ZPL, NESL: High Level Languages

Producing languages with better abstractions is possible …
why are there not more such languages?

Today's Goal

- We have acknowledged that present day || programming facilities are inadequate … a good question would be, what do we want?
- Today we give two languages that exhibit a high level of parallel abstraction
  - ZPL from UW
  - Nesl from CMU
- The languages have weaknesses, but mostly we're focused on their strengths as exemplars

ZPL

- ZPL, a research parallel language w/ 3 goals
  - Performance == as good as platform-specific custom code
  - Portability == runs well on all platforms
  - Convenience == clean, easy-to-understand programs; no parallel grunge
- Developed at UW by 6 really smart grad students: Brad Chamberlain, Sung-Eun Choi, Steve Deitz, E Chris Lewis, Calvin Lin, Derrick Weathersby

The Team

Secosky
Lewis
Snyder
Weathersby
Ngo
Watson
Deitz
Choi
Chamberlain
Lin
ional Symposium on
R ARCHITECTURE

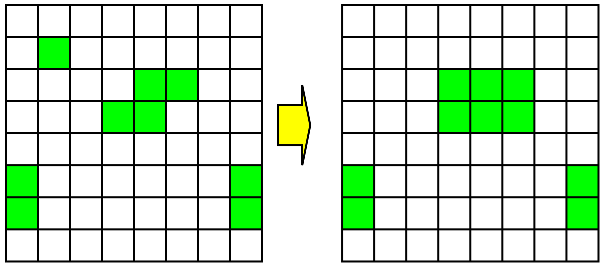
ZPL Is Important To Us

- ZPL is our representative 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"

Conway's Game of Life

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

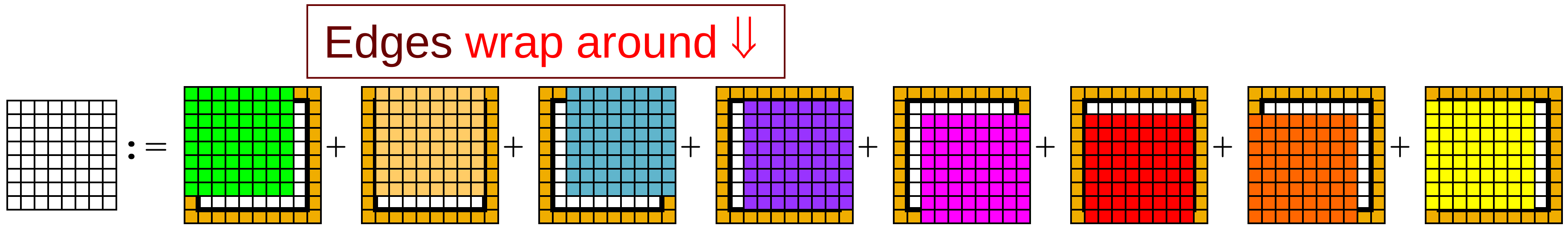


- Organism in next generation if position is alive in this generation and has 2 neighbors, or in this generation it has 3 neighbors
- Or: (thisGen && neighbors== 2) || (neighbors==3)

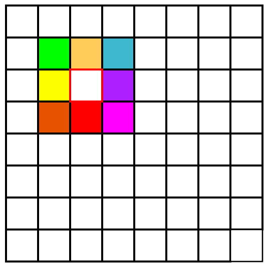
See Life As An Array Computation

Compute Over Whole Arrays

- Count neighbors by adding organisms (bits)



- Consider the sum of these arrays from a single cell's perspective …



Express Array Computation in ZPL

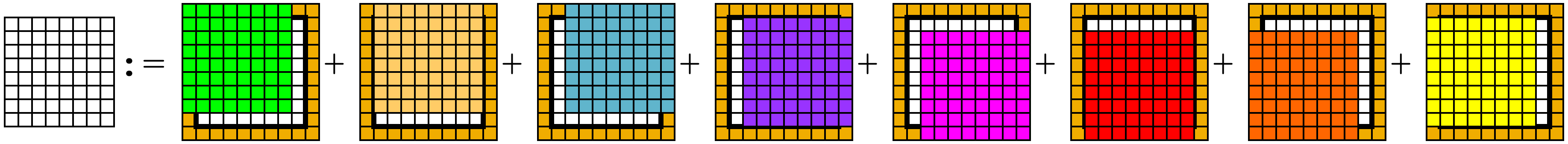
Conway's Life: The World is bits

```
[R] 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

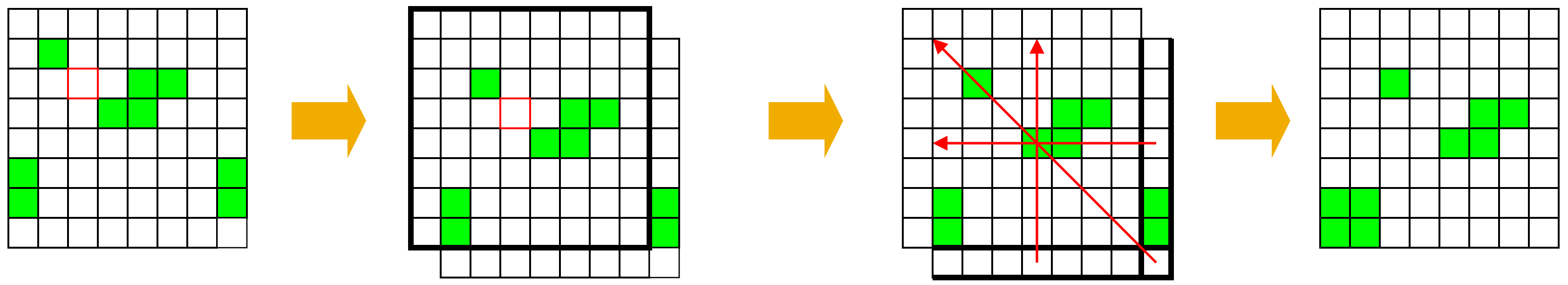
Apply rules to live by

"Or" bits in world to see if any alive



A Closer Look At TW@^nw

- The torus world says that each direction wraps – expressed as @^



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

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;
```

Life In ZPL -- The Detail

```
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;
```

Topics
"Typical" Form
Regions
Directions
Config Vars
Reduce

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

Using Regions to Declare Arrays

- Regions are used to declare arrays … it's like adding data to the indices
- Capitals are used by convention to distinguish arrays from scalars
- Named or literal regions are OK

```
var A, B, C : [R] double;
var Seq : [V] boolean;
var Huge : [0..2^n, -5..5] float;
```

- Regions are used once; no array has more than one region component
- Regions are a source of parallelism…

