Part V: Programming Alternatives

Goal: Take a brief look at languages that to provide compiler help and raise abstraction level

Improving on Message Passing

- Message passing is “the lowest of the low”, but remains in widespread use because …
  - It is required for clusters, supercomputers, etc.
  - Achieving performance is possible, though with effort
  - Portability is essential for long-lived programs
- There must be a better way … but it’s been harder than expected to find
Recent Notable Efforts: PGAS

- Greatest potential to assist programmer comes from hiding communication calls
  - Compilers can generate the calls
  - Need interface to specify which are local/global

- Concept: Partitioned Global Address Space
  - Overlay global addressing on separate memories
  - PGAS tends to use 1-sided comm as simplification


Extend Languages

- Three PGAS language
  
<table>
<thead>
<tr>
<th>CAF</th>
<th>UPC</th>
<th>Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co-Array Fortran</td>
<td>Universal Parallel C</td>
<td>Titanium</td>
</tr>
<tr>
<td>Numrich &amp; Reed</td>
<td>El Ghazawi, Carlson &amp; Draper</td>
<td>Yelick</td>
</tr>
<tr>
<td>Extends Fortran</td>
<td>Extends C</td>
<td>Extends Java</td>
</tr>
</tbody>
</table>

- Developed around 2000 +/- & Implemented
  - Similarities: GAS, comm handled by compiler/rt, programmer controls work/data assignment
  - Differences: Most everything else
**Co-Array Fortran**

- Incredibly elegant (for Fortran) extension

```fortran
real, dimension(n,n)[p,*]: a,b,c
...
do k=1,n
do q=1,p
    c(i,j)[myP,myQ]=c(i,j)[myP,myQ]+a(i,k)[myP,q]*b(k,j)[q,myQ]
enddo
enddo
```

**UPC**

- Data can be allocated either shared or private; shared is assigned cyclically or BC
- Pointers are an issue

<table>
<thead>
<tr>
<th>Property of pointer</th>
<th>Private</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property of reference</td>
<td>Private-Private, p1</td>
<td>Private-Shared, p2</td>
</tr>
<tr>
<td></td>
<td>Shared-Private, p3</td>
<td>Shared-Shared, p4</td>
</tr>
</tbody>
</table>

```c
int *p1; /* private ptr pointing locally */
shared int *p2; /* private ptr pointing into shared space */
int *shared p3; /* shared ptr pointing locally */
shared int *shared p4; /* shared ptr pointing into shared space */
```
UPC Code for Vector Sum

```c
shared int v1[N], v2[N], v1v2sum[N];

void main()
{
    int i;
    shared int *p1, *p2;
    p1=v1;
    p2=v2;
    upc_forall(i=0; i<N; i++, p1++, p2++;i)
    {
        v1v2sum[i] = *p1 + *p2;
    }
}
```

Titanium

- **Java extensions including**
  - "regions", which support safe, performance-oriented memory management as an alternative to garbage collection."
  - **foreach** is an unordered iteration, which logically raises the concurrency:
    ```java
    foreach ( ... ) {
    }
    ```
  - Used with the concept of a **point**, tuple of integers that range over a **domain**
Titanium Code for MM

```java
public static void matMul(double [2d] a,
    double [2d] b,
    double [2d] c)
{
    foreach (ij in c.domain())
    {
        double [1d] aRowi = a.slice(1, ij[1]);
        double [1d] bColj = b.slice(2, ij[2]);
        foreach (k in aRowi.domain())
        {
            c[ij] += aRowi[k] * bColj[k];
        }
    }
}
```

Summarizing PGAS Languages

- The languages improve on the alternative--base language + MPI
- Compiler provides significant help, but the need to be attuned to subtle detail remains
- Deep issues
  - Global address space+private are good, but how they "play together" remains unclear
  - Better abstractions to reduce detail
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

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)

- Edges wrap around

- Closer look at World@^NW

- TW@^nw is the array of Northwest neighbors
Express Array Computation in ZPL

Conway's Life: The World is bits

```
[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

![Matrix of neighbor bits]

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 x n: default to 10
Index set of computation
Problem state, The World
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); carat(^) means
toroidal neighbor reference
Update world with next generation
Continue till all die out
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;  

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
  
  \[
  \begin{align*}
  \text{var } & A, B, C : [R] \text{ double;} \\
  \text{var Seq : [V] boolean;} \\
  \text{var Huge : [0..2^n, -5..5] float;} \\
  \end{align*}
  \]
- 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
  
  \[
  \begin{align*}
  [1..n,1..n] & A := B + C; \\
  [R] & A := B + C; \quad \text{-- Same as above} \\
  \end{align*}
  \]
- Regions are scoped
  
  - [R] begin
  
  \[
  \begin{align*}
  \text{… } & \text{All array computations in compound} \\
  \text{end;} & \text{statements are performed over indices} \\
  \text{[Left]} & \text{in [R], except statement prefixed by} \\
  \text{[Left]} & \text{end;} \\
  \end{align*}
  \]
- 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

\[
\text{var } A, B : [1..n,1..n] \text{ float}; C : [2..n-1,2..n-1] \text{ float};
\]

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

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

Then

\[ [2..n-1,2..n-1] A := C^\text{east}; \]
\[ [2..n-1,2..n-1] A := C^\text{ne} + B^\text{ne}; \]
\[ [2..n-1,2] A^\text{east} := B; \]

Reductions, Global Combining Operations

Reduction (\(<\text{=}\)) “reduces” the size of an array by combining its elements

Associative (and commutative) operations are
\(<\text{=}\), *\(<\text{=}\), &\(<\text{=}\), |\(<\text{=}\), max\(<\text{=}\), min\(<\text{=}\)

\[ [1..n, 1..n] \text{ biggest } := \text{max\(<\text{=}A;} \]
\[ [R] \text{ all_false } := \text{|<\text{=} TW;} \]

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

