ZPL, A Parallel Programming Language

ZPL is an implicitly parallel array programming language based on the CTA machine model. Though designed for scientific computation, ZPL illustrates fundamental ideas in parallel computing essential to all application areas.

Practical Considerations

- The purpose of learning ZPL is to illustrate the fundamental point from the first lecture that a parallel machine model enables one to write programs independent of target machine, yet still have sufficient understanding of their performance to estimate how they will run.
- Find documentation on the ZPL home page: www.cs.washington.edu/research/zpl/docs/descriptions/guide.html
- ZPL has been installed on orcas/sanjuan

Homework Assignment

- This lecture provides sufficient instruction to write many ZPL programs
- Two straightforward computations are
  - Game of Life
  - All Pairs Shortest Path, based on Warshall’s Algorithm
- These problems are further specified on the class web page

ZPL Overview

- ZPL’s main data structure is a dense array
- Computation is expressed as operations on whole arrays, i.e. A+B adds arrays elementwise
- Parallelism is implicit, i.e. inferred by the compiler from the array expressions
- ZPL is compiled, not interactive like MATLAB
- ZPL compiles to ANSI C which is compiled with machine specific libraries to the target parallel computer

ZPL Factoids

- Development Milestones
  - ZPL design & implementation began in 3/93
  - Portability & performance demonstrated 7/94
  - Compiler and run-time system released 7/97
- Claims
  - Portable to any (MIMD) parallel computer
  - Performance comparable to C with user specified communication
  - Generally out performs High Performance Fortran
  - Convenient and intuitive
- ZPL is a proper subset of Advanced ZPL

By Observation ...

All variables are declared
White space is ignored

```zpl
program Sample_Stats;
/* Program to compute mu & sigma */
config var n : integer = 100;
region V = [1..n];
procedure Sample_Stats(); -- Entry point
var Sample : [V] float;
mu, sigma: float;
[V]:begin
read(Sample);
mu := +<<Sample/n;
sigma:= sqrt(+<<((Sample-mu)^2)/n);
writeln("Mean: ",mu,"S.D. :", sigma);
end;
```

```
program Sample_Stats;
/* Program to compute mu & sigma */
config var n : integer = 100;
region V = [1..n];
procedure Sample_Stats(); -- Entry point
var Sample : [V] float;
mu, sigma: float;
[V]:begin
read(Sample);
mu := +<<Sample/n;
sigma:= sqrt(+<<((Sample-mu)^2)/n);
writeln("Mean: ",mu,"S.D. :", sigma);
end;
```
ZPL Is Intuitive: Find \( \mu \) and \( \sigma \)

1 program Sample_Stats;
2 config var n : integer = 100;
3 region V = [1..n];
4 procedure Sample_Stats();
5 var Sample : [V] float;
6 mu, sigma: float;
7 [V] begin
8     read(Sample);
9     mu := +<<Sample/n;
10    sigma:= sqrt(+<<((Sample-mu)^2)/n);
11     writeln("Mean: ",mu,"S.D. :", sigma);
12 end;

Convention: Scalars are in lower case; an array’s first letter is capitalized

One Slide of Standard Stuff ...

Data Types: boolean, ubyte, sbyte, char, integer, uinteger, float, double, quad, complex, ...

Unary Operators: +, -, !

Binary Operators: +, -, *, /, ^, %, &, |

Relational Operators: =, !=, <, >, <=, >=

Bit Operators: bnot(),band(),bor(),bxor(),bsl(),bsr()

Assignments: :=, +=, -=, *=, /=, %=, &=, |=

Control Structures: if-then-{elsif}-else,
repeat-until, while-do, for-do, exit, return, continue, halt, begin-end

Jacobi Iteration, The Loop

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

Jacobi Iteration, The Region

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

Jacobi Iteration, The Direction

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

Jacobi Iteration, The Border

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

- ZPL allows arrays to combine with scalars, a convention called "scalar promotion"
  \[
  \text{Temp} := (A@N + A@E + A@W + A@S)/4.0
  \]
  Scalars assume shape of the arrays they're operands with
- Another form is "function promotion"
  \[
  \text{abs(Temp - A)}
  \]
  The (scalar) function is applied to each element of the array
- Programmer-written scalar functions can be promoted, too

Regions: State What, not How

- Most languages define indices operationally by looping
- Regions are index sets of arbitrary size
- Regions and region operators (of, at, in, etc.) replace indexing and simplify programming

