

# ZPL - Parallel Programming Language



---

Barzan Mozafari

Amit Agarwal

Nikolay Laptev

Narendra Gayam



# Outline

---

- Introduction to Language
- Strengths and Salient features
- Demo Programs
- Criticism/Weaknesses



# Parallelism Approaches

---

- Parallelizing compilers
- Parallelizing languages
- Parallelizing libraries



# Parallelism Challenges

---

- Concurrency
- Data distribution
- Communication
- Load balancing
- Implementation and debugging



# Parallel Programming Evaluation

---

- Performance
- Clarity
- Portability
- Generality
- Performance Model



# Syntax

---

- Based on Modula-2 (or Pascal)
- Why?
  - To enforce C and Fortran programmers rethink
  - Lack of features that conflict w/ paralellism
    - Pointers
    - Scalar array indexing
    - Common blocks
  - Both readable and intuitive



# Data types

---

- Types:
  - Integers of varying size
  - Floating point
  - Homogeneous arrays types
  - Heterogeneous record types



# Constants, variables

---

**type**

```
age = shortint;  
coord = record  
    x: integer;  
    y: integer;  
end;
```

**constant**

```
pi: double = 3.14159265;  
tabsize: integer = 1000;  
maxage: age = 128;
```

**var** done: **boolean**;

```
length: integer;  
name: string;  
origin: coord;  
table: array [1..tabsize] of complex;
```



# Configuration variables

---

- **Definition:**
  - Constant whose values can be deferred to the beginning of the execution but cannot change thereafter (**loadtime constant**).
- **Compiler:** treats them as a constant of unknown value during optimization
- **Example:**

`config var`

```
n: integer = 100;           -- a sample problem size
verbose: boolean = true;   -- use to control output

logn: integer = lg2(n);    -- log of the problem size
nsq: integer = n^2;        -- the problem size squared
npi: double = pi*n;       -- n times the constant pi
```



# Scalar operators

---

## Arithmetic Operators

+ addition  
- subtraction  
\* multiplication  
/ division  
% modulus  
^ exponentiation

## Logical Operators

& and  
| or  
! not

## Relational Operators

= equality  
!= inequality  
< less than  
> greater than  
<= less than/equal  
>= greater than/equal

## Bitwise Operators

band and  
bor or  
bnot complement  
bxor xor

## Assignment Operators

:= standard  
+= accumulative  
-= subtractive  
\*= multiplicative  
\= divisive  
&= conjunctive  
|= disjunctive



# Syntactic sugar

---

- Blank array references
  - `Table[] = 0`
  - To encourage array-based thinking (avoid trivial loops)



# Procedures

---

- Exactly resembling Modula-2 counterparts
- Can be recursive
- Allows external code
  - Using **extern prototype**
  - **Opaque:** Omitted or partially specified types
    - Cannot be modified or be operated on, only pass them around



# Regions

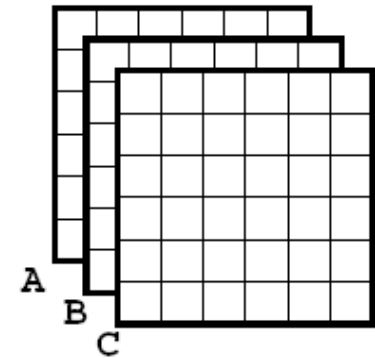
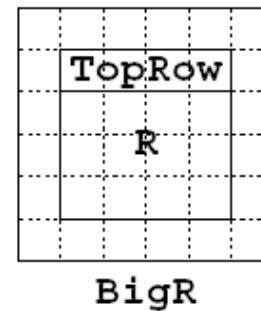
---

- Definition: An index set in a coordinate space of arbitrary dimension
- Naturally, regular (=rectangular)
- Similar to traditional array bounds (reflected in syntax too!)
- Singleton dimension  $[1,1..n]$  instead of  $[1..1,1..n]$

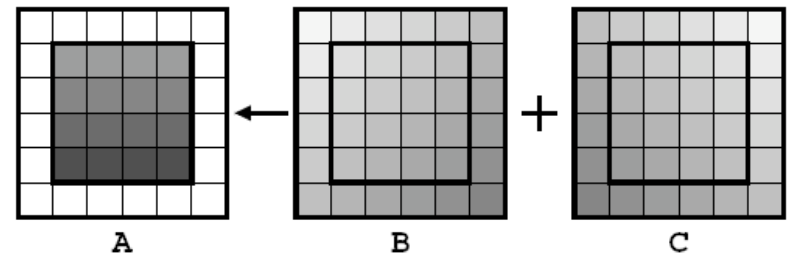
$$[1..m, 1..n] = \{(1,1), (1,2), \dots, (1,n), (2,1), \dots, (m,n)\}$$

