Region R ==> apply ops to all indices
      NN := TW@^nw + TW@^no + TW@^ne   Add 8 nearest neighbor bits  (type
          + TW@^w  +          TW@^e    coercion like C); carat(^) means
          + TW@^sw + TW@^so + TW@^se;  toroidal neighbor reference
      TW := (TW & NN = 2) | (NN = 3);  Update world with next generation
    until !(|<< TW);                   Continue till all die out
end;
```

# Outline

- Context
- Global View Languages
  - NESL
  - ZPL
- ZPL details including SUMMA
- Transparent Performance Model

## Regions, A Key ZPL Idea

- Regions are index sets … not arrays
- Any number of dimensions, any bounds

```
region V = [1..n];
region R = [1..m, 1..m]; BigR = [0..m+1,0..m+1];
region Left = [1..m, 1];
region Odds = [1..n by 2];
```

- Short names are preferred--regions are used everywhere--and capitalization is a coding convention
- Naming regions is recommended, but literals are OK

## Regions Control Computation

- Statements containing arrays need a *region* (index set) to specify which items participate

```
[1..n,1..n] A := B + C;
      [R] A := B + C;      -- Same as above if region R = [1..n,1..n]
```

- Regions are used to declare variables

```
var A, B, C : [R] double;
var Seq : [1..n] boolean;
```

- Regions are scoped

```
    [R] begin          All array computations in compound
          …            statements are performed over indices
[Left]    …            in [R], except statement prefixed by
        end;           [Left]
```

- Operations over region elements performed in parallel

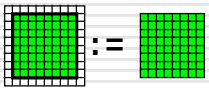
# Parallelism In Statement Evaluation

- Let A, B be arrays over [1..n,1..n], and C be an array over [2..n-1,2..n-1] as in

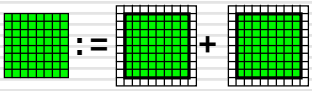
  var A, B : [1..n,1..n] float; C : [2..n-1,2..n-1] float;

- Then

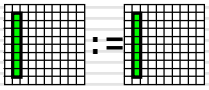
  [2..n-1,2..n-1] A := C;

  [2..n-1,2..n-1] C := A + B;

  [2..n-1,2] A := B;

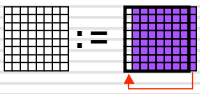
  
  

# @ Uses Regions & Directions

The @ operator combines regions with directions to allow references to neighbors

- Two forms, standard(@) and wrapping(@^)
  - Syntax: A@east A@^east
- Semantics: the direction is added to elements of region giving new region, whose elements are referenced; think of a region translation

  [1..n,1..n] A := A@^east; -- shift array left with wrap around

- @-modified variables can appear on l or r of :=

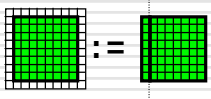
## Parallelism In Statement Evaluation

- Let

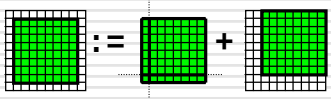
```
var A, B : [1..n,1..n] float; C : [2..n-1,2..n-1] float;
direction east = [0,1]; ne = [-1,1];
```

- Then

```
[2..n-1,2..n-1] A := C@^east;
```




```
[2..n-1,2..n-1] A := C@^ne + B@^ne;
```




```
[2..n-1,2] A@east := B;
```


## Reductions, Global Combining Operations

- Reduction (<<) "reduces" the size of an array by combining its elements
- Associative (and commutative) operations are +<<, *<<, &<<, |<<, max<<, min<<

```
[1..n, 1..n] biggest   := max<<A;
        [R] all_false := |<< TW;
```

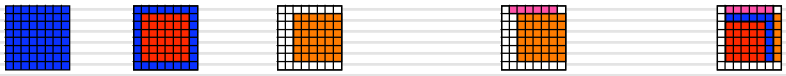
- All elements participate; order of evaluation is unspecified … caution floating point users
- ZPL also has partial reductions, scans, partial scans, and user defined reductions and scans

# Operations On Regions

- The importance of regions motivates region operators
- Prepositions: at, of, in, with, by … take region and direction and produce a new region
  - at translates the region's index set in the direction
  - of defines a new region adjacent to the given region along direction edge and of direction extent

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

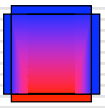
```
direction e = [ 0,1];
          n = [-1,0];
         ne = [-1,1];
```



`[R]X:=` `[C]X:=` `[C at e]Y:=` `[n of C]Y:=` `[C]Y:=X@ne`

execution

# Applying Ideas: Jacobi Iteration



- Model heat defusing through a plate
- Represent as array of floating point numbers
- Use a 4-point stencil to model defusing
- Main steps when thinking globally

Initialize
Compute new averages
Find the largest error
Update array
… until convergence

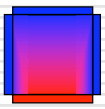
High-level Language should match high-level thinking

## “High Level” Logic Of J-Iteration