Region Calculus

- ZPL's region operators induce a "region calculus"
- Let a dense \( r \)-dimensional region be specified by its upper and lower limit pairs: \( <l_1, u_1>, <l_2, u_2>, ..., <l_r, u_r> \)
- When \( d = (d_1, d_2, ..., d_r) \) and \( R = <l_1, u_1>, <l_2, u_2>, ..., <l_r, u_r> \), then

  \[
  R \text{ at } d = <l_1 + d_1, u_1 + d_1>, <l_2 + d_2, u_2 + d_2>, ..., <l_r + d_r, u_r + d_r>
  \]

  \( d \text{ of } R \) satisfies...
  \[
  \begin{align*}
  <l_{i+1}, u_{i+1}> & \text{ if } d_i > 0 \\
  <l_i', u_i'> & = <l_i, u_i> & \text{ if } d_i = 0 \\
  <l_1 + d_1, l_i - 1> & \text{ if } d_i < 0
  \end{align*}
  \]

  (A more general formulation handles ZPL's more general regions)

Global Operations

- Reduce (<<) and scan (||) are array functionals that perform global operations
- \( <<A \) reduces \( A \) to its sum
  \[
  <<2 4 6 8 = 20
  \]
- \( + || \) are parallel prefixes of \( A \)
  \[
  + || 2 4 6 8 = 2 6 12 20
  \]

  The operators are associative allowing parallel prefix techniques to be used in their evaluation

Reduce and scan apply only over applicable region

\[
[1..i] \text{ first } i := <<A; \text{ sum first } i \text{ elements}
\]

Regions In Computation

- The region \( r \) prefixing a statement gives the indices over which all computation on rank \( r \) arrays is applied
- Regions are scoped, i.e. a region on an inner statement "over-rides" a region on outer stmt
- Regions can be dynamic, i.e. bounds are evaluated on each execution of the statement

Defining Regions Using of

- \( \text{of} \) defines a region adjacent to the given region in the given direction
- \( \text{E} \) of \( C \) defines the region \([8, 2..7] \)
- \( \text{E} \) of \( R \) defines the region \([9, 1..8] \)

Global Operations

- Reduce (<<) and scan (||) are array functionals that perform global operations
- \( <<A \) reduces \( A \) to its sum
- \( + || 2 4 6 8 = 20 \)
- \( + || 2 4 6 8 = 2 6 12 20 \)

The operators are associative allowing parallel prefix techniques to be used in their evaluation

Reduce and scan apply only over applicable region

\[
[1..i] \text{ first } i := <<A; \text{ sum first } i \text{ elements}
\]
Finding The Bounding Box

- Let X and Y be 1D arrays of coordinates such that \((X_i, Y_i)\) is a position in the plane.
- The bounding box uses four reduces:

```plaintext
[B] begin
  rightedge := max <X;  
  topegedge := max <Y;  
  leftedge := min <X;  
  bottomedge := min <Y;  
end;
```

Bounding Box With point Type

- Rather than using arrays of integers, define a type

```plaintext
type point = record
  x : integer;  -- x coordinate
  y : integer;  -- y coordinate
end;
```

8-way Connected Components

The Levaldi morphological operator is the basis for a simple program to find 8-way connected components:

- Assume an array of binary pixels
- Define connectedness 8-ways
- Reduce each component to the lower right corner of its bounding box using morphology:

```
When an isolated pixel is removed, count it
```

ZPL Connected Components

```plaintext
Count := 0;
repeat
  Next := Im & (Im@n | Im@nw | Im@w);
  Next := Next | (Im@w & Im@n & !Im);
  Conn := Im@e | Im@se | Im@s;
  Conn := Im & !Next & !Conn;
  Count += Conn;
  Im := Next;
  smore := |<<Next;
until !smore;
```

Support for Boundaries

- Automatically extends arrays to have borders
- Borders seamlessly participate in computation
- Wrap and reflect assist in computing boundaries

Cannon’s Algorithm

Recall Cannon's Algorithm was claimed to be effective ... it should be programmable in ZPL

```
c11 c12 c13 a11 a12 a13 a14
  c21 c22 c23 a21 a22 a23 a24
  c31 c32 c33 a31 a32 a33 a34
  c41 c42 c43 a41 a42 a43 a44
```

