The Performance Model of ZPL

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

One Solution to Game of Life ...

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

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

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

Moving up

Moving down

Drainage (continued)

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

Boundaries not included
User Defined Reductions

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

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

Three Components

• Reduction op<<A has three parts:
  – Initialization ... first step is initial_val op A[1]
  – Local Combine ... intermed_val op A[i]
  – Global Combine ... intermed_val op intermed_val

• Users define all three (overloaded with the same name) using standard procedure syntax

 Initialization P1 result P2 result Local

∅ ⊗ 3 ⊗ 1 ⊗ 4 ⊗ 1  ∅ ⊗ 5 ⊗ 9 ⊗ 2 ⊗ 6

Global
Largest Two Elements

`max<<A` finds the largest ... find the largest two

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

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

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

Permute

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

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

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

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

- To transpose an array, simply write
  \[1..n,1..n\] A := A# [Index2, Index1];
  \[
  \begin{array}{ccc}
  1 & 2 & 3 \\
  1 & 1 & 1 \\
  \end{array}
  \]
  \[1..3,1..3\] A := A# [1 2 3 , 2 2 2 ]
  \[
  \begin{array}{ccc}
  1 & 2 & 3 \\
  3 & 3 & 3 \\
  \end{array}
  \]

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

Permute (continued)

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

In recent months permute has been upgraded
  - Syntax
  - Combining
  - Rank change
  - Serious Optimizations
ZPL’s got lots more ...

“Core ZPL” has been presented, but there is more

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

... and greater unification

ZPL Built On The CTA

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

How is ZPL’s performance model given?

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

Assigning Work To Processors

- Two views for data parallel computation, e.g. 
  \[ A := B + C \text{@east} \]
  as found in ZPL’s dense arrays

  - Virtual Processors: Each processor is allocated the work of scalar data values, that is, think of a situation where there are as many processors as array elements

  - Variable Points Per Processor: Each processor is allocated from 1 to n values, but usually many
    - 1 element is virtual processor case
    - n elements is the sequential case
    - Algorithms with this feature are scalable
Assigning Work (continued)

- Many claim the two models are equivalent, since virtual can be emulated.
- They’re not equivalent because ...
  - Emulation misses the advantages of long instruction sequences:
    - pipelining
    - caching
    - prefetch
  - Virtual misses significant costs of local shifting:
    \[ V := V @ \text{right} \]

ZPL Assumes Many Points Per Processor

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

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

Consider a walk-through of the process
Union The Regions Together

Create the “footprint” of the regions

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

The bounding region is allocated to processors

Get Processor Number + Arrangement

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

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

The CTA does not favor any arrangement, so use a generic one
Allocate the Bounding Region to the Grid

The bounding region is allocated to the grid in the “most balanced” arrangement possible:

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

ZPL allocates multiple elements per processor

Break
Fundamental Fact of ZPL

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

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

Corollary: Element-wise operations do not require any communication: $[R] \ldots A+B \ldots$

Performance Model (WYSIWYG)

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

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

• Rules…
  $A + B$ -- Element-size array operations
  • No communication
  • Work comparable to $C$
  • Work fully parallelizable, i.e. time $=$ work/$P$
Rules of Operation (continued)

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

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

**+<<** -- Reduce and scan

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

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

- Multicast array segments
- Represent data non-redundantly

---

**Rules of Operation (continued)**

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

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

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

**ZPL is only parallel language with performance model**
Applying The WYSIWYG Model To Life...

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

Analyzing Life By Color

- Blue: Effectively no time ... each processor does set-up and scalar computation locally
- Gold: Element-wise computation perfectly parallel ... Index\(i\) constants are generated

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

```
+ Delay is c\(\lambda\)
```
Analyzing By Color (continued)

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

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

    ![Diagram](image)

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

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.
Comparing Cannon’s and SUMMA MM

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

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
Programming Cannon’s In ZPL