Regions Control Computation

- Statements containing arrays need a region to specify which items participate

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

- 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

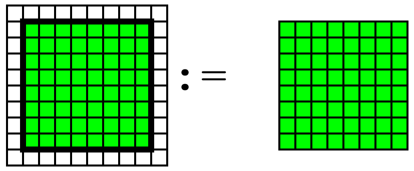
Parallelism In Statement Evaluation

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

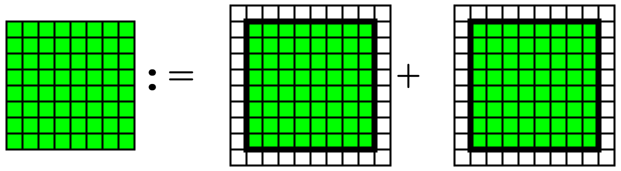
  var A, B : [1..n,1..n] float; C : [2..n-1,2..n-1] float;

- Then

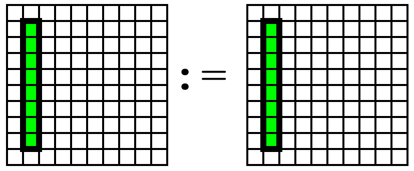
  [2..n-1,2..n-1] A := C;



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



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

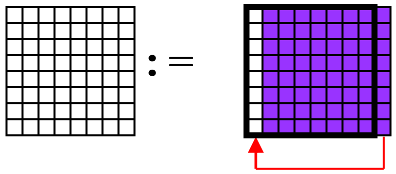


@ Uses Regions & Directions

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

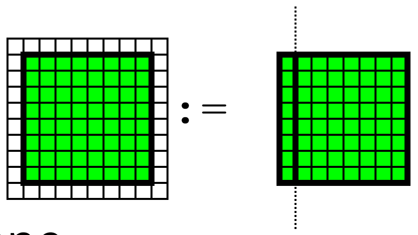
- Let

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

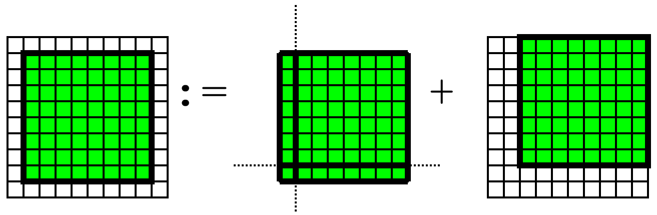
  direction east = [0,1]; ne = [-1,1];

- Then

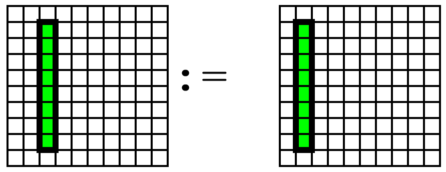
  [2..n-1,2..n-1] A := C@^east;



  [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;
```

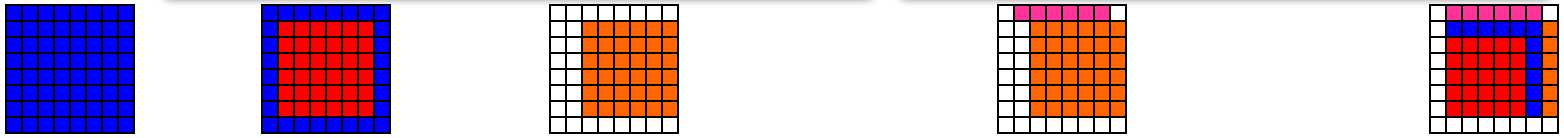
- All elements participate; order of evaluation is unspecified … caution floating point users
- ZPL also has partial reductions, scans, partial scans, and user defined reductions and scans

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;
```

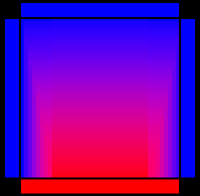
```
direction e = [ 0,1];
          n = [-1,0];
         ne = [-1,1];
```



`[R]X:=`🟦 `[C]X:=`🟥 `[C at e]Y:=`🟧 `[n of C]Y:=`🟪 `[C]Y:=X@ne`

execution →

Applying Ideas: Jacobi Iterat



- 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

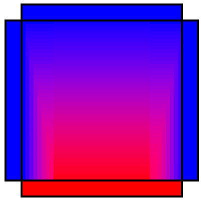
Initialize
Compute new averages
Find the largest error
Update array
… until convergence

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

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

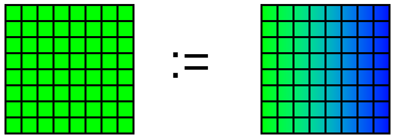
    [1,1..n] C := min<<[1..n,1..n] A; -- min down columns
  - In these examples, result stored in 1st row/col

Collapsed dimensions indicate reduce dimension(s)

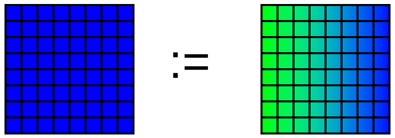
Flood -- Inverse of Partial Reduce

- 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: 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			

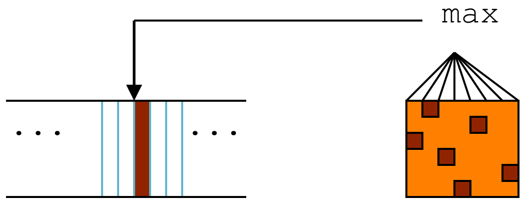
Keep MaxC a 2D array to control allocation

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 arrays (continued)

Since flood arrays have some unspecified dimensions, they can be “promoted” in those dimensions, i.e logically replicated

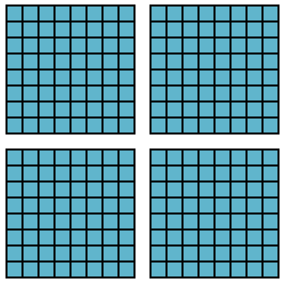
- Scaling a value by max of row w/o flooding:

```
   [1..m,*]    MaxC := max<< [1..m,1..n] A;
[1..m,1..n]       A := A / MaxC;     --Scale A;
```

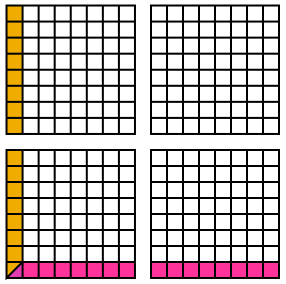
The promotion of flooded arrays is only logical

Flood v. Singleton Difference

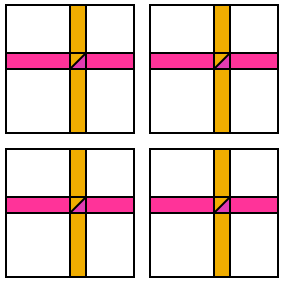
- Lower dimensional arrays can specify a singleton or a flood … it affects allocation



