The Basics of ZPL

Like sequential computation with its C programming language and von Neumann model of computation explaining the performance of programs, parallel computation needs a language calibrated to the CTA model. ZPL is the only such language.

ZPL -- A Practical Parallel Language

- ZPL was designed from “1st principles” meaning...
  - ZPL is not an extension of existing language -- it’s new
  - Careful analysis of programming task: XYZ-levels
  - No programming “fads”: functional, OO, “miracle” solutions
  - Search for new ideas that help parallel programmers
  - Focus on “user needs,” e.g. scientific computation

- ZPL emphasizes
  - Performance ... user’s programs run as well or better than message passing
  - Portability ... it runs on any parallel computer or cluster
  - Convenience ... parallel programming is tough, so ZPL is high level
ZPL...

Is an array language -- whole arrays are manipulated with primitive operations

• Requires new thinking strategies --
  • Forget one-operation-a-time scalar programming
  • Think of the computation globally -- make the global logic work efficiently and leave the details to the compiler

• Is parallel, but there are no parallel constructs in the language; the compiler...
  • Finds all concurrency
  • Performs all interprocessor communication
  • Implements all necessary synchronization (almost none)
  • Performs extensive parallel and scalar optimizations

A Sample of ZPL Code

```pascal
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;
end;
```

ZPL is an imperative array language with the usual data types and operators, the familiar statement forms, and a few new concepts added
Is a mix of Pascal, C and new syntax
A Sample of ZPL Code

```zpl
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;
end;
```

New features
- config vars
- region
- direction
- prefixing [ ]
assists with the global view of computation

Jacobi Iteration: How does it work?

```zpl
program Jacobi;
config var n : integer = 512;
    eps : float = 0.00001;

.:=(+[+[+]])/4.0;

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;
end;
```

Think of averaging the 4 nearest neighbors as whole array operations.
Regions: A New Concept

```plaintext
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;
end;
```

Directions: Another New Concept

```plaintext
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;
end;
```
Operations on Regions

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;
end;

Referencing 4 Nearest Neighbors

program Jacobi;
config var n : integer = 512;
eps : float = 0.00001;
@d shifts applicable region in d direction

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;
end;
The “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;
end;

ZPL In Detail ...

ZPL has the usual stuff
- Datatypes: boolean, float, double, quad,
  complex, signed and unsigned integers: byte,
  ubyte, integer, uinteger, char, ...
- Operators:
  • Unary: +, -, !
  • Binary: +, -, *, /, ^, %, &, |
  • Relational: <, <=, =, !=, >=, >=
  • Bit Operations: bnot(), band(), bor(), bxor(), bs1(), bsr()
  • Assignments: :=, +=, -=, *=, /=, %=, &=, |=
- Control Structures: if-then-[elsif]-else, repeat-until,
  while-do, for-do, exit, return, continue, halt, begin-end
ZPL Detail (continued)

- White space ignored
- All statements are terminated by semicolon (;)
- Comments are
  -- to the end of the line
  /* */ all text within pairs including newlines
- All variables must be declared using `var`
- Names are case sensitive
- Programs begin with
  `program <name>;;`
  the procedure with `<name>` is the entry point

ZPL Detail (continued)

- The unary global operation reduction (<<) “reduces” an entire array to a single value using an associative operator: +=, *<<, max<<, min<<, &<<, |<<
- For example, += is summation (∑) and max<< is global maximum
  ```
  err := max<< abs(Temp - A);
  ```
Bounding Box

• Let X, Y be 1-dimensional \( n \) element arrays such that \((x_i, y_i)\) is a position in the plane
• The bounding box is the extreme coordinates in each dimension

\[
\begin{align*}
\text{begin} & \quad \text{right edge : } = \max <\ X; \\
& \quad \text{top edge : } = \max <\ Y; \\
& \quad \text{left edge : } = \min <\ X; \\
& \quad \text{bottom edge : } = \min <\ Y; \\
\text{end}
\end{align*}
\]

Alternative Data Representation

• ZPL allows programmers to define a type
• Rather than using X and Y arrays, define

\[
\begin{align*}
\text{type cartPoint = record} & \quad \text{x : integer; -- x coordinate} \\
& \quad \text{y : integer; -- y coordinate} \\
\text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{var Pts : [1..n] cartPoint; -- an array of points} & \quad \text{right edge : } = \max <\ Pts.x; \\
& \quad \text{top edge : } = \max <\ Pts.y; \\
& \quad \text{left edge : } = \min <\ Pts.x; \\
& \quad \text{bottom edge : } = \min <\ Pts.y;
\end{align*}
\]
ZPL Inherits from C

- ZPL is translated into C
- Mathematical functions come from math.h
- ZPL’s Input and Output follow C conventions and formatting, though the behavior on parallel machines can differ

Configuration variables (config vars) and constants are a list of command line assignable identifiers with specified defaults ... config const cannot be reset

```c
config const prob_size : integer = 64;
```

Mean and Standard Deviation ...

Find \( \mu \) and \( \sigma \) for array of Sample values

```c
program Sample_Stats;
    config var n : integer = 100;
    region   V = [1..n];

procedure Sample_Stats();
    var Sample : [V] float;
    mu,sigma: float;
    [V] begin
        read(Sample);
        mu := +<Sample/n;
        sigma := sqrt(+<(Sample-mu)^2)/n);
        write ("Mean: ", mu,"S.D. :", sigma);
    end;
```

\[
\mu = \frac{\sum Sample_i}{n}
\]

\[
\sigma = \sqrt{\frac{\sum (Sample_i - \mu)^2}{n}}
\]

Basically, a direct translation into imperative form