ZPL - Parallel Programming Language

Barzan Mozafari
Amit Agarwal
Nikolay Laptev
Narendra Gayam
Outline

- Introduction to the 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 with parallelism
    - Pointers
    - Scalar indexing of parallel arrays
    - Common blocks
  - Both readable and intuitive
Data types

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

```plaintext
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

<table>
<thead>
<tr>
<th>Arithmetic Operators</th>
<th>Relational Operators</th>
<th>Assignment Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>+  addition</td>
<td>=  equality</td>
<td>:=  standard</td>
</tr>
<tr>
<td>-  subtraction</td>
<td>!=  inequality</td>
<td>+=  accumulative</td>
</tr>
<tr>
<td>*  multiplication</td>
<td>&lt;   less than</td>
<td>-=  subtractive</td>
</tr>
<tr>
<td>/  division</td>
<td>&gt;   greater than</td>
<td>*=  multiplicative</td>
</tr>
<tr>
<td>%  modulus</td>
<td>&lt;=  less than/equal</td>
<td>=  divisive</td>
</tr>
<tr>
<td>^  exponentiation</td>
<td>&gt;=  greater than/equal</td>
<td>&amp;=  conjunctive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical Operators</th>
<th>Bitwise Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;  and</td>
<td>band and</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td>!  not</td>
<td>bnot complement</td>
</tr>
<tr>
<td></td>
<td>bxor xor</td>
</tr>
</tbody>
</table>
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 \([1..1,1..n]\)

\[
[1..m, 1..n] = \{(1,1),(1,2),\ldots,(1,n),(2,1),\ldots,(m,n)\}
\]
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 and Reduce operator

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

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

[R] biggest := max<< B
Region operators (I)

<table>
<thead>
<tr>
<th>base region/direction</th>
<th>of</th>
<th>in</th>
<th>at</th>
<th>by</th>
</tr>
</thead>
<tbody>
<tr>
<td>R north</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NorthernBoundary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R east2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EasternInterior</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OddCols</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)

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

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

- Introduction to the Language
- **Strengths and Salient Features**
- Demo Programs
- Criticism/Weaknesses
Desirable traits of a parallel language

- Correctness – **cannot** be compromised for speed.
  - Correct results irrespective of the no. of processors and their layout.

- Speedup
  - Ideally linear in number of processors.

- Ease of Programming, Expressiveness
  - Intuitive and easy to learn and understand
  - High level constructs for expressing parallelism
  - Easy to debug - Syntactically identifiable parallelism constructs

- Portability
ZPL’s Parallel Programming model

- ZPL is an array language.
  - Array Generalization for most constructs
  - \[ R \] \( A = B + C \text{@east} \); Relieves the programmer from writing tedious loops and error prone index calculations.
  - Enables the processor to identify and implement parallelism.

![Illustration of ZPL's Parallel Programming Model](image)

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

- Implicit Parallelism though parallel execution of associative and commutative operators on arrays.
  - Parallel arrays distributed evenly over processors.
  - Same indices go to the same processor
  - Variables and regular indexed arrays are replicated across processors.

- Excellent sequential implementation too (caches, multi-issue instruction execution).
  - Comparable to hand written C code.
ZPL’s Parallel Programming model

- Statements involving scalars executed on all processors.

- Implicit consistency guarantee through an array of static type checking rules.
  - Cannot assign a parallel array value to a scalar
  - Conditionals involving parallel arrays cannot have scalars.
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.
ZPL believes in machine independence

- Constructs are largely p-independent. Compiler handles machine specific implementation details.
- Much easier to code and debug – Example race conditions and deadlocks are absent.
P-dependent vs. p-independent... 

- But sometimes, a low level control may help improve performance.
  - Small set of p-dependent abstractions – provide the programmer control on performance
    - Free Scalars and Grid dimensions
  - 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
  - High latency of “off chip” data accesses
  - Often requires synchronization
- 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.
  - But makes communication invisible.
  - Cannot tell between local and remote accesses and hence the cost involved.
Syntactically identifiable communication...

- ZPL makes communication syntactically identifiable – Let the programmer know what are they getting into
  - Communication between processors induced only by a set of operators