```
program Jacobi;
config var n : integer = 512;
           eps : float = 0.00001;
region     R = [1..n, 1..n];
         BigR = [0..n+1,0..n+1];
direction  N = [-1, 0];  S = [ 1, 0];
           E = [ 0, 1];  W = [ 0,-1];
var     Temp : [R] float;
           A : [BigR] float;
         err : float;

procedure Jacobi();
     [R] begin
  [BigR]  A := 0.0;
[S of R]  A := 1.0;
          repeat
            Temp := (A@N + A@E + A@S + A@W)/4.0;
            err  := max<< abs(Temp - A);
            A    := Temp;
          until err < eps;

          end;
```

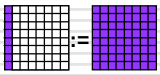


Initialize
Compute new averages
Find the largest error
Update array
… until convergence

## Partial Reduce

- ZPL has ‘full’ reduce: +<<, *<<, max<<, …
- ZPL also has ‘partial’ reduce
  - Applies reduce across rows, down columns,…
  - Requires two regions:
    - One region on statement, as usual
    - One region between operator and operand

```
[1..n,1] B := +<< [1..n,1..n] A;    -- add across rows
[1,1..n] C := min<<[1..n,1..n] A; -- min down columns
```




  - In these examples, result stored in 1st row/col

Collapsed dimensions indicate reduce dimension(s)

## Flood -- Inverse of Partial Reduce

- Reduce "reduces" 1 or more dimensions
- Opposite is flood -- fill 1 or more dimensions

[1..n,1..n] B := >> [1..n, 1] A;

[1..n,1..n] B := >> [1..n, n] A;

- The replication uses multicast, often an efficient operation

## Closer Look At Scaling Each Row

[1..m,1] MaxC := max<<[1..m,1..n] A; *Max of each row*

[1..m,1..n] A := A / >>[1..m,1] MaxC; *Scale each row*

- Flooding distributes values (efficiently) so that the computation is element-wise … lowers communication

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 4 | 2 | | 4 | | 4 | 4 | 4 | 4 |
| 0 | 2 | 3 | 6 | | 6 | | 6 | 6 | 3 | 6 |
| 3 | 3 | 3 | 3 | | 3 | | 3 | 3 | 3 | 3 |
| 8 | 2 | 4 | 0 | | 8 | | 8 | 8 | 8 | 8 |
| | A | | | | MaxC | | >>[1..m,1] MaxC | | | |

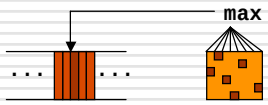
Keep MaxC a 2D array to control allocation

# Flood Regions and Arrays

Flood dimensions recognize that specifying a particular column *over specifies* the situation

Need a *generic* column -- or a column that does not have a specific position … use '*' as value

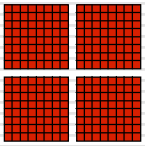
```
region FlCol = [1..m, *];      -- Flood regions
       FlRow = [*, 1..n];
var    MaxC : [FlCol] double; --An m length col
       Row  : [FlRow] double; --An n length row
[1..m,*] MaxC := max<< [1..m,1..n] A; -- Better
```



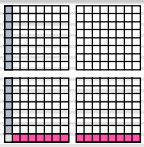

**Think of column in every position**

# Flood *v*. Singleton Difference

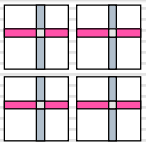
- Lower dimensional arrays can specify a singleton or a flood … it affects allocation



**Region [1..n,1..n] allocated to 4 processors**



**Regions [1..n,1] and [n,1..n] allocated to 4 processors**



**Regions [1..n,*] and [*,1..n] allocated to 4 processors**

# SUMMA Algorithm

For each col-row in the common dimension, flood the item and combine it...