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
```

```
Col               Row                C
 a11 a11 a11       b11 b12 b13        a11b11 a11b12 a11b13
 a21 a21 a21   ×   b11 b12 b13   =    a21b11 a21b12 a21b13
 a31 a31 a31       b11 b12 b13        a31b11 a31b12 a31b13
 a41 a41 a41       b11 b12 b13        a41b11 a41b12 a41b13
```

SUMMA is the easiest MM algorithm to program in ZPL

van de Geijn & Watts say it's the fastest machine independent

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;
```

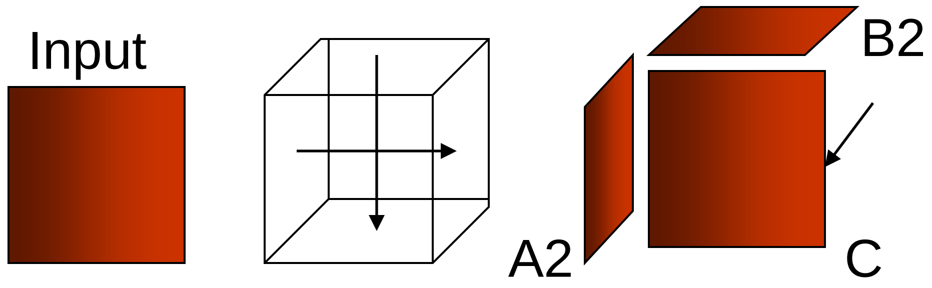
--- or, more simply ---

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

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);
```



Partial Scan

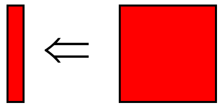
- Partial scans are possible too, but unlike reduce they do not reduce dimensionality, so the compiler cannot tell which dimension to reduce … so specify

`+||[2]A` is a partial scan in the 2nd dimension

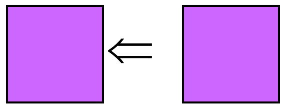
```
                1   1   1   1        1    2    3    4
+||[2]          1   1   1   1   ⇒    1    2    3    4
                1   1   1   1        1    2    3    4
                1   1   1   1        1    2    3    4
```

Recalling Reduce, Scan & Flood

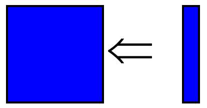
- The operators for reduce, scan and flood are suggestive …
  - Reduce **<<** produces a result of smaller size



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



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



Index1 …

- ZPL comes with “constant arrays” of any size
- Index*i* means indices of the ith dimension

```
[1..n,1..n]begin
             Z := Index1; -- fill with first index
             P := Index2; -- fill with second index
             L := Z=P;    -- define identity array
           end;
```

Index*i* arrays: compiler created using no space

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Index1

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

Index2

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

L

Remap

The remap operator (#) implements general data motion, including rank change

- Two cases:

  Gather, A := B#[C1,C2];

  Scatter, A#[C1,C2] := B;
- For *r* rank array, provide *r* rank *r* arrays giving indices to be referenced
- Transpose: AT[i,j] = A[j,i]

```
[R] AT := A#[Index2,Index1]; -- Standard Idiom for transpose
```

Remap (Gather)

The index array in the ith position gives the indices for the ith dimension

Gather: For a position, where does value come from

```
a c e b d f ⇔ a b c d e f#[1 3 5 2 4 6]
```

$$
\begin{matrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ \boxed{d} & h & l & p \end{matrix}
:=
\begin{matrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{matrix}
\#
\left[\begin{matrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ \boxed{1} & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{matrix}
,
\begin{matrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ \boxed{3} & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{matrix}\right]
$$

AT A Index2 Index1

Remap (Scatter)

- Scatter Remap has potential problem in that values can map to the same place … order is unspecified … use +=, etc. if not unique

Scatter: For a value, where does it go?