# Region example

```
region R = [1..m, 1..n];  
  TopRow = [1, 1..n];  
  BigR = [0..m+1, 0..n+1];
```



```
var A, B, C: [BigR] integer;
```

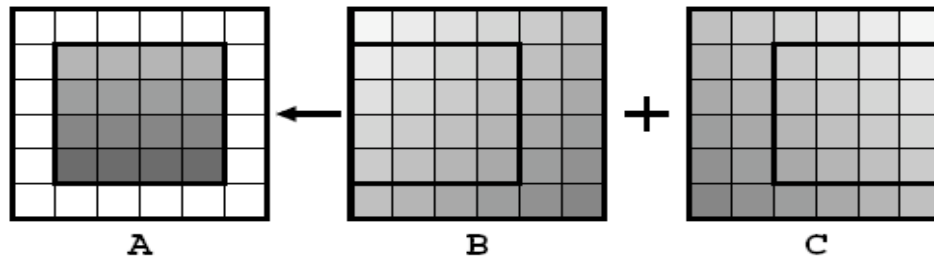


```
[R] A := B + C;
```

# Directions

- Special vectors, e.g. cardinal directions
- @ operator

```
direction north = [-1, 0];  
            south = [ 1, 0];  
            east  = [ 0, 1];  
            west  = [ 0,-1];
```



```
[R] A := B@west + C@east;
```

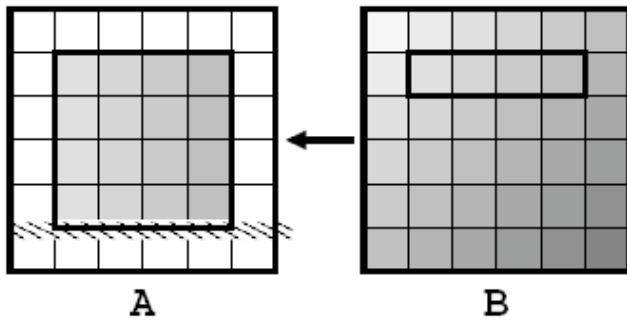


# Array operators

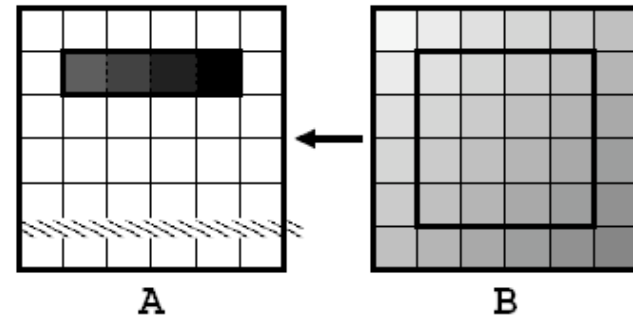
---

- @ operator
- Flood operator: >>
- Region operators:
  - At
  - Of
  - In
  - By