```
var   A:[1..m, 1..n] double;
      B:[1..n, 1..p] double;
      C:[1..m, 1..p] double;
    Col:[ 1..m,*]    double;
    Row:   [*, 1..p] double;
...
[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;
```

Inherit the prevailing dimension

# SUMMA, The First Step

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

C

| | | |
|---|---|---|
| a11b11 | a11b12 | a11b13 |
| a21b11 | a21b12 | a21b13 |
| a31b11 | a31b12 | a31b13 |
| a41b11 | a41b12 | a41b13 |

=

Col

| | | |
|---|---|---|
| a11 | a11 | a11 |
| a21 | a21 | a21 |
| a31 | a31 | a31 |
| a41 | a41 | a41 |

×

Row

| | | |
|---|---|---|
| b11 | b12 | b13 |
| b11 | b12 | b13 |
| b11 | b12 | b13 |
| b11 | b12 | b13 |

**SUMMA is the easiest MM algorithm to program in ZPL**

# SUMMA Algorithm (continued)

For each col-row in the common dimension, flood the item and combine it...

```
[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;
```

--- or, more simply ---

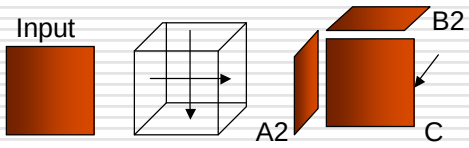
```
           for k := 1 to n do
[1..m,1..p]  C += (>>[ ,k] A)*(>>[k, ] B);
           end;
```

This code is shorter than C's 3-loop
This is SUMMA, fastest || MM alg
New abstractions are key to the win

# Still Another MM Algorithm

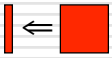
If flooding is so good for columns/rows, why not use it for whole planes?

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

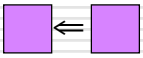


## Recalling Reduce, Scan & Flood

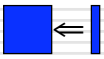
- The operators for reduce, scan and flood are suggestive …
  - Reduce **<<** produces a result of smaller size



  - Scan **||** produces a result of the same size



  - Flood **>>** produces a result of greater size



## Outline

- Context
- Global View Languages
  - NESL
  - ZPL
- ZPL details including SUMMA
- Transparent Performance Model

## There's More

- ZPL has many more important features
  - Not needed here … check out the docs
  - The abstractions are nice, but they are not the reason to be using ZPL … it is the WYSIWYG performance model

## Outline

- Context
- Global View Languages
  - NESL
  - ZPL
- ZPL details including SUMMA
- Transparent Performance Model

## CTA and ZPL

- ZPL was built on the CTA
  - Semantics of operation customized to CTA
  - Compiler targets CTA machines
  - Performance model reflects the costs of CTA
- The benefit of building on the CTA:
  - Programming constraints are realistic, scalable
  - Programs are portable *with performance*
  - Programmers can reliably estimate performance and observe it (or better) on every platform

Building on CTA is a main contribution of ZPL

## Knowing Performance of Programs

- Recall that in the sequential case, writing in a performance-sensitive language (C), the RAM model describes how the program will run
- Writing in ZPL, the CTA model describes how the program will run
  - Programmer needs to know the CTA
  - Language constructs' performance must be described in CTA terms
  - Information must "compose"
- This is called ZPL's "What you see is what you get" (WYSIWYG) Performance Model"

## Goal For Nex Few Minutes

- Let's explain the WYSIWYG Model
  - What is it? It is a way to look at your code and see how fast it will run -- like algorithm analysis
  - What other languages have it? None.
  - How does it work?
    - We explain how ZPL arrays map on to CTA model
    - We explain how ZPL's operators -- scan and flood etc -- run on the CTA with that array allocation