```
a d b e c f #[1 3 5 2 4 6] ⇔ a b c d e f
```

$$
\underset{\text{AT}}{\begin{matrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{matrix}}
\quad \#\left[\underset{\text{Index2}}{\begin{matrix} 1 & 2 & \bigcirc\!\!\!\!3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{matrix}} \,,\, \underset{\text{Index1}}{\begin{matrix} 1 & 1 & \bigcirc\!\!\!\!1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{matrix}}\right]
:= \underset{\text{A}}{\begin{matrix} a & b & c & \bigcirc\!\!\!\!d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{matrix}}
$$

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 key contribution of ZPL

Heads Up

- We now explain ZPL's performance model
- What is it?
    - It is the way programmers know how fast (or slow) the statements of their programs will run"
- We all "know" the performance model for C
- Every || language should have a performance model
- Learning this idea is why we've learned ZPL

Knowing Performance of Programs

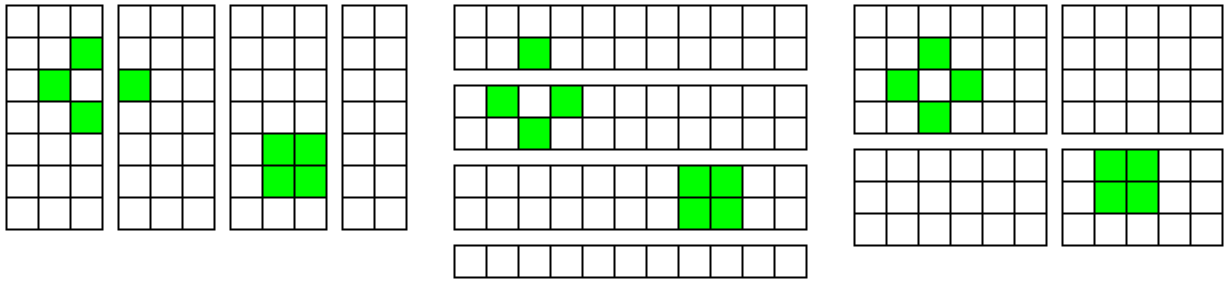
- How does it work?
  - First, the language designers, knowing the CTA, formulate operations compatible with it
  - The compiler “targets” the CTA
  - The performance of the language features is expressed in terms of CTA concepts and “given” to the programmers
  - In ZPL’s case the performance is “syntactically visible” and called the WYSIWYG performance model

Performance Model (WYSIWYG)

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

- Performance is given in terms of the CTA (and RAM)
- Performance is *relative*, e.g. *x* is more expensive in communication than *y* -- absolute not possible

- To start allocate work (owner computes) and data:



P=4 allocations for 2D arrays: columns, rows, blocks

WYSIWYG Performance Model

Describe the costs for all language constructs

- Declarations, control flow have negligible cost
- Scalar computations are redundant, also “free”
- Array operations costs depend on operators:

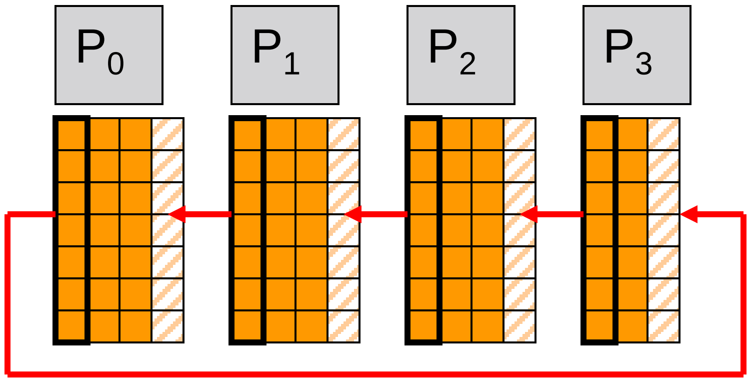
- Rules…

  `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)

`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 $\sqrt{n}$ comm load
