

CSE524 Parallel Computation

Lawrence Snyder

www.cs.washington.edu/CSEp524

15 May 2007

1

Announcements

- Recall that you need to turn in a brief (paragraph) description ON PAPER of your progress on the project this week
- Account information given out after break

Projects should be underway

2

Comment on Projects

- As the civil rights song puts it, “Keep your eyes on the prize.”
 - n I want to understand conceptually how you used parallelism to solve your problem
 - n I want to understand how your P1 works
 - n I want to know how you analyzed P1’s ||ism using the CTA model and other classroom material to understand its parallel performance
 - n I want to know what you did in response to create P2 and how P2 works

— Once that’s done, anything else is gravy ... I like gravy

ZPL

- ZPL, a research parallel language w/ 3 goals
 - n Performance == as good as platform-specific custom code
 - n Portability == runs well on all platforms
 - n 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

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 ...
 - n ZPL is designed and built on the CTA
 - n ZPL is the first high-level language to achieve “performance portability”
 - n ZPL presents programmers with a visually- cued performance model: WYSIWYG
 - n ZPL is insensitive to shared or message passing architectures, making it universal

ZPL is “designed from first principles”

5

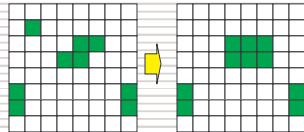
Today’s Plan

- Example programs with discussion of principles
- Key language features
- Review of pgmming w/ performance model
- WYSIWYG performance model
 - n Reflect on Life

6

Conway's Game of Life

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



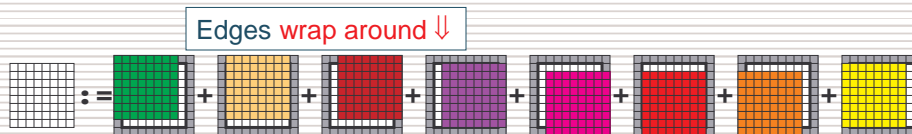
- Organism in next generation if position is alive in this generation and has 2 neighbors, or in this generation it has 3 neighbors
- Or: `(thisGen && neighbors== 2) || (neighbors==3)`

See Life As An Array Computation

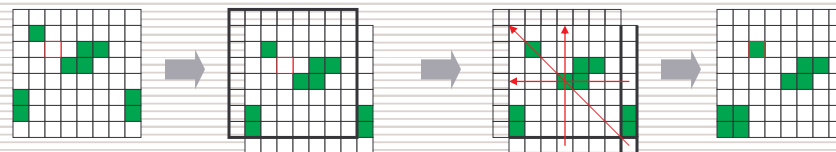
7

Compute Over Whole Arrays

- Count neighbors by adding organisms (bits)



- Closer look at World@^NW



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

8

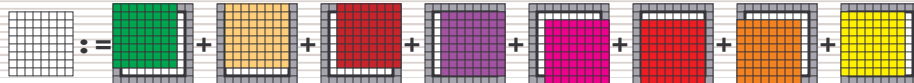
Express Array Computation in ZPL

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);
```

Annotations:

- Red arrow pointing to the neighbor bit additions: "Add up neighbor bits"
- Red arrow pointing to the rule application: "Apply rules to live by"
- Red arrow pointing to the OR operation: "'Or' bits in world to see if any alive"



9

Life In ZPL

```
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 Life

The world is $n \times n$; default to 10

Index set of computation

Problem state, The World

Work array, Number of Neighbors

Entry point procedure

I/O or other data specification

Region R ==> apply ops to all indices

Add 8 nearest neighbor bits - (type coercion like C); caret (^) means toroidal neighbor reference

Update world with next generation

Continue till all die out

10

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

11

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

12

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 separate 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...

13

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 ... in [R], except statement prefixed by [Left] ... end; [Left]
- Operations over region elements performed in parallel

14

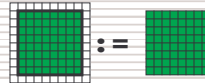
Parallelism In Statement Evaluation

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

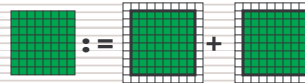
var A, B : $[1..n, 1..n]$ float; C : $[2..n-1, 2..n-1]$ float;

- Then

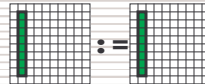
$[2..n-1, 2..n-1]$ A := C;



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



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



15

@ Uses Regions & Directions

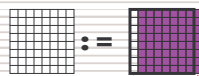
The @ operator combines regions with directions to allow references to neighbors

