ZPL's Performance Model

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

1

Remap Review

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

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

The lists in brackets are arrays of indices saying where the elements come from or go to

$A_{i,j} := B[I1_{i,j}, I2_{i,j}]$ Gather

$A[I1_{i,j}, I2_{i,j}] := B_{i,j}$ Scatter

2

Remap (continued)

- The transpose idiom ...

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

- Because scattered values could collide, use the “aggregate” assignments, e.g. +=, to combine … *or take your chances*
- The use of “computable” subscripts such as Index*i* is common and leads to optimizations
- Rank Change:

  [1..n] V := A#[Index1, Index1]; Select diagonal

3

Indexed Arrays

ZPL has a (nonparallel) indexed array

- Like scalars, they’re replicated on each processor and kept coherent … good for look-up tables

```
var cell : array [0..3] of char;
cell[0]:= 'R'; cell[1]:= 'B'; cell[2]:= ' ';
  cell[3]:= ' ';
[R] ZeroTo3 := ((Index1-1)*n+Index2)%4;
[R] A := cell[ZeroTo3];
```

- Indexed arrays are often used as constituents of other data structures

  var B : [R] array[1..10] of float; -- array of 10 item vectors

4

Comments on Procedures

- Procedures … must be defined or prototype'd before used
- Old syntax:
  - default call by value
  - `var` is call by reference
- New syntax:
  - `const` : parameter not changed or assigned to (default)
  - `in` : call by value, i.e. not changed, but formal assigned
  - `out`: call by reference, uninitialized
  - `inout` : call by reference initialized

"Applicable" region for proc may be from call site

5

ZPL's Performance Model

In F or C we write fast (or efficient in other ways) programs because we know roughly how they will run … thanks to vN model

ZPL's goal is to provide the equivalent for parallel computing … thanks to CTA model

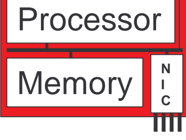
Strategy:

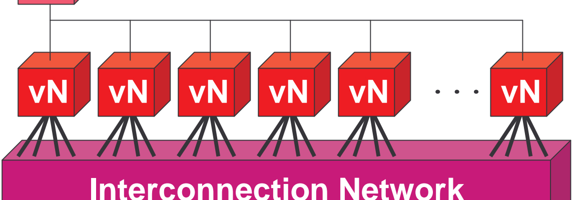
- Adopt the CTA as the machine model
- Implement compiler with CTA as target
- Explain compiler/runtime behavior w.r.t. all operations in the language

6

ZPL Compiles to the CTA

- The CTA is an abstract machine modeling practical parallel computers
  - P processors
  - λ >>1 latency to nonlocal memory
  - Unspecified interconnect





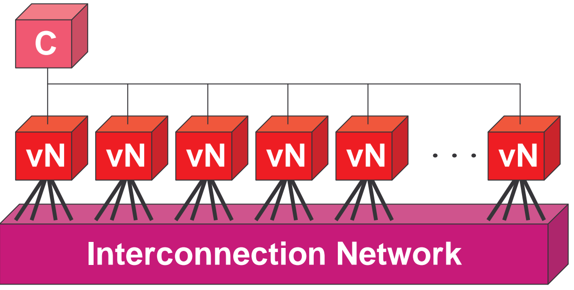
Properties of the CTA ...

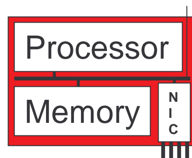
- Processor Elements are von Neumann machines -- they could be SMPs or other parallel designs, but they execute vN ISA
- PEs are connected by a sparse (O(n) edges) point-to-point network, not specified
- PE's connection to network is "narrow"
- Latency λ >> 1 for non-local memory reference
- Controller has "thin" broadcast connection to all processors

CTA Emphasizes Locality

CTA models … where the data is & where it goes

- The λ parameter captures the benefit of exploiting locally stored values
  - λ says moving data to other PEs has significant cost

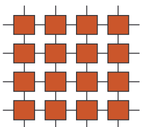




9

The λ value

- λ [it's big, ~3 digits] is purposely vague
  - The cost is not known and cannot be known because every machine is different
    - There is no agreement yet on *the* parallel architecture
    - λ depends on bandwidth, engineering, interfaces, mem
  - The cost cannot be a single number because of network topology, conflicts, etc.



  - Optimizing for communication -- if possible -- should be done by the compiler for the machine not the programmer for eternity

10

CTA Abstracts Existing Architectures

- The MIMD parallel machines -- all machines since mid 1980s -- are modeled by CTA
- iPSC, HyperCube, UltraComputer, Cedar, SP-1, iPSC2, KSR-1, CM-5, Sequent, Touchstone, T3D, Dash, J-Machine, SP-2, KSR-2, T3E, Tera, SC-2000, etc., etc., etc.
- A few minor adjustments may be needed
  - Some machines do not have a controller (front end machine), but PE_0 can be both a PE and Controller

11

Clusters and SMPs

- (Beowulf) Clusters are popular because they are easy to build, simple to operate and most of the software is Open Source
  - Clusters built using a point-to-point network (Myrinet) are CTA machines
  - Clusters with bus-architectures fail to meet the point-to-point network feature, but a decent bus is an OK approximation (and there is no other model!)
- SMPs are technically not CTA machines, but since they are “better” ($\lambda \approx 1$), using CTA guidelines is OK

12

The Task ...