- Local data motion, possibly

<< || >>

`+<<A` -- Reduce

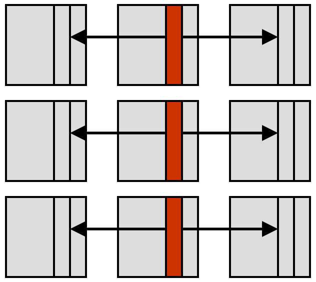
- Accumulate local elements
- O(log P) tree accumulation, or better
- Broadcast, which is worst case O(log P), but usu. less

`+||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

More on Array Allocation

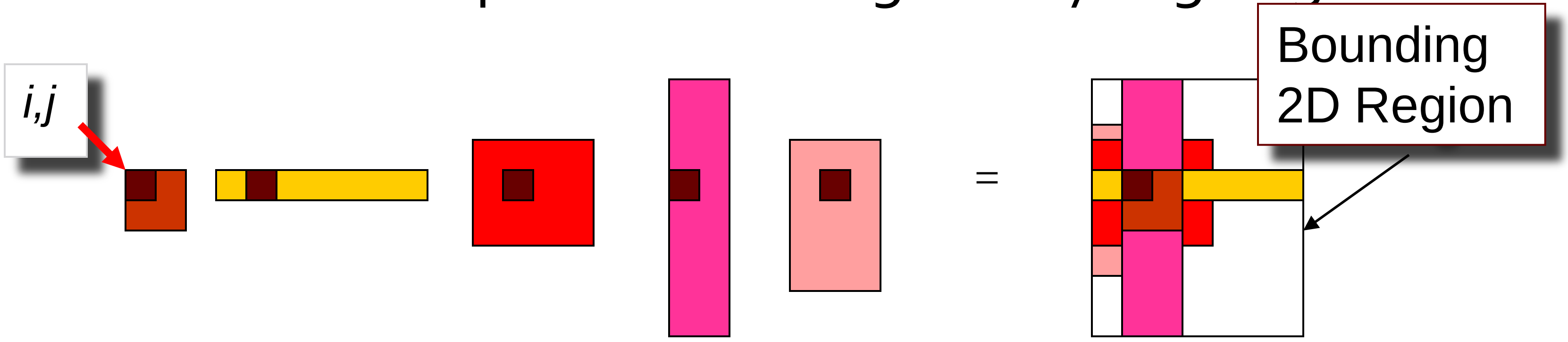
ZPL allocates regions (and therefore arrays) to processors so many contiguous elements are assigned to each to exploit locality

- 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



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*

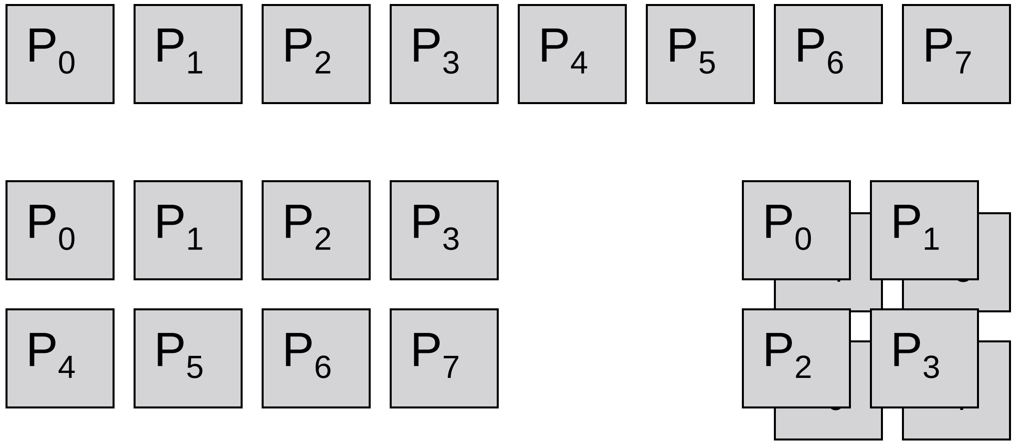
The bounding region is allocated to processors

Get Processor Num + Arrangement

The number and arrangement of processors is given by the programmer on the command line [or programmed; more later]

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

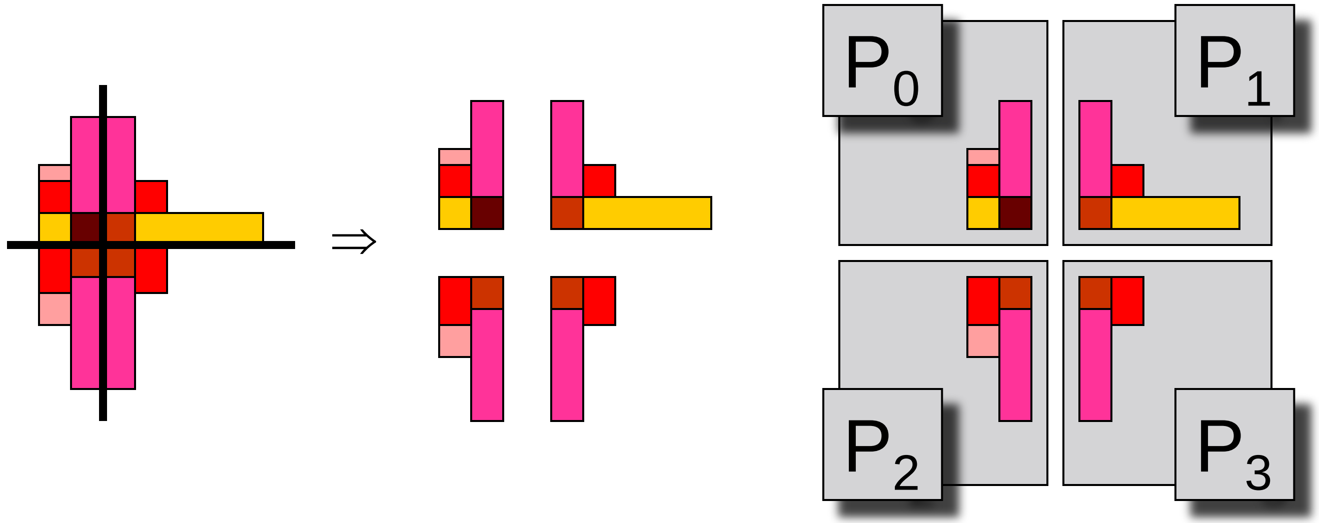
The CTA does not favor any arrangement, so use a generic one



Allocate Bounding Region to Grid

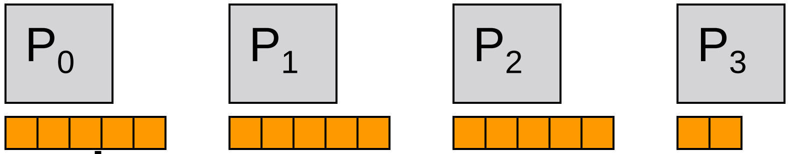
The bounding region is allocated to processor grid in the "most balanced" way possible

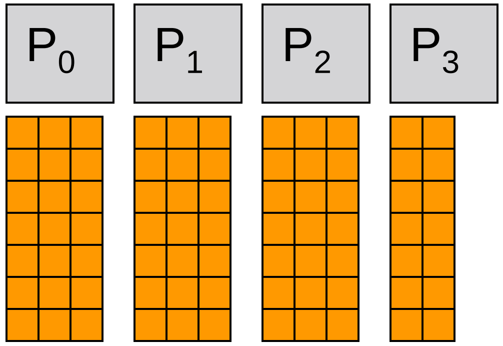
- Regions inherit their position from their position in the bounding region
- Array elements inherit their positions from their index's position in the region, and hence their allocation

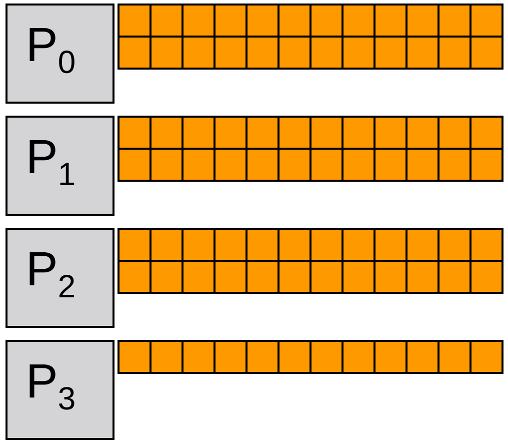


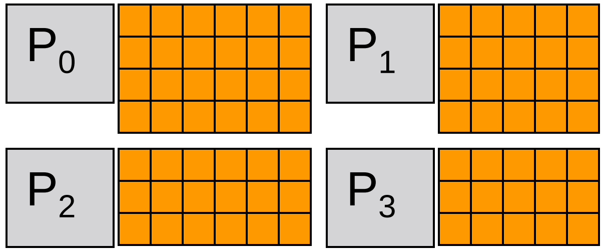
More Typical Allocations

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









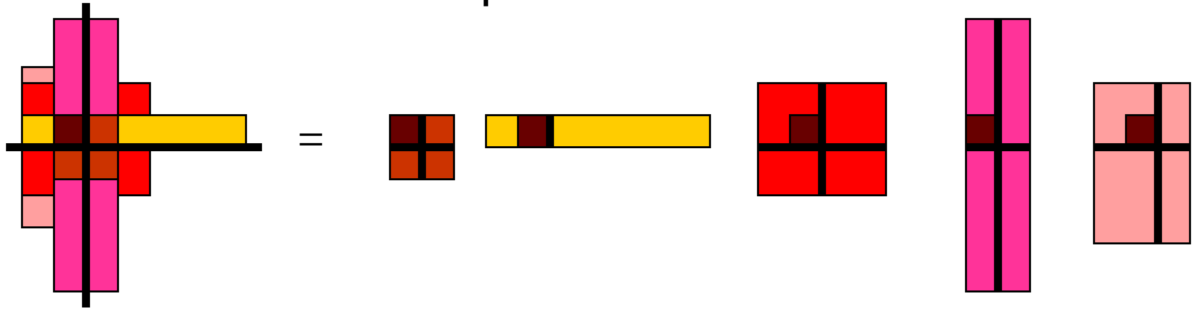
- 3D …

ZPL uses Ceiling/Floor and includes fluff

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 …`**

Applying WYSIWYG In Real Life…

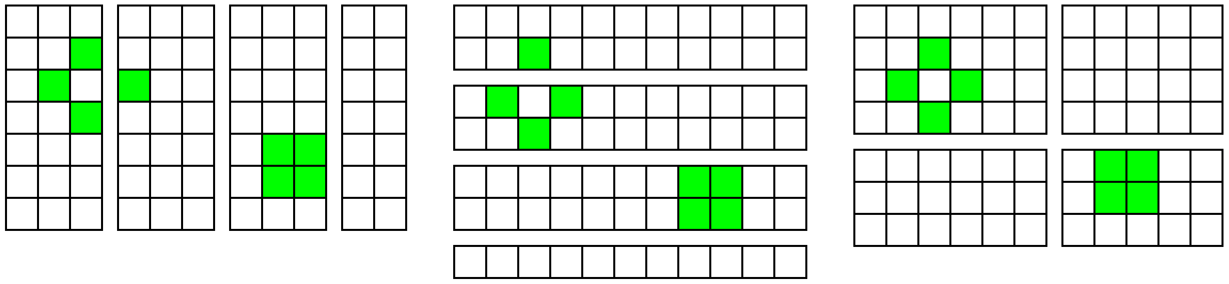
```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];
        BigR = [0..n+1,0..n+1];
direction  N = [-1, 0]; NE = [-1, 1];
           E = [ 0, 1]; SE = [ 1, 1];
           S = [ 1, 0]; SW = [ 1,-1];
           W = [ 0,-1]; NW = [-1,-1];
var NN : [R] ubyte; TW : [BigR] boolean;
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