  - Operators also indicate the kind of communication involved - WYSIWYG.
  - Though communication implemented by the compiler, easy to tell where and what are the communications.

[R] A + B – No communication
[R] A + B@east - @ induces communication
[R] A + B#[c..d] - # (remap) induces communication
WYSIWYG Parallel Execution

A unique feature of the language, and one of its most important contributions.

- Sure, the concurrency is implicit and implemented by the compiler. But the let the programmer know the cost.
- Enables programmers to accurately evaluate the quality of their programs in terms of performance.
Every parallel operator has a cost and the programmer knows exactly how much the cost is.

Table 1. Sample WYSIWYG Model Information. Work is the amount of computation for the operator measured as implemented in C; $P$ is number of processors. The model is more refined than suggested here.

<table>
<thead>
<tr>
<th>Syntactic Cue</th>
<th>Example</th>
<th>Parallelism ($P$)</th>
<th>Communication Cost</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>[R] array ops</td>
<td>[R] ... A+B ...</td>
<td>full; work/$P$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@ array transl.</td>
<td>... A@east ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; reduction</td>
<td>... +&lt;&lt;A ...</td>
<td>work/$P$ + log $P$</td>
<td>1 point-to-point</td>
<td>xmit “surface” only</td>
</tr>
<tr>
<td>&lt;&lt; partial red</td>
<td>... +&lt;&lt;[ ] A ...</td>
<td>work/$P$ + log $P$</td>
<td>2log $P$ point-to-point</td>
<td>fan-in/out trees</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scan</td>
<td>... +</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; flood</td>
<td>... &gt;&gt; [ ] A ...</td>
<td></td>
<td>2log $P$ point-to-point</td>
<td>data not replicated</td>
</tr>
<tr>
<td># remap</td>
<td>... A#[I1, I2] ...</td>
<td></td>
<td>2 all-to-all, potentially</td>
<td>general data reorg.</td>
</tr>
</tbody>
</table>
Using the WYSIWYG model

- Programmers use the WYSIWYG model in making the right choices during implementation.

Compute - A[a..b] + B[c..d]

- Naïve implementation
  - Remap – [a..b] A + B#[c..d] -- Very expensive
  - Say you know c = a + 1, and d = b + 1,
    - A better implementation would be:
    - [a..b] A + B@east; -- Less expensive
Portability

- For a parallel program, portability is not just about being able to run the program on different architectures.
- We want the programs to perform well on all architectures.
  - What good is a program, if it is specific to a particular hardware and has to be rewritten to take advantage of newer, better hardware.
- Programs should minimize attempts to exploit the characteristics of underlying architecture.
  - Let the compiler do this job.
- ZPL works well for both Shared Memory and Distributed memory parallel computers.
Speedup

- Speedup comparable or better than carefully hand crafted MPI code.
Expressiveness – Code size

- High level constructs and array generalizations lead to compact and elegant programs.
Outline

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

- HelloWorld
- Jacobi Iteration
  - Solves Laplace’s equation
HelloWorld

program hello;

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

config var n: integer = 100;          -- the problem size
    epsilon: double = 0.00001;        -- the convergence condition
    verbose: boolean = false;         -- verbose output?

region R = [1..n, 1..n];                -- the computation indices
    BigR = [0..n+1, 0..n+1];         -- the declaration indices

var A: [BigR] double;                  -- the main data values
    New: [R] double;                -- the new iteration’s values
    delta: double;                  -- change between iterations

direction north = [-1, 0];            -- the four cardinal directions
    south = [ 1, 0];
    east = [ 0, 1];
    west = [ 0,-1];

Variable Declaration
procedure init(var X: [ , ] double);   -- array initialization routine
begin
  X := 0;
  [north of "] X := 0.0;
  [south of "] X := 1.0;
  [east of "] X := 0.0;
  [west of "] X := 0.0;
end;
procedure jacobi(); -- the main entry point
begin
  init(A);

  repeat
    New := (A@north + A@south + -- five-point stencil on A
           A@east + A@west)/4.0;

    delta := maxf< fabs(A - New); -- find maximum change

    A := New; -- copy back
    until (delta < epsilon); -- continue while change is big

    if (verbose) then
      writeln("A: \n", A); -- write data if desired
    end;

    writeln("delta: %le": delta); -- always write delta
  end;

Main Computation
Outline

- Introduction to the Language
- Strengths and Salient Features
- Demo Programs
- Criticism/Weaknesses
Limited DS support

- 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.
Insufficient expressiveness

- 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.
Remap and Fluff size effect

- 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.
Data vs. Task parallelism

- 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.
Lacking Chapel’s extensions!

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