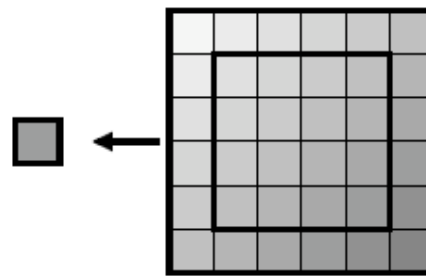
# Flood operator



`[R] A := >> [TopRow] B;`



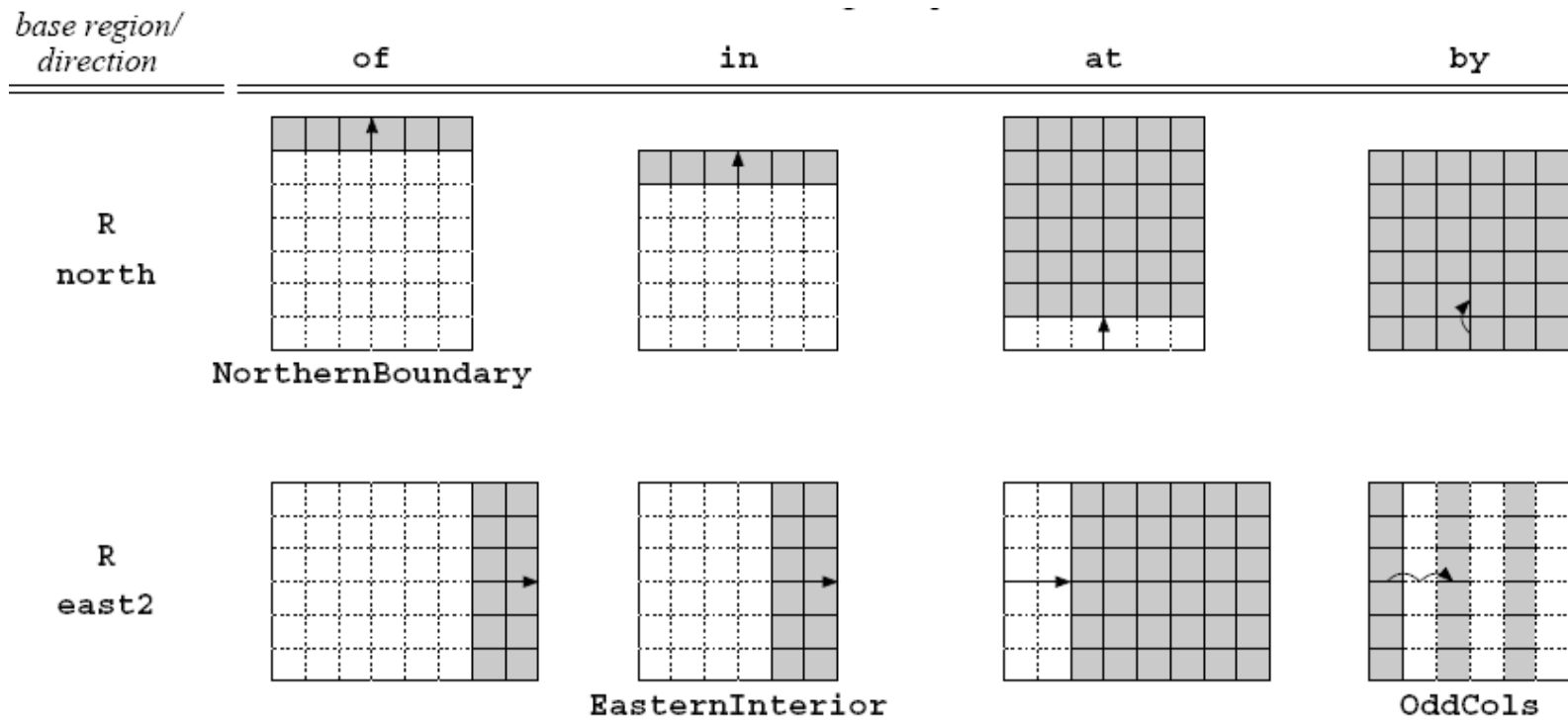
`[TopRow] A := +<< [R] B;`



biggest B

`[R] biggest := max<< B`

# Region operators (I)





## Region operators (II)

---

$$\delta \text{ of } (l, h, s, a) \Rightarrow \begin{cases} (l + \delta, l - 1, s, a) & \text{if } \delta < 0 \\ (l, h, s, a) & \text{if } \delta = 0 \\ (h + 1, h + \delta, s, a) & \text{if } \delta > 0 \end{cases}$$

$$\delta \text{ in } (l, h, s, a) \Rightarrow \begin{cases} (l, l - (\delta + 1), s, a) & \text{if } \delta < 0 \\ (l, h, s, a) & \text{if } \delta = 0 \\ (h - (\delta - 1), h, s, a) & \text{if } \delta > 0 \end{cases}$$



# Region operators (III)

---

$$(l, h, s, a) \text{ at } \delta \Rightarrow (l + \delta, h + \delta, s, a + \delta)$$

$$(l, h, s, a) \text{ by } \delta \Rightarrow \begin{cases} (l, h, |\delta| \cdot s, (h - ((h - a) \bmod s)) + (|\delta| \cdot s)) & \text{if } \delta < 0 \\ (l, h, s, a) & \text{if } \delta = 0 \\ (l, h, |\delta| \cdot s, (l + ((a - l) \bmod s)) + (|\delta| \cdot s)) & \text{if } \delta > 0 \end{cases}$$



# Outline

---

- Introduction to the Language
- **Strengths and Salient Features**
- Demo Programs
- Criticism/Weaknesses



# Desirable traits of a parallel programming language

---

- Correctness – Cannot be compromised for speed.
  - Correct results irrespective of the no. of processors and their layout.
- Ease of Programming
  - Intuitive and easy to learn and understand
  - High level constructs for expressing parallelism
  - Easy to debug - Syntactically identifiable parallelism constructs
- Speedup



# P-dependent vs. P-independent

---

- P-dependent - behavior dependent on the number or arrangement of processors.
- Extremely difficult to locate problems specific to a particular number and layout of processors
  - NAS CG MPI benchmark failed only when run on more than 512 processors. 10 years before the bug was caught.
  - Compromises programmer productivity by distracting them from the main goal of improving performance.