How Many Generations?

- Compute count of generations before life dies out

```
program Life;
config var n : integer = 512;
region      R = [1..n, 1..n];
direction NW = [-1,-1]; N = [-1, 0]; NE = [-1, 1];
           W = [ 0,-1];                 E = [ 0, 1];
          SW = [ 1,-1]; S = [ 1, 0]; SE = [ 1, 1];
var NN:[R] ubyte; TW:[R] boolean; count:integer = 0;
procedure Life();
      [R] begin read(TW); -- Input
           repeat
             count += 1;
             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;
           writeln(count, " generations");
          end;
```

Add a counter to previous program

How Many Generations?

Testing on each generation my be excessive -- analyze

```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];
direction NW = [-1,-1]; N = [-1, 0]; NE = [-1, 1];
           W = [ 0,-1];               E = [ 0, 1];
          SW = [ 1,-1]; S = [ 1, 0]; SE = [ 1, 1];
var NN:[R] ubyte; TW:[R] boolean; count:integer = 0;
procedure Life();
      [R] begin read(TW); -- Input
           repeat
             count += 1;
             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;
           writeln(count, " generations");
          end;
```

Optimize To Reduce End Tests

```
config var n : integer = 512; epoch : integer = 50;
...
var NN:[R] ubyte; TW,TWo:[R] boolean; count:integer = 0;
procedure Live(integer:gens);
   begin var i : integer;
         for i := 1 to gens do
             NN := (TW@^N + TW@^NE + TW@^E + TW@^SE
                  + TW@^S + TW@^SW + TW@^W + TW@^NW);
             TW := (NN=2 & TW) | NN=3;
         end;
   end;
procedure Life();
         [R] begin read(TW);
              while |<<TW do
                TWo:=TW; Live(epoch); count += epoch;
              end;
              count -= epoch; TW := TWo; -- Roll back
              repeat
                Live(1); count += 1;
              until ! |<<TW;
              writeln(count, " generations");
             end;
```

Analyze Costs

Do Epochs

Recover State

Redo World End

Report

Applying WYSIWYG in Alg Design

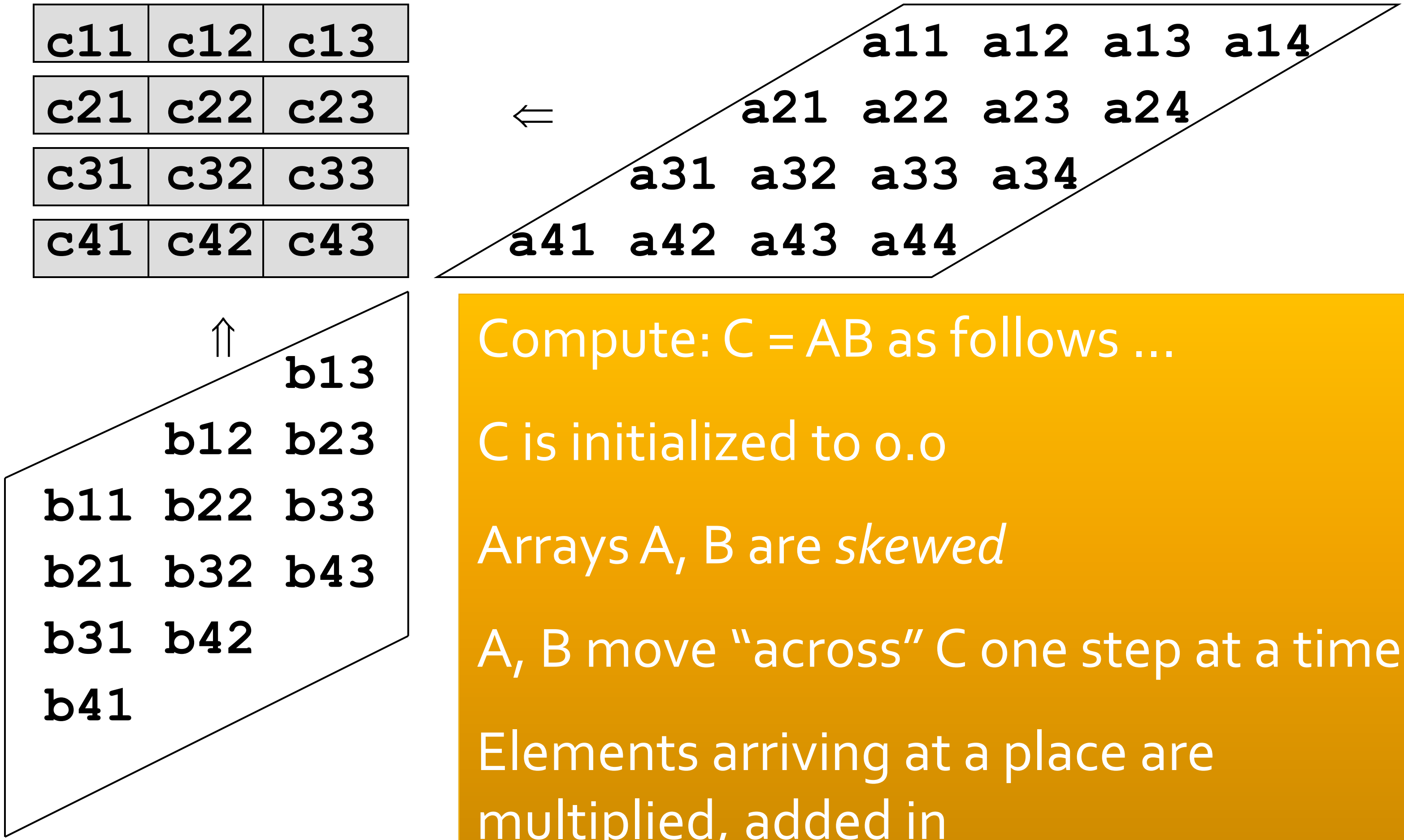
WYSIWYG, a key tool for parallel algorithm design … work through the logic of balancing costs

- 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
- Which is better as a ZPL program, i.e. better for the CTA model