  - Can I see it in action?
    - Yes, we'll give examples at the end

**This is the most important idea in today's lecture**

## Assumes Many Pts Per Processor

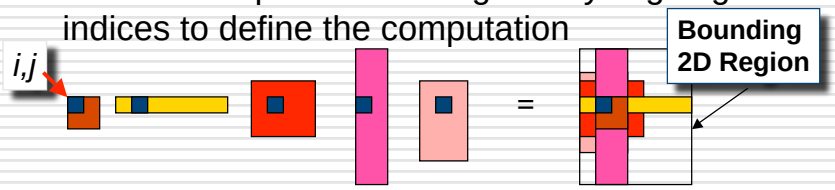
ZPL allocates regions (and therefore arrays) to processors so that many contiguous elements are assigned to each processor to exploit locality

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

**Let's walk-through the process**

# Union The Regions Together

Create the "footprint" of the regions by aligning indices to define the computation




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*

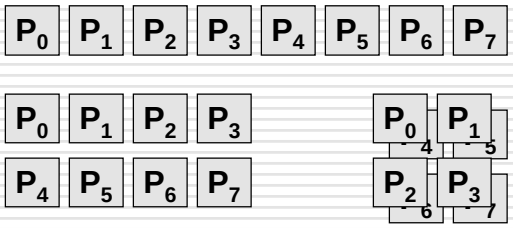
**The bounding region is allocated to processors**

# Get Processor Num + Arrangement

The number and arrangement of processors 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:

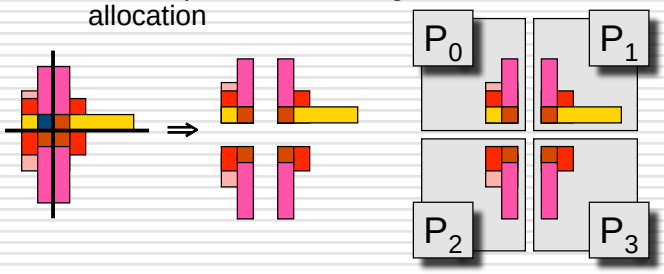
**The CTA does not favor any arrangement, so use a generic one**



# Allocate Bounding Region to Grid

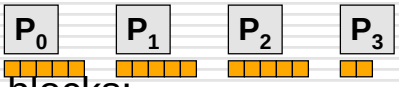
The bounding region is allocated to processor grid in the "most balanced" way possible

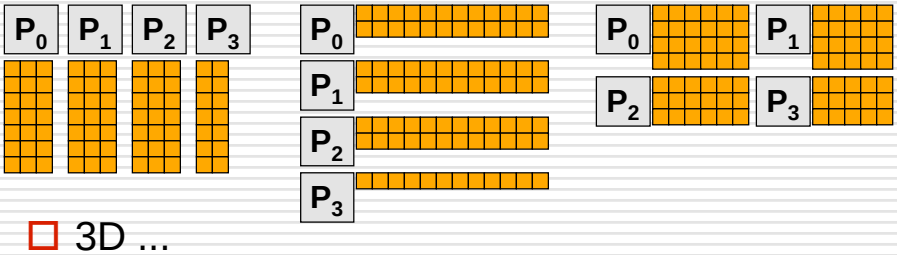
- Regions inherit their position from their position in the bounding region
- Array elements inherit their positions from their index's position in the region, and hence their allocation




# More Typical Allocations

- 1D is segmented;
- 2D is panels, strips or blocks;



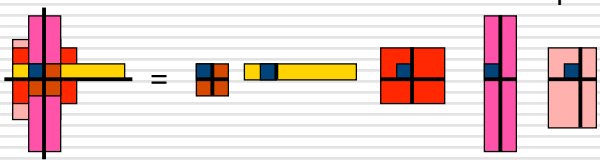




- 3D ...

# 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 the 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

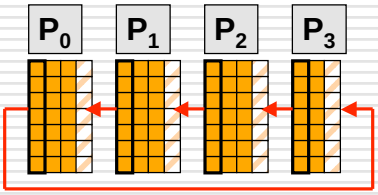
- Performance is given in terms of the CTA
- Performance is relative, e.g. *x* is more expensive in communication than *y*

- Rules…
  - **`A + B`** -- Element-wise array operations
    - No communication
    - Per processor work is comparable to C
    - Work fully parallelizable, i.e. time = work/P

# Rules Of Operation (continued)

`B+A@^east` -- @ references including @^

Arrays allocated with “fluff” for every direction used




- Nearest neighbor point-to-point communication of edge elements, i.e. small communication, little congestion
- Edge communication benefits from surface-to-volume advantage: an *n* increase in elements, adds √*n* comm load
- Local data motion, possibly

# << || >>

`+<<A` -- Reduce

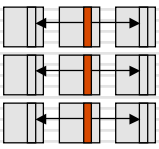
- Accumulate local elements
- O(log P) tree accumulation, or better
- Broadcast, which is worst case O(log P), but usu. less

`+||A` -- Scan

- Accumulate local elements
- Ladner/Fischer O(log P) tree parallel prefix logic
- Update of local elements

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

- Multicast array segments, O(log P) w.c.
- Represent data “non-redundantly”



## Rules of Operation (continued)

**`A#[I1, I2]`** -- Remap, both gather and scatter

- (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
- Heavily optimized, esp. to save first all-to-all

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

**ZPL is only parallel language with performance model**

## Applying WYSIWYG In Real Life...

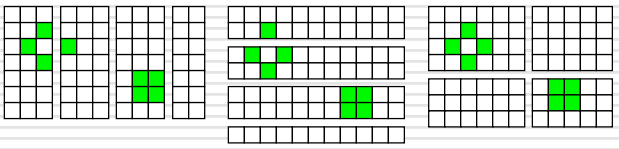
```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];
        BigR = [0..n+1,0..n+1];
direction  N = [-1, 0]; NE = [-1, 1];
           E = [ 0, 1]; SE = [ 1, 1];
           S = [ 1, 0]; SW = [ 1,-1];
           W = [ 0,-1]; NW = [-1,-1];
var NN : [R] ubyte; TW : [BigR] boolean;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