- Two forms, standard(@) and wrapping(@^)

n 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 :=

16

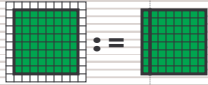
Parallelism In Statement Evaluation

- Let

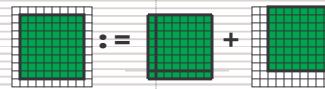
```
var A, B : [1..n,1..n] float; C : [2..n-1,2..n-1] float;
direction east = [0,1]; ne = [-1,1];
```

- Then

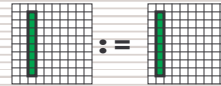
```
[2..n-1,2..n-1] A := C@^east;
```



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



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



17

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

18

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:=

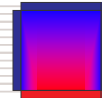
[n of C]Y:=

[C]Y:=X@ne

execution →

19

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

20

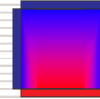
“High Level” Logic Of J-Iteration

```

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

21

Reduce

- ZPL has ‘full’ reduce: +<<, *<<, max<<, ...
 - ZPL also has ‘partial’ reduce
 - n Applies reduce across rows, down columns,...
 - n 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
- n In these examples, result stored in 1st row/col

Collapsed dimensions indicate reduce dimension(s)

22

Flood -- Inverse of Partial Reduce

- Reduce “reduces” 1 or more dimensions
- Opposite is flood -- fill 1 or more dimensions

$[1..n, 1..n] B := \gg [1..n, 1] A;$



$[1..n, 1..n] B := \gg [1..n, n] A;$



- The replication uses multicast, often an efficient operation

23

Closer Look At Scaling Each Row

$[1..m, 1] \text{MaxC} := \max\ll [1..m, 1..n] A;$ *Max of each row*

$[1..m, 1..n] A := A / \gg [1..m, 1] \text{MaxC};$ *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	3	6	
3	3	3	3	3	3	3	3	
8	2	4	0	8	8	8	8	
	A			MaxC		$\gg [1..m, 1] \text{MaxC}$		

Keep MaxC a 2D array to control allocation

24

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

25

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:

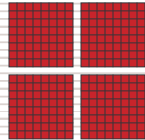
```
[1..m,*]   MaxC := max<< [1..m,1..n] A;
[1..m,1..n] A := A / MaxC;      --Scale A;
```

The promotion of flooded arrays is only logical

26

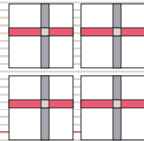
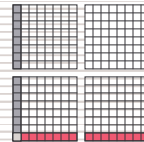
Flood v. Singleton Difference

- 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

27

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

28

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

29

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;
```

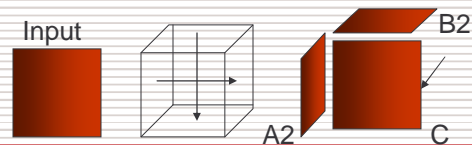
30

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);
    
```



31

Partial Scan

- Partial scans are possible too, but unlike reduction 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

```

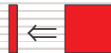
      1  1  1  1  1  2  3  4
+|[2] 1  1  1  1  1  2  3  4
      1  1  1  1  1  2  3  4
      1  1  1  1  1  2  3  4
    