Cannon's Algorithm: MM in Motion



Compute: C = AB as follows …

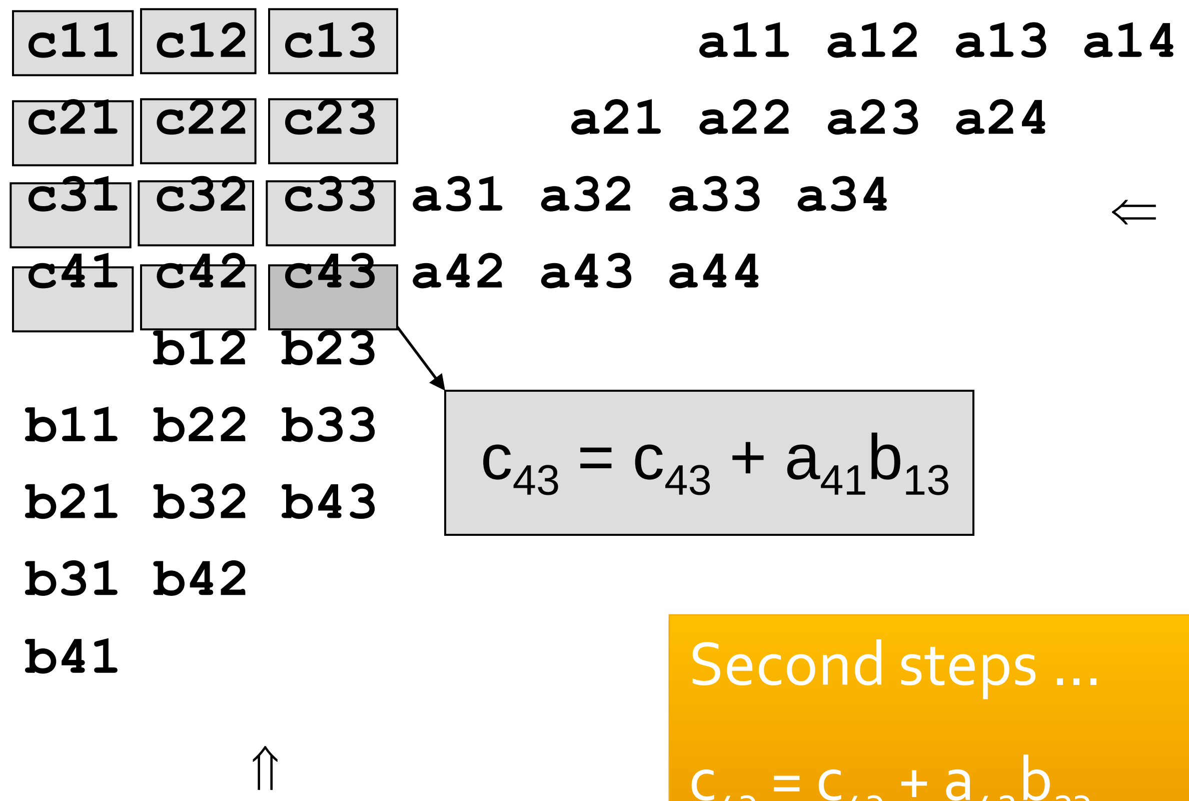
C is initialized to 0.0

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 Algorithm Step1



$$c_{43} = c_{43} + a_{41}b_{13}$$

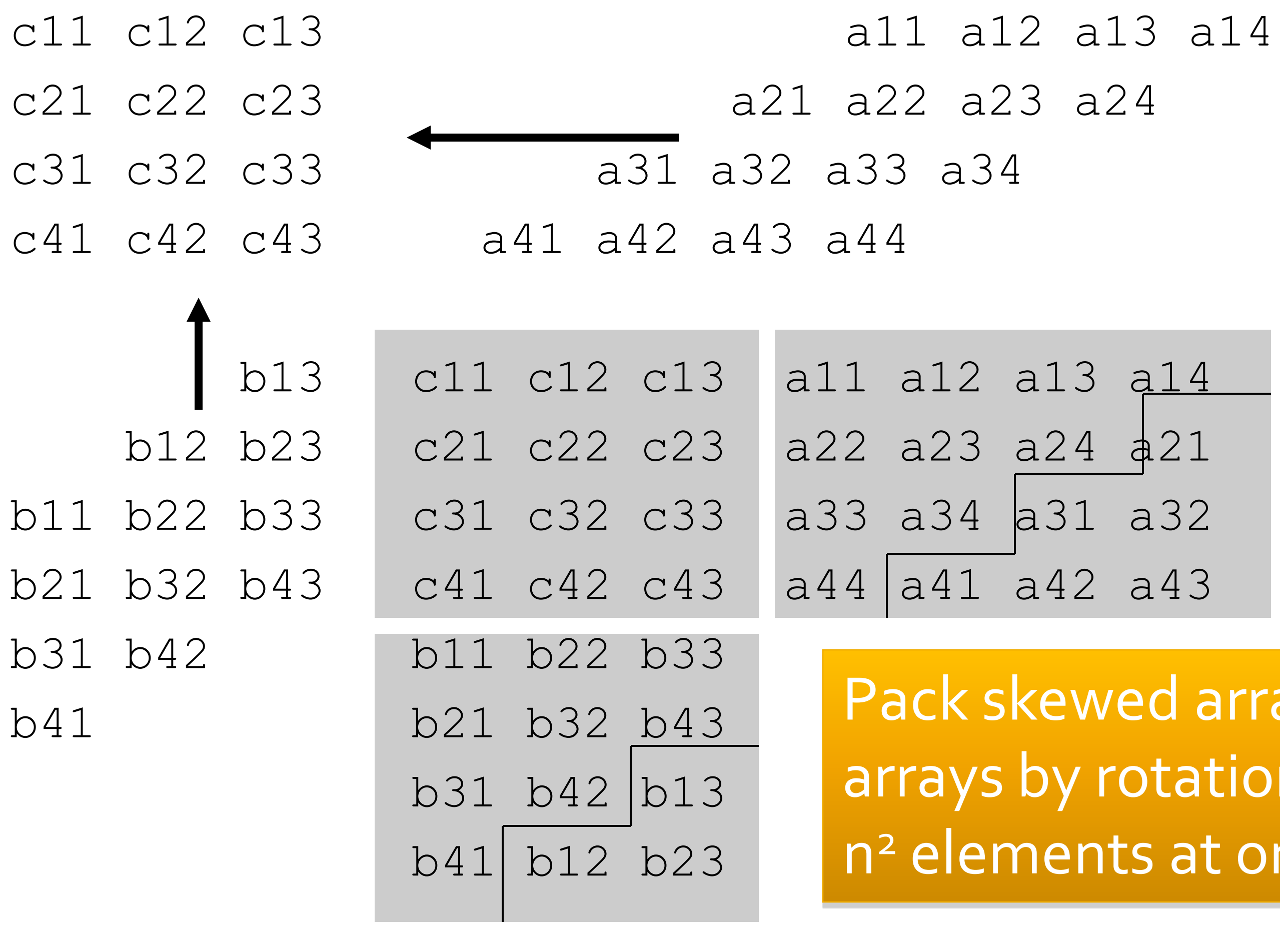
Second steps …

$c_{43} = c_{43} + a_{42}b_{23}$

$c_{33} = c_{33} + a_{31}b_{13}$

$c_{42} = c_{42} + a_{41}b_{12}$

Programming Cannon's In ZPL



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

Cannon's Declarations

For completeness, when A is m×n, B is n×p, and 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, Accum, 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;
```

SUMMA Algorithm in ZPL

```
    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;
```

Comparing 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	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	

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

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, Accum, 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

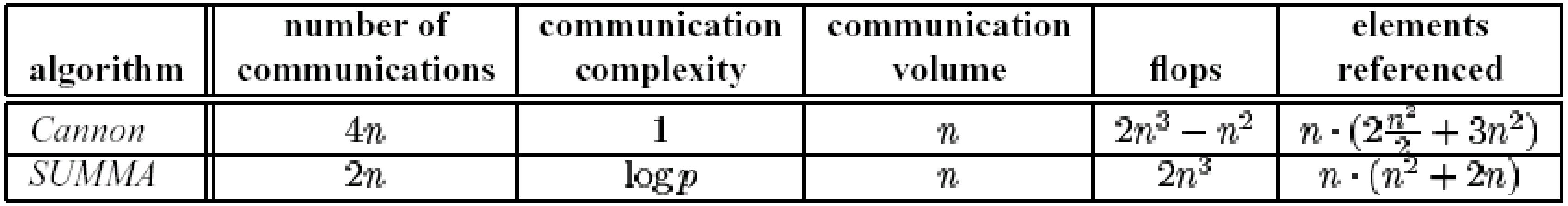
The flood is (likely) more expensive than λ time, but less that λ(log P) … probably much less, and there are fewer of them