```plaintext
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];
```

Applying Ideas: Jacobi Iteration

- Model heat defusing through a plate
- Represent as array of floating point numbers
- Use a 4-point stencil to model defusing
- Main steps when thinking globally
  - Initialize
  - Compute new averages
  - Find the largest error
  - Update array
  - ... until convergence

- High-level Language should match high-level thinking
“High Level” Logic Of J-Iteration

program Jacobi;
config var n : integer = 512;
    eps : float = 0.00001;
region R = [1..n, 1..n];
    BigR = [0..n+1,0..n+1];
direction N = [-1, 0]; S = [ 1, 0];
    E = [ 0, 1]; W = [ 0,-1];
var Temp : [R] float;
    A : [BigR] float;
    err : float;
procedure Jacobi();
    [R] begin
        [BigR] A := 0.0;
        [S of R] A := 1.0;
        repeat
            Temp := (A@N + A@E + A@S + A@W)/4.0;
            err := max<< abs(Temp - A);
            A := Temp;
        until err < eps;
    end;

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

\[ [1..m,1] MaxC := max<<[1..m,1..n] A; \quad \text{Max of each row} \]
\[ [1..m,1..n] A := A / >>[1..m,1] MaxC; \quad \text{Scale each row} \]

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

\[
\begin{array}{cccccccc}
2 & 4 & 4 & 2 & 4 & 4 & 4 & 4 \\
0 & 2 & 3 & 6 & 6 & 6 & 3 & 6 \\
3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
8 & 2 & 4 & 0 & 8 & 8 & 8 & 8 \\
\end{array}
\]

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 vs. 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...

```plaintext
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

<table>
<thead>
<tr>
<th>c11</th>
<th>c12</th>
<th>c13</th>
<th>a11</th>
<th>a12</th>
<th>a13</th>
<th>a14</th>
<th>b11</th>
<th>b12</th>
<th>b13</th>
</tr>
</thead>
<tbody>
<tr>
<td>c21</td>
<td>c22</td>
<td>c23</td>
<td>a21</td>
<td>a22</td>
<td>a23</td>
<td>a24</td>
<td>b21</td>
<td>b22</td>
<td>b23</td>
</tr>
<tr>
<td>c31</td>
<td>c32</td>
<td>c33</td>
<td>a31</td>
<td>a32</td>
<td>a33</td>
<td>a34</td>
<td>b31</td>
<td>b32</td>
<td>b33</td>
</tr>
<tr>
<td>c41</td>
<td>c42</td>
<td>c43</td>
<td>a41</td>
<td>a42</td>
<td>a43</td>
<td>a44</td>
<td>b41</td>
<td>b42</td>
<td>b43</td>
</tr>
</tbody>
</table>

Col = a11 a11 a11 × b11 b12 b13 = a11b11 a11b12 a11b13
<table>
<thead>
<tr>
<th>a21</th>
<th>a21</th>
<th>a21</th>
<th>b11</th>
<th>b12</th>
<th>b13</th>
</tr>
</thead>
<tbody>
<tr>
<td>a31</td>
<td>a31</td>
<td>a31</td>
<td>b11</td>
<td>b12</td>
<td>b13</td>
</tr>
<tr>
<td>a41</td>
<td>a41</td>
<td>a41</td>
<td>b11</td>
<td>b12</td>
<td>b13</td>
</tr>
</tbody>
</table>

SUMMA is the easiest MM algorithm to program in ZPL

SUMMA Algorithm (continued)

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

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

--- or, more simply ---
\[
\text{for } k := 1 \text{ to } n \text{ do}
\]
\[
[1..m,1..p] \quad C += ( >>[ ,k] \ A)*( >>[k, ] \ B); \\
\]
\[
\text{end;}
\]
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

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}
\Rightarrow
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
\end{array}
\]
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

ZPL comes with “constant arrays” of any size
- Index means indices of the \( i \)th dimension

\[
\begin{align*}
1 & 1 & 1 & 1 & 1 & 2 & 3 & 4 & 1 & 0 & 0 & 0 \\
2 & 2 & 2 & 2 & 1 & 2 & 3 & 4 & 0 & 1 & 0 & 0 \\
3 & 3 & 3 & 3 & 1 & 2 & 3 & 4 & 0 & 0 & 1 & 0 \\
4 & 4 & 4 & 4 & 1 & 2 & 3 & 4 & 0 & 0 & 0 & 1 \\
\end{align*}
\]