## P-dependent vs. p-independent...

---

- ZPL believes in machine independence
  - Constructs are largely p-independent – compiler finds and implements parallelism.
  - Much easier to debug – Example race conditions and deadlocks are absent.



# P-dependent vs. p-independent...

---

- But performance is critical to parallel languages.
  - Implicit parallelism for most part –the compiler finds and implements parallelism in sequential programs.
  - Small set of p-dependent abstractions – provide the programmer control on performance
  - Conscious choice of performing low level optimizations using these constructs.
  - P-independent constructs for explicit data distribution and layout.



# Syntactically identifiable communication

---

- Inter-processor communication is the main performance bottleneck
- The portions of code inducing communication should be easily distinguishable.
  - Allows users to focus on relevant portions of the code only, for performance improvement



# Syntactically identifiable communication...

---

- MPI, SHMEM
  - It's only communication – Explicit communication specified by the programmer using low level library routines.
  - Very little abstraction – originally meant for library developers.
- Titanium, UPC
  - Global address space makes programming easier.
  - Communication is not visible to the programmer.
- High Level languages – HPF, OpenMP
  - High level directives to suggest parallel execution strategies.
  - Communication is not syntactically identifiable
  - Difficult to optimize – OpenMP now runs only on Shared Memory systems.



# Syntactically identifiable communication...

---

- ZPL makes communication syntactically identifiable
  - Communication between processors induced by a small number of operators
  - Operators also indicate the kind of communication involved.
  - Though communication implemented by the compiler, easy to tell where and what are the communications.



# Parallel Programming model

---

- Simple programming model
  - Data is either distributed or replicated.
  - Parallel arrays distributed over processors.
  - Variables and regular indexed arrays are replicated across processors.
  - Implicit consistency guarantee through an array of static type checking rules.

# Salient features

- ZPL is an array language.
  - Array Generalization
  - $[R] A = B + C$  ; Relieves the programmer from writing tedious loops and error prone index calculations.
  - Enables the processor to identify and implement parallelism.

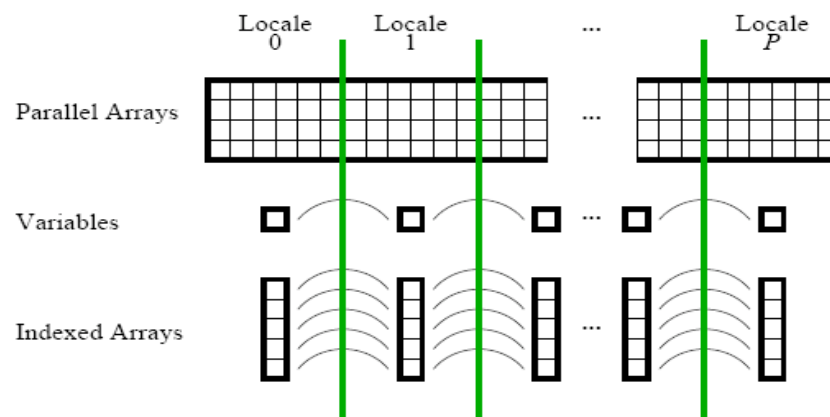


Figure 2.1: An Illustration of ZPL's Parallel Programming Model



# WYSIWYG Parallel Execution

---

- Can program in ZPL without considering parallel execution.
- Characteristics preferred for parallel execution also promote high performance on sequential computers too (caches, multi-issue instruction execution).



# WYSIWYG Parallel Execution...

---

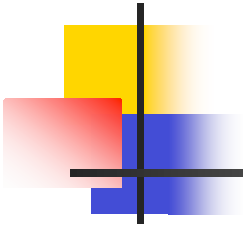
- Concurrency realized by distributing arrays and executing operations in parallel.
- Speed comparable to hand coded C programs.
- Works well for both Shared Memory and Distributed memory parallel computers.