- Handle the skewed arrays

\[
\begin{array}{cccc}
  c_{11} & c_{12} & c_{13} & a_{11} & a_{12} & a_{13} & a_{14} \\
  c_{21} & c_{22} & c_{23} & a_{21} & a_{22} & a_{23} & a_{24} \\
  c_{31} & c_{32} & c_{33} & a_{31} & a_{32} & a_{33} & a_{34} \\
  c_{41} & c_{42} & c_{43} & a_{41} & a_{42} & a_{43} & a_{44} \\
\end{array}
\]

\[
\begin{array}{cccc}
  b_{13} & c_{11} & c_{12} & c_{13} & a_{11} & a_{12} & a_{13} & a_{14} \\
  b_{12} & b_{23} & c_{21} & c_{22} & c_{23} & a_{22} & a_{23} & a_{24} & a_{21} \\
  b_{11} & b_{22} & b_{33} & c_{31} & c_{32} & c_{33} & a_{33} & a_{34} & a_{31} & a_{32} \\
  b_{21} & b_{32} & b_{43} & c_{41} & c_{42} & c_{43} & a_{44} & a_{41} & a_{42} & a_{43} \\
  b_{31} & b_{42} & b_{13} & c_{11} & c_{12} & c_{13} & a_{11} & a_{12} & a_{13} & a_{14} \\
  b_{41} & b_{12} & b_{23} & a_{11} & a_{12} & a_{13} & a_{14} & a_{22} & a_{23} & a_{24} & a_{21} \\
\end{array}
\]

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

Four Steps of Skewing A

\[
\begin{array}{cccc}
  \text{for } i := 2 \text{ to } m & & \text{end;}
  \end{array}
\]

\[
\begin{array}{cccc}
  a_{11} & a_{12} & a_{13} & a_{14} & a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} & a_{21} & a_{22} & a_{23} & a_{24} & a_{21} \\
  a_{31} & a_{32} & a_{33} & a_{34} & a_{31} & a_{32} & a_{33} & a_{34} & a_{31} \\
  a_{41} & a_{42} & a_{43} & a_{44} & a_{41} & a_{42} & a_{43} & a_{44} & a_{41} \\
\end{array}
\]

\[
\begin{array}{cccc}
  \text{Initial} & & \text{end;}
  \end{array}
\]

\[
\begin{array}{cccc}
  i = 3 \text{ step} & & i = 4 \text{ step} \\
  a_{11} & a_{12} & a_{13} & a_{14} & a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{22} & a_{23} & a_{24} & a_{21} & a_{22} & a_{23} & a_{24} & a_{21} \\
  a_{32} & a_{33} & a_{34} & a_{31} & a_{32} & a_{33} & a_{34} & a_{31} \\
  a_{43} & a_{44} & a_{41} & a_{42} & a_{44} & a_{41} & a_{42} & a_{43} \\
\end{array}
\]

… And Skew B vertically
Cannon's Declarations

For completeness, when $A$ is $m \times n$ and $B$ is $n \times p$, 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}

for i := 2 to m do -- Skew $A$
[1..m, 1..n] A := A@^right;
end;
for i := 2 to p do -- Skew $B$
[1..n, 1..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
[Res] A := A@^right; -- Rotate $A$
[Res] B := B@^below; -- Rotate $B$
end;
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
  C := C + A*B; -- For product
[Res] A := A^right; -- Rotate A
[Res] B := B^below; -- Rotate B
end;

Comms have λ latency, but much data motion

SUMMA Algorithm Analysis

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

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

SUMMA does not require as much comm or data motion as Cannon’s, nor does it “touch” the array as much
A Parallel Programming Technique

ZPL’s approach admits new programming tricks

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

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

3D MM Is A Problem Space Promo Alg

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

\[ \text{region } IK = [1..n,*,1..n] \]
\[ JK = [*,1..n,1..n]; \]
\[ IJ = [1..n,1..n,*]; \]
\[ IJK = [1..n,1..n,1..n]; \]
\[ [IK] A2 := A[#[Index1,Index3,Index2]]; \]
\[ [JK] B2 := B[#[Index3,Index2,Index1]]; \]
\[ [IJ] C := +<<[IJK] (>>[IK] A2) * (>>[JK] B2); \]
Sorting By PSP

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

\[1,1..n\] begin
\[1..n,1\]
\[ST := S#^[Index2,Index1]; \quad \text{- Transpose input}\]
\[P := \langle\langle [1..n,1..n]\{\langle\ast,1..n\rangle \Rightarrow \Rightarrow [1..n,\ast]\}\rangle\rangle ST;\]
\[\quad \text{- Compare } n^2 \text{ items, reduce}\]
\[S := S#^[Index1,P]; \quad \text{- Reorder by permutation}\]
end;

Summary

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