A and B are skewed and conceptually “pass across” the result array \(C\) that's initialized to \(0\). As \(aik\) and \(bkj\) pass over \(cij\), they are multiplied and the result is added into the \(cij\).
Skewing The Arrays

ZPL supports only dense arrays, not skewed arrays or general data structures ... no worries

Performing Skewing Computation

Skewing can be realized by wrapping the first column to the right border, then shifting left

• Assume declarations
  
  \[ region Lop = [1..m,1..n] \]
  \[ direction right = [0,1] \]
  \[ a11 a22 a33 a44 \]
  \[ a12 a23 a34 a41 \]
  \[ a13 a24 a31 a42 \]
  \[ a14 a21 a32 a43 \]

  for \( i := 2 \) to \( n \) do
  
  \[ \text{[right of Lop]} \text{ wrap A}; \]
  \[ \text{[i..m,1..n]} A := A@right; \]
  
  end;

Cannon's Algorithm

Skew A, Skew B, Multiply, Accumulate, Rotate

• Control is said to shatter

Indexi

• ZPL doesn’t need subscripts, but it is still useful to have indices.

  • Indexi is a (compiler created) constant array giving the value of the ith subscript

    \( [1..50] A := 2*Index1; \text{-- A=even nums 2 to 100} \)

  • The “i” must be a number of a legal dimension

    \( [1..n,1..n] \text{Ident} := \text{Index1=Index2}; \text{-- 1s on diag} \)

  • Indexi arrays are logical, they use no storage

  • It is not legal to assign to Indexi

Control-flow Chacteristics

• ZPL has “sequential” control flow, i.e. under most circumstances statements execute one at a time to completion

  \[ \text{fact} := 1; \]
  \[ \text{for } i := 2 \text{ to } n \text{ do fact }*:=i; \text{ end; } -- n! \]

  • Consider the affect of replacing a scalar with an array in control predicates

    \[ \text{Fact} := 1; \]
    \[ \text{for } i := 2 \text{ to } N \text{ do Fact }*:=i; \text{ end; } -- N! \]
    \[ N = 3 \text{! } 4 \text{! } 5 \text{ implies Fact }* = 6 \text{! } 24 \text{! } 120 \]

  • Control is said to shatter
Conditons on Shattered Control Flow

- Any use of an array in a control flow expression results in shattering
  - while T>0 do ...
  - repeat ... until S=0;
  - if B != C then ... else ...;
  - for i := A to B do ...
- A sequence of statements will be executed for each index in the applicable region
- The order of execution is unspecified

Restrictions: No assignment to scalars; instances of @-modified variables must be identical; no wrap, reflect, flooding, permute, reduction, scan or other "array operations"

Applications of Shattered Control Flow

- Use shattered control flow to adapt to different situations
  - -- Take square root, preserve sign
    if X>=0 then Y := sqrt(X);
    else Y := -sqrt(-X);
  - end;
- Shattering saves writing procedures for promotion, i.e. a shattered statement acts like an anonymous promoted function
- Most applications of shattering can be realized by masking

Flooding Abstraction

- Flooding is a ZPL abstraction for replication
- Fortran 90 has spread, MATLAB has "Tony's Trick"

\[
\begin{align*}
\text{ZPL} & \quad [1..n,*] \ F := \gg[1..n,1] \ A; \\
\text{MATLAB} & \quad F = A(:,\text{ones}(1,\text{size}(A,2))); \\
\text{F-90} & \quad F = \text{SPREAD}(A[1,1],\text{DIM}=2,\text{N})
\end{align*}
\]

Flooding Operator

- Flooding uses two regions, the region on the statement and a region following the operator
- One (or more) of the operator region's dimensions must be collapsed, i.e. be a singleton ... replication occurs in this dimension

\[
\begin{align*}
[1..n,1..n] \ Col := \gg[1..n,k] \ A; \\
[1..n,1..n] \ Row := \gg[k,1..n] \ A;
\end{align*}
\]

Matrix Product

- SUMMA: Iteratively flood a column of A and a row of B into temporary matrices, multiply & accumulate in C

\[
\begin{align*}
[1..n,1..n] \ C := 0.0; & \quad \text{-- Initialize C} \\
[1..n,1..n] \text{ for } k := 1 \text{ to } n \text{ do} \\
[*,1] \ Col := \gg[k,1] \ A; & \quad \text{-- Flood kth col of A} \\
[1,*] \ Row := \gg[k,1] \ B; & \quad \text{-- Flood kth row of B} \\
C := C + \text{Col} \times \text{Row}; & \quad \text{-- Accumulate product} \\
\end{align*}
\]