Code for performance costs implied by WYSIWYG

## Analyzing Life By Color

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

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



**Delay is cλ**

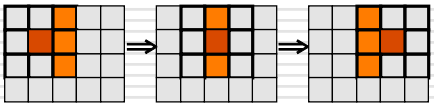
## Analyzing By Color (continued)

- Purple: Element-wise computation with @ operations … expect λ delay for @ (all at once if synch'ed) and then full parallel speed-up for local operations
- Red: Reduce uses Ladner/Fischer parallel prefix, with local combining and log(P) tree to communicate … potentially the most complex operation in Life

**Knowing the relative costs of the program allows us to optimize it for some purpose … count generations**

# Optimizations Can Help

- WYSIWYG is the worst case … optimizations are possible …
- Sequential Optimizations e.g. stencil opts



Sum of orange items performed once

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

- Parallel Optimizations e.g. communication motion -- prefetching to overlap communication with computation

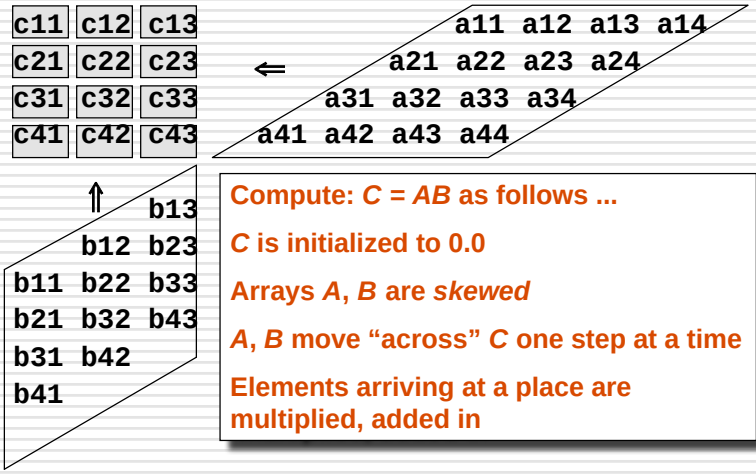
# Applying WYSIWYG in Alg Design

WYSIWYG, a key tool for parallel algorithm design … work through the logic of balancing costs

- There are dozens (hundreds?) of matrix product algorithms … which do you want?