```
[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 …

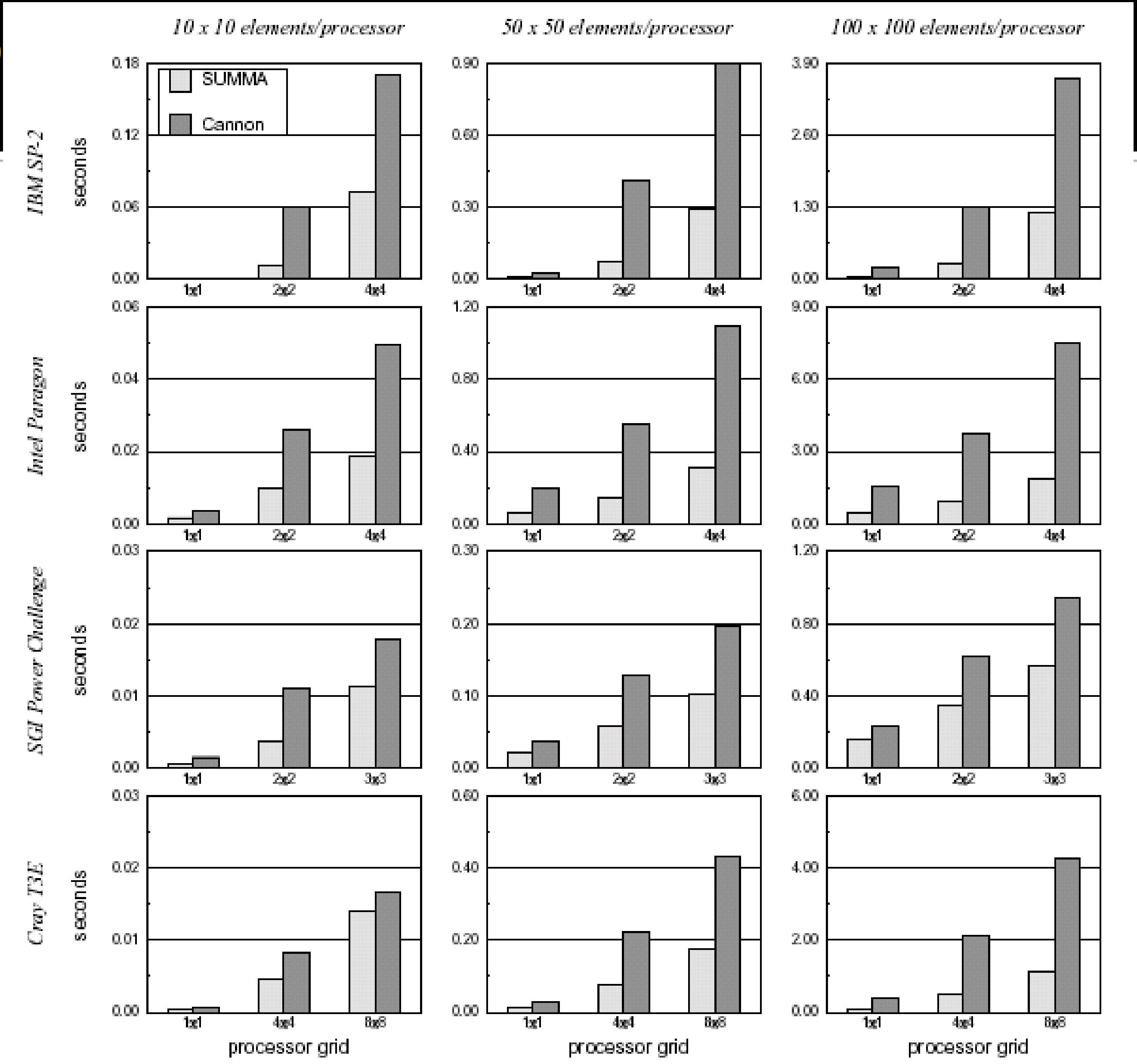
- We assert that SUMMA is the better algorithm
  - Though it does "potentially more expensive" communication, it does less of it
  - It's "nonredundant" flood arrays cache well
  - There is less data motion
- Analytically …



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

Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. **ZPL's WYSIWYG performance model.** In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.

MM No.s



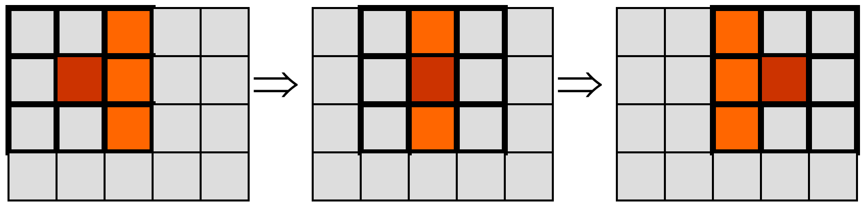
Y-axis is time …

▢ SUMMA

■ Cannon

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

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 ZPL In 524

- The reason we learned ZPL was because it illustrates how a high level parallel language can give access to the CTA machine model, allowing programmers to write intelligent parallel programs easily and portably
- 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

Global View Language

- ZPL Classic (the portion we learned) is a global view language, meaning it's
  - *P-independent*, all executions of the program produce the same result regardless of the number or arrangement of the processors
  - Functional languages tend to be P-independent
  - P-independent is a very desirable property from a programmers view
- Another is NESL

NESL

- NESL was developed by Guy Belloch at 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]}` producing `[3 14 1 5 9]`
  `{a+b: a in [1 2 3]; b in [8 7 6]}` producing `[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 as they are not conditioned on P, λ, locality, etc.

Summary

- Parallel programming will be convenient and non-disruptive when languages provide the kinds of abstractions programmers need
- ZPL abstracts above the HW, but not so far that we loose track of the underlying (logical) machine
  - ZPL achieves performance-portability
  - ZPL "works" because it has a built-in performance model: WYSIWYG
  - You use a performance model – it might as well be one that the compiler-writers target

Homework 7

- Write a ZPL program to solve the Red/Blue simulation using a new termination criterion, and analyze its performance w.r.t. WYSIWYG model
  - Terminate if any row or column outside the range [toofew, toomany] (N.B. *This is different from the earlier assignment*.)
  - Classify the statements in terms of their approximate cost using the WYSIWYG model
  - Submit a document with program & analysis