# Outline

---

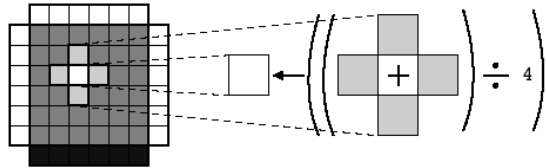
- Introduction to the Language
- Strengths and Salient Features
- **Demo Programs**
- Criticism/Weaknesses



# HelloWorld

---

```
program hello;  
  
procedure hello();  
begin  
  writeln("Hello, world!");  
end;
```



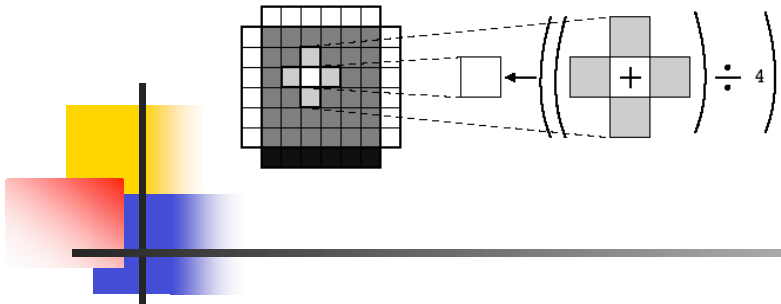
# Jacobi

```

1 program jacobi;
2
3 config var n: integer = 100;           -- the problem size
4         epsilon: double = 0.00001;   -- the convergence condition
5         verbose: boolean = false;    -- verbose output?
6
7 region R = [1..n, 1..n];              -- the computation indices
8         BigR = [0..n+1, 0..n+1];     -- the declaration indices
9
10 var A: [BigR] double;                 -- the main data values
11     New: [R] double;                  -- the new iteration's values
12     delta: double;                   -- change between iterations
13
14 direction north = [-1, 0];           -- the four cardinal directions
15         south = [ 1, 0];
16         east  = [ 0, 1];
17         west  = [ 0, -1];
18

```

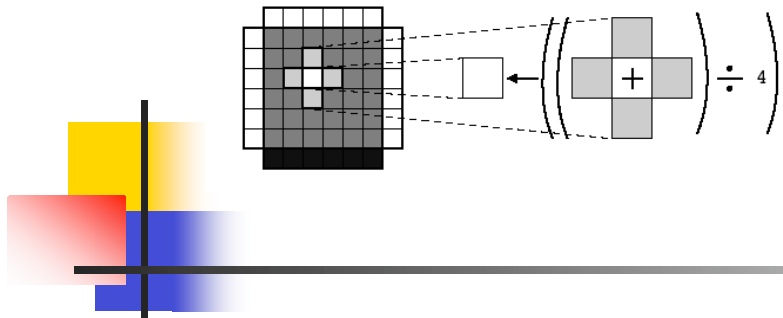
Variable Declaration



# Jacobi(continued)

```
19 procedure init(var X: [ , ] double);  -- array initialization routine
20 begin
21   X := 0;
22   [north of "] X := 0.0;
23   [south of "] X := 1.0;
24   [east of "] X := 0.0;
25   [west of "] X := 0.0;
26 end;
27
```

Initialization



# Jacobi(continued)

```

28 procedure jacobi();                                -- the main entry point
29 [R] begin
30     init(A);
31
32     repeat
33         New := (A@north + A@south +                -- five-point stencil on A
34                A@east  + A@west)/4.0;
35
36         delta := max<< fabs(A - New);             -- find maximum change
37
38         A := New;                                  -- copy back
39     until (delta < epsilon);                       -- continue while change is big
40
41     if (verbose) then
42         writeln("A:\n", A);                       -- write data if desired
43     end;
44
45     writeln("delta: %le": delta);                 -- always write delta
46 end;
```

Main Computation



# Outline

---

- Introduction to the Language
- Strengths and Salient Features
- Demo Programs
- **Criticism/Weaknesses**



# Criticism

---

- ZPL could afford to provide support for arrays at the exclusion of other data structures. As a consequence, ZPL is not ideally suited for solving certain type of dynamic and irregular problems.

ZPL's region concept does not support distributed sets, graphs, and hash tables.



# Criticism

---

- Exchanging of indexes between processors greatly affects the performance.
- Determining Fluff size or how much Fluff is required is not clear enough. And when it can't be determined statically then we have to dynamically resize the array. This degrades the performance.



# Criticism

---

- ZPL being a data parallel language cannot handle certain expressions :
  - Asynchronous producer-consumer relationships for enhanced load balancing are still difficult to express
  - The 2D FFT problem in which the series of iterations are executed in multiple independent pipelines in a round-robin manner. If suppose the time needed for the computation to proceed through pipeline is dependent on the data. ZPL would result in a possibly inefficient use of the resources.



# Criticism

---

- ZPL is data parallel but not task parallel.
- ZPL supports at most a single level of data parallelism.
- Limitations of ZPL led to the philosophical foundation of the Chapel language.



# Criticism

---

- Chapel supports multiple levels of parallelism for both task-parallel and data-parallel algorithms.
- Chapel provides support for distributed sets, graphs, and hash tables.