How is ZPL's performance model given to programmers?

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

ZPL Assumes Many Points Per Processor

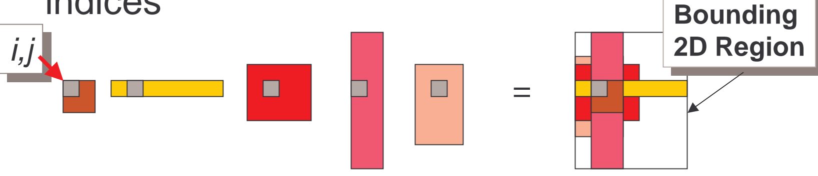
ZPL allocates regions (and therefore arrays) to processors so many contiguous elements are assigned to each

- Array Allocation Rules
  - Union the regions together 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

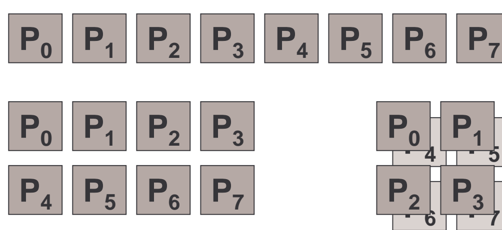
15

Get Processor Number + 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:

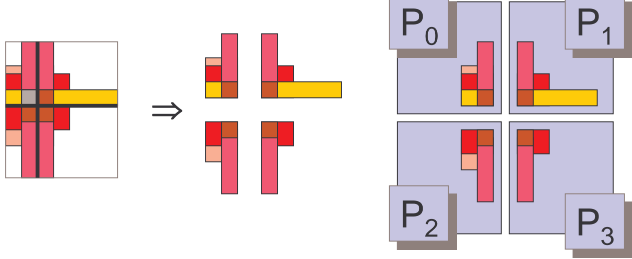
The CTA does not favor any arrangement, so use a generic one



Allocate the Bounding Region to the Grid

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

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



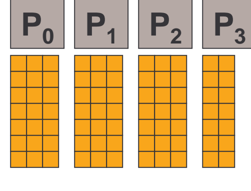
17

More Typical Allocations

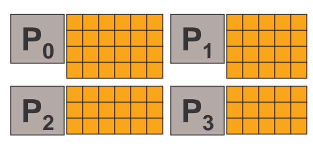
- 1D is segmented;

P_0 P_1 P_2 P_3

- 2D is panels, strips or blocks;



P_0 P_1 P_2 P_3



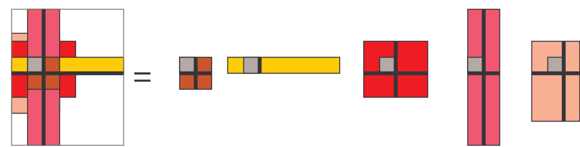
- 3D ...

18

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 …`

19

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

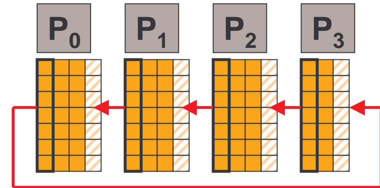
- No communication
- Per processor work is comparable to C
- Work fully parallelizable, i.e. time = work/P

20

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

21

<< || >>

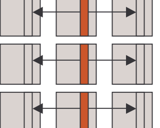
`+<<A` -- Reduce

- Accumulate local elements
- Ladner/Fischer 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”

22

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 say 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 The WYSIWYG In Real Life...

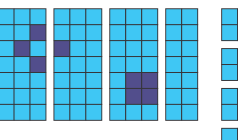
```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];
        BigR = [0..n+1,0..n+1];
direction  N = [-1, 0]; NE = [-1, 1];
           E = [ 0, 1]; SE = [ 1, 1];
           S = [ 1, 0]; SW = [ 1,-1];
           W = [ 0,-1]; NW = [-1,-1];
var NN : [R] ubyte; TW : [BigR] boolean;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

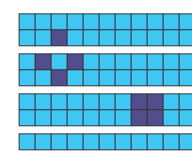
Code for performance costs implied by WYSIWYG

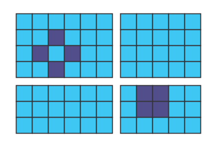
Analyzing Life By Color

- 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

Add a counter to previous program

```
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;
```

27

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;
```

28

Optimize To Reduce Termination Tests

```
program Life;
config var n : integer = 512; epoch : integer = 50;
...
var NN:[R] ubyte; TW,TWo:[R] boolean; count:integer = 0;
procedure Live(gens:integer);
   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

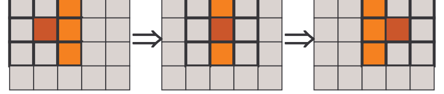
Redo World End

Report

29

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

30

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

31

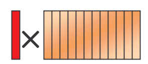
Summarizing WYSIWYG Model

- 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

32

ZPL Exercise

- Given a [1..m,1..n] array of floats, each column of which corresponds to an "observation" of m properties of something
- Given a weighting vector [1..m,1]
- Task: Order the observations in numerical order (smallest to largest) of their weighted sum
  - Compute weighted sum, S, of each column 
  - Sort S, as below
  - Reorder columns based on the sort

Sorting by Rank

- Sort with an algorithm that compares S with itself using <=, yielding a Boolean array; count up the ones to determine where in the final sequence an element should go

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