Indexed Arrays

- ZPL has a second kind of arrays called indexed arrays
- Indexed arrays are similar to arrays in conventional languages:
  - var TABLE : array [1..3,1..100] of integer;
  - name keywd bounds kw type
- Indexed arrays are subscripted: [ i, j]

Indexed arrays are not a source of parallelism
- Use indexed arrays for local tables, building data structures, local serial computation, etc.
Indexed Arrays As Array Elements

- An array of indexed arrays is a common data structure
- The elements of the array are evaluated concurrently, though the computation on each element is sequential

Indexed arrays give an easy parallel implementation for solving independent instances problems

Procedures -- Declarations

- The form of a procedure declaration is
  procedure PName ((Formals) :: Type);
  {Locals}
  Statement;
- Formal parameters are listed with their types
  procedure P(A :: R byte, x :: float) :: float;
- Values are returned by:
  return ... ;
- Formal parameters can be called by-value, the default, or by-reference by prefixing the name with var
  procedure G(var A :: R, n :: integer);

Procedure Factoids

- Formals can be rank defined
  procedure H(var A :: [], m :: ubyte);
- Procedures inherit the region of the call site
  procedure AddLast(A :: [] float) :: float;
  begin
    sum := +<< A;
    return sum
  end;
  for i := 1 to n do
    [i..n] AddLast(A) ...
- Procedures can be recursive
- Use prototypes to specify a procedure header
  prototype H(var A :: [], m :: ubyte);

More Procedural Facts

- Procedures can be declared in any order, but they must at least be prototyped before they are referenced
- A ZPL program begins with a program statement
  program PName;
- There must be a procedure with the identical name as the program; the procedure is the entry point (main)
  procedure PName();
- Notice that global state information is typically defined as global variables rather than as variables "passed in" to each procedure

Vector Quantization

- VQ is a lossy image compression technique
- A code book is constructed on training set
- Use 256 entries to map 2x2 bytes to byte
- Declarations ...
  config var n :: integer = 512;
  region R = [1..n, 1..n];
  type block = array [1..2, 1..2] of ubyte;
  var CB : array [0..255] of block;
  Im : [R] block;
  Coding : [R] ubyte;
  Disto, Distn : [R] float;

A Distance Procedure

- To compute the mean square distance between to blocks, define the function
  procedure dist(b1, b2 :: block) :: float;
  return ((b1[1,1] - b2[1,1])^2 + (b1[1,2] - b2[1,2])^2 + (b1[2,1] - b2[2,1])^2 + (b1[2,2] - b2[2,2])^2)/4.0;
- The dist() function will be applied so the first argument argument is from the code book and the second is from the image
VQ Compression Loop

- Assume code book is input

[R] repeat
  -- Input next image, blocked into Im
  Disto := dist(CB[0],Im); -- Init w/ dist entry 1
  Coding := 0; -- Set coding to 1st
  for i := 1 to 255 do -- Sweep thru code bk
    Distn := dist(CB[i],Im); -- dist to ith entry
    if Disto > Distn then -- Is new dist less?
      Disto := Distn; -- Y, update distance
      Coding := i; -- record the best
    end;
  end;
  -- Output the compressed image in Coding
  until no_more_images;

VQ Observations

- All pixel blocks of an image handled at once
- Iteration sweeps thru, trying code book entries
- dist() is f-promoted in its second parameter
- The Distn > Disto predicate is on arrays implying the if is shattered
- The code book as an indexed array, so it is stored redundantly on each processor

Permutation

- ZPL supports non-local data movement with the permutation operators, <<< gather and >>> scatter
- A reordering array must be provided for each dimension
  
  Let Order = 5 4 3 2 1 and Data = 'ABCDE'
  [1..5] Result := <<<[Order] Data;
  Then Result = 'EDCBA'
- A common operation is transpose:
  [1..n,1..n] AT := <<<[Index2,Index1] A;
- Permutation is ZPL’s most expensive operator

Summary

- ZPL is a new language designed to simplify programming scientific computations
- Most of the language structures have been introduced, but much detail remains ... see the ZPL Programmer’s Guide for specifics
- Techniques for finding a solution have been emphasized so far ... the next topic is techniques for finding fast, parallel solutions