```

32

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



33

Index1 ...

- ZPL comes with “constant arrays” of any size
- Index i means indices of the i^{th} 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	1	2	3	4	1	0	0	0
2	2	2	2	1	2	3	4	0	1	0	0
3	3	3	3	1	2	3	4	0	0	1	0
4	4	4	4	1	2	3	4	0	0	0	1

Index1

Index2

L

34

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

35

Remap (Gather)

The index array in the i th position gives the indices for the i th dimension

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

Gather: For a position, where does value come from

$a\ c\ e\ b\ d\ f \Leftrightarrow a\ b\ c\ d\ e\ f\#[1\ 3\ 5\ 2\ 4\ 6]$

a	e	i	m	a	b	c	d	#	1	2	3	4	1	1	1	1
b	f	j	n	e	f	g	h		1	2	3	4	2	2	2	2
c	g	k	o	i	j	k	l		1	2	3	4	3	3	3	3
d	h	l	p	m	n	o	p		1	2	3	4	4	4	4	4
AT				A				$Index2$				$Index1$				

36

Remap (Scatter)

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

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

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

a	e	i	m	[1	2	3	4	1	1	1]	a	b	c	d	
b	f	j	n	#	1	2	3	4	2	2	2	2	:=	e	f	g	h
c	g	k	o		1	2	3	4	3	3	3	3		i	j	k	l
d	h	l	p		1	2	3	4	4	4	4	4		m	n	o	p

AT
Index2
Index1
A

37

Break

38

CTA and ZPL

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

Building on CTA is a main contribution of ZPL

39

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
 - n Programmer needs to know the CTA
 - n Language constructs' performance must be described in CTA terms
 - n Information must "compose"
- In CSE524 we've always explained performance this way

40

Assumes Many Pts Per Processor

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

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

Let's walk-through the process

41

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

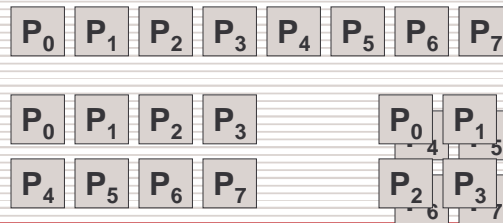
42

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

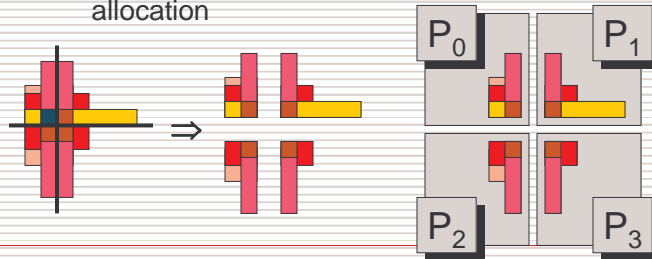


43

Allocate Bounding Region to Grid


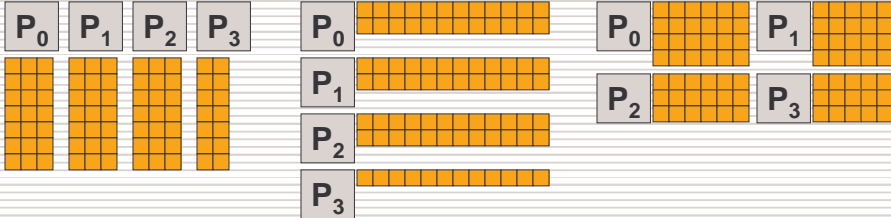
The bounding region is allocated to processor grid in the “most balanced” way possible

- 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



44

More Typical Allocations

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

- 3D ...

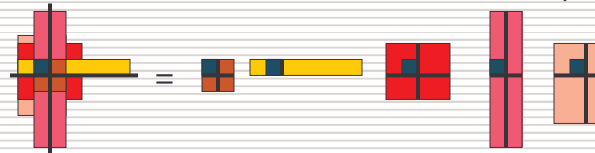
ZPL uses Ceiling/Floor and includes fluff

45

Fundamental Fact of ZPL

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

ZPL has the property that for any arrays \mathbf{A} , \mathbf{B} of the same rank and having an element $[i, \dots, k]$, that element of each will be stored on the same processor



Corollary: Element-wise operations do not require any communication: $[\mathbf{R}] \dots \mathbf{A} + \mathbf{B} \dots$

46

Performance Model (WYSIWYG)

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

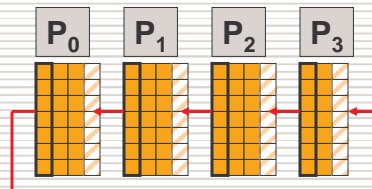
- n Performance is given in terms of the CTA
- n Performance is relative, e.g. x is more expensive in communication than y
- o Rules...
 - A + B -- Element-wise array operations
 - o No communication
 - o Per processor work is comparable to C
 - o Work fully parallelizable, i.e. time = work/P

47

Rules Of Operation (continued)

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

Arrays allocated with "fluff" for every direction used



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

48

<< || >>

+<<A -- Reduce

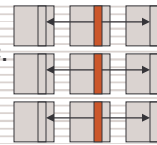
- Accumulate local elements
- $O(\log P)$ tree accumulation, or better
- Broadcast, which is worst case $O(\log P)$, but usu. less

+|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”



49

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

50

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;

```

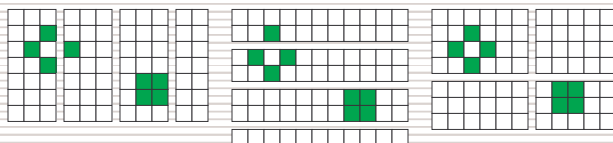
Code for performance costs implied by WYSIWYG

51

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 $c\lambda$

52

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

53

How Many Generations?

- Compute count of generations before life dies out

```
program Life;
config var n : integer;
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;
```

54

How Many Generations?

Testing on each generation may 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;
```

55

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

56

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

57

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 ...
 - n It may be better ... WYSIWYG is a worst case that often doesn't materialize
 - n Aggressive optimizations help a lot

If there are any surprises, they'll be pleasant

58

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

59

Bottom Line for ZPL In 524

- 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

60

Homework

- No Textbook Reading For Next Week
- Project
 - n Bring **paper** statement of progress to class