Index arrays: compiler created using no space
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: \( AT[i,j] = A[j,i] \)
  - \([R] AT := A\#[\text{Index2,Index1}]\); -- Standard Idiom for transpose

Remap (Gather)

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

Gather: For a position, where does value come from

\[
\begin{array}{cccccc}
\text{a} & \text{c} & \text{e} & \text{b} & \text{d} & \text{f} \\
\text{a} & \text{e} & \text{i} & \text{m} & \text{b} & \text{f} & \text{j} & \text{n} & \text{e} & \text{f} & \text{g} & \text{h} \\
\text{c} & \text{g} & \text{k} & \text{o} & \text{i} & \text{j} & \text{k} & \text{l} & \text{d} & \text{h} & \text{l} & \text{p} & \text{m} & \text{n} & \text{o} & \text{p} \\
\end{array}
\]

\[
\begin{array}{c}
\# \[1 3 5 2 4 6\] \\
\end{array}
\]

\[
\begin{array}{c}
\text{AT} \\
\text{A} \\
\text{Index2} \\
\text{Index1} \\
\end{array}
\]
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?**

<table>
<thead>
<tr>
<th>a</th>
<th>d</th>
<th>b</th>
<th>e</th>
<th>c</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

AT: Index2, Index1, A

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
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
### 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 \([i, \ldots, k]\), that element of each will be stored on the same processor.

\[
\begin{align*}
A & = B \\
\text{Corollary: Element-wise operations do not require any communication:} \ & \ [R] \ ... \ A + B \ ...
\end{align*}
\]
Applying WYSIWYG In Real 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 NN : [R] ubyte; TW : [BigR] boolean;
procedure Life();
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;
```

Analyzing Life By Color

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

How is TW allocated on 4 procs? Three basic choices...
Analyzing By Color (continued)

- **Purple**: Element-wise computation with @ operations … expect \( \lambda \) 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;
```
How Many Generations?

Testing on each generation may be excessive -- analyze

```pascal
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

```pascal
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;
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 Step 1

\[ c_{43} = c_{43} + a_{41}b_{13} \]

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

And Skew B vertically

Cannon’s Declarations

For completeness, when A is m\times n, B is n\times 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 : [*,*..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**
  - Declare
  - Skew A
  - Skew B
  - Initialize
  - loop til n
  - C+=A*B
  - Rotate A,B

  **SUMMA**
  - Declare
  - Initialize
  - loop til n
  - Flood A
  - Flood B
  - C+=A*B

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 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 $\lambda$, latency, but much data motion

SUMMA Algorithm Analysis

The flood is (likely) more expensive than $\lambda$ time, but less that $\lambda(\log P)$ ... 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 ...

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of communications</th>
<th>Communication complexity</th>
<th>Communication volume</th>
<th>Flops</th>
<th>Elements referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannon</td>
<td>4n</td>
<td>1</td>
<td>n</td>
<td>$2n^3 - n^2$</td>
<td>$n \cdot (2\frac{n^2}{p} + 3n^2)$</td>
</tr>
<tr>
<td>SUMMA</td>
<td>2n</td>
<td>$\log p$</td>
<td>n</td>
<td>$2n^3$</td>
<td>$n \cdot (n^2 + 2n)$</td>
</tr>
</tbody>
</table>

MM No.s
Optimizations Can Help

- WYSIWYG is the worst case … optimizations are possible …
- Sequential Optimizations e.g. stencil opts
  7 additions are used for each element, but fewer adds are sufficient
- Sum of orange items performed once

- 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
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 $P$, $\lambda$, locality, etc.
Assessing the State of the Art

- There are many criteria we can use to assess parallel programming tools
  - Correctness
  - Performance
  - Portability
  - Scalability

Correctness

- A key property of a parallel programming tool promoting correctness is ...
  - \textit{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 \textit{P}-independent
  - Which languages are \textit{P}-independent?
- Call a \textit{P}-independent language \textit{Global View}
Performance

- Performance is often the point of writing parallel programs ... which facilities allowed us to be successful?
  - Low-level control -- do-it-yourself programming
  - Awareness of locality, dependences, etc.
  - Conform to restricted model
- High-level and low-level tools differ ... for high level, approximate “custom” solution

Portability

- By “universality theorem” all || programs are portable in the sense of “running” on a platform ... we want performance portability
- “Fast on one machine, fast on all machines”
- Greater abstraction improves portability by insulating from machine specifics, but ...
  - Does “insulation” introduce overhead?
  - Can performance-critical programming decisions be made?
Scalability

☐ As the cost of hardware continues to fall, everyone wants to add more processors…

☐ Considerations
  - Is problem large enough?
  - What dependences do added processors add?
  - How frequently are processors waiting for serialized or synchronized behavior?

List of Facilities

☐ We touched on these || tools
  - Posix Threads
  - OpenMP
  - MPI
  - PGAS languages: CAF, UPC, Ti
  - ZPL
  - NESL