  MM is a common building block, so someone else should have done this (vdG&W did!), but we use it as an example of process
- Two popular choices are
  - Cannon's algorithm
  - SUMMA (vdG&W)
- Which is better as a ZPL program, i.e. better for scalable parallel machines, clusters, CTA model

# Cannon's Algorithm, A Classic




**Compute: *C* = *AB* as follows ...**

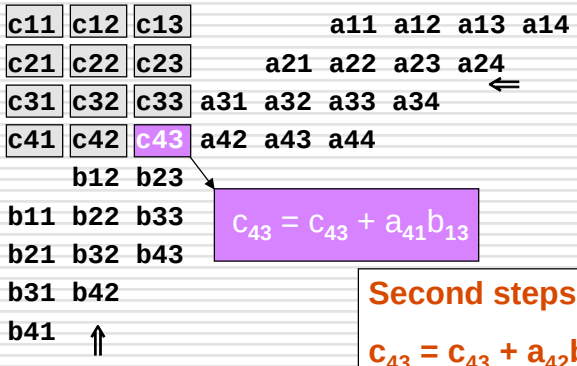
***C* is initialized to 0.0**

**Arrays *A*, *B* are *skewed***

***A*, *B* move "across" *C* one step at a time**

**Elements arriving at a place are multiplied, added in**

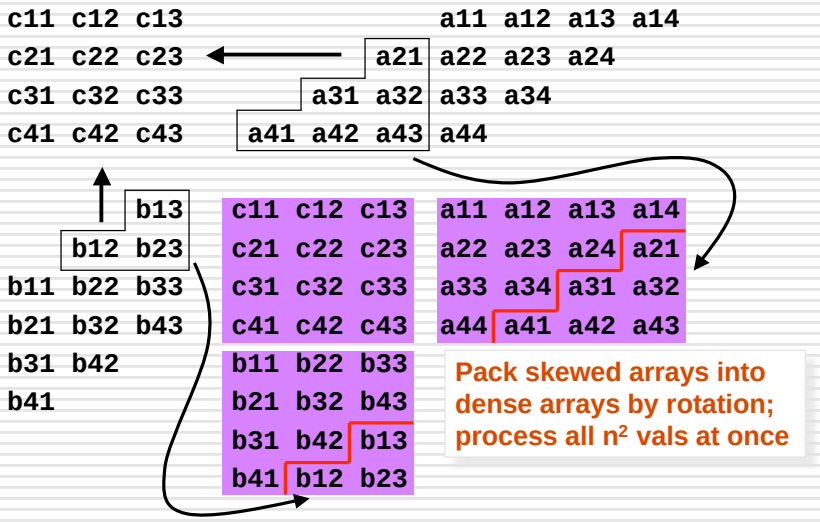
# Motion of Cannon's, First Step




**Second steps ...**

$c_{43} = c_{43} + a_{42}b_{23}$
$c_{33} = c_{33} + a_{31}b_{13}$
$c_{42} = c_{42} + a_{41}b_{12}$

# Programming Cannon's In ZPL




# Four Steps of Skewing A

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

*Shift last m-i rows left*

| | |
|---|---|
| a11 a12 a13 a14<br>a21 a22 a23 a24<br>a31 a32 a33 a34<br>a41 a42 a43 a44 | a11 a12 a13 a14<br>a22 a23 a24 a21<br>a32 a33 a34 a31<br>a42 a43 a44 a41 |
| Initial | i = 2 step |
| a11 a12 a13 a14<br>a22 a23 a24 a21<br>a33 a34 a31 a32<br>a43 a44 a41 a42 | a11 a12 a13 a14<br>a22 a23 a24 a21<br>a33 a34 a31 a32<br>a44 a41 a42 a43 |
| i = 3 step | i = 4 step |

**… And Skew B vertically**

## Cannon's Declarations

For completeness, if 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;
```

## Cannon's Algorithm

**Skew A, Skew B, {Multiply, Accumulate, Rotate}$^n$**

```
      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;
```

## SUMMA Algorithm To Compare To

```
   var   Col : [1..m,*]    double;   Col flood array
         Row : [*,1..p]    double;   Row flood array
           A : [1..m,1..n] double;
           B : [1..n,1..p] double;
           C : [1..m,1..p] double;
                  ...
[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;
```

## Compare Cannon's & SUMMA MM

- Analyze the choices with WYSIWYG …
  - SUMMA has shortest code [so what?]
  - Cannon's uses only local communication
- The two algorithms abstractly:

```
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
```

# Compare Cannon's & 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

| Cannon's | SUMMA |
|---|---|
| ~~Declare~~ | ~~Declare~~ |
| Skew A | ~~Initialize~~ |
| Skew B | ~~loop til n~~ |
| ~~Initialize~~ | Flood A |
| ~~loop til n~~ | Flood B |
| ~~C+=A*B~~ | ~~C+=A*B~~ |
| Rotate A,B | |

# 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 (likely) more expensive than λ time, but less that λ(log P) ... probably much less, and there are fewer of them

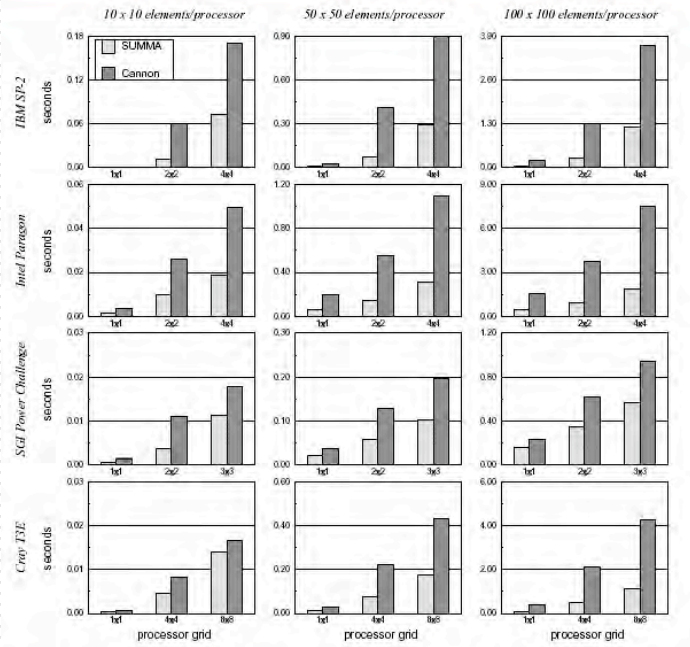
```
[1..m,1..p]    C := 0.0;              Initialize C
          for k := 1 to n do
   [1..m,*]  Col := >>[ ,k] A;        Flood kth col of A
   [*,1..p]  Row := >>[k, ] B;        Flood kth row of B
[1..m,1..p]    C += Col*Row;          Combine elements
          end;
```

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

# Bottom Line ...

- We assert that SUMMA is the better algorithm
  - Though it does "potentially more expensive" communication, it does less of it
  - It's "nonredundant" flood arrays cache well
  - There is less data motion
- Analytically ...

| algorithm | number of communications | communication complexity | communication volume | flops | elements referenced |
|---|---|---|---|---|---|
| *Cannon* | $4n$ | $1$ | $n$ | $2n^3 - n^2$ | $n \cdot (2\frac{n^2}{p} + 3n^2)$ |
| *SUMMA* | $2n$ | $\log p$ | $n$ | $2n^3$ | $n \cdot (n^2 + 2n)$ |




ZPL's WYSIWYG Performance Model Chamberlain, Choi, Lewis, Lin, Snyder, Weathersby IEEE *HIPS-98*, 1998

# Guarantees

ZPL uses a different approach to performance than other parallel languages

- *Historically, performance came from compiler optimizations that might/might not fire …*
- WYSIWYG guarantees (it's a contract) that ZPL programs will work a certain way …
  - It may be better … WYSIWYG is a worst case that often doesn't materialize
  - Aggressive optimizations help a lot

**If there are any surprises, they'll be pleasant**

## 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

## Bottom Line for Learning About ZPL

- The reason we're learning ZPL is because it illustrates how a parallel programming language can give access to the CTA machine model, allowing programmers to write intelligent parallel programs
- You want your programming language to have that property, too!
- If it doesn't, dump it and use a library that lets you apply the CTA model yourself

## The Future: Transparent Parallelism

- Higher level abstractions --
  - Write less, but get more done

    **Notice: This will require actual research**

  - Rely on compilers to generate parallel code
- Expose the machine's behavior by implicit means
  - Sound foundations for abstractions & compiler
  - Visible performance model: WYSIWYG

## What Can Microsoft Do?

- "Parallelism requires adjustments at every level of the stack … the repartitioning of different tasks to different layers … So look for a rebalancing of roles and runtimes. We need to formalize that in the operating system. Expect the first pieces in the next generation of Windows." Craig Mundy, Microsoft Chief Research & Strategy Officer, 2008/10/3