ZPL and Other Global View Languages

Alexandre David
1.2.05
adavid@cs.aau.dk

Introduction

- So far:
  - libraries in C for threads & message passing
  - only libraries, same base language, no syntactic support for parallelism (omp special)
- High-level parallel language
  - see the whole computation
  - implicit parallelism
  - ZPL is one example, interesting for the benefits at the concept level

ZPL

- Focus on arrays & their manipulations.
- Provides implicit parallelism.
  - Generated threads, communication, sync.
- Goal: parallelism & parallel performance, including the communication cost, without low-level code.
- Example:
  [1..n] count:=+<<(array==3);

Basics

- Array language – arrays as units.
  - A+=1; – updates done logically in parallel.
- Regions: computations on partial arrays
  - [1..n] A+=1;
[1..n/2] A+=1;
  - Several dimensions possible, e.g., [1..8, 1..8]
  - Implicit reference of sub-arrays
[1..m, 1..m] E:=1/B;
works if B “larger” array than E.

Regions

- Limit case: one element.
  - [x,y] D:=sqrt(2);
- Used to declare sizes of arrays.
  - var B, C : [1..m,1..n] float;
- Named regions.
  - region R=[1..m,1..n];
    var B,C : [R] float;
    [R] B:=2*C+D;
- Scope: next statement or block of statements.

Primitive Types

Byte Types	2-Byte Types	4-Byte Types	8-Byte Types	16-Byte Types
`boolean`				
`sbyte`	`shortint`	`integer`	`longint`	
`ubyte`	`ushortint`	`uinteger`	`ulongint`	
		`float`	`double`	`quad`
		`complex`	`dcomplex`	`qcomplex`

The prefix 'u' indicates that the representation is unsigned, giving it an additional bit of precision. The `quad` type is available only if it is available in C on the target architecture; otherwise it defaults to `double`. A *k*-byte complex type uses *k* bytes for the real and *k* bytes for the imaginary parts of the number.

Lesson: Specialized types for numerical computations.

Control-Flow Statements

ZPL Control-Flow Statements

```
if logical-expression then statements {else statements} end;
for var := low to high {by step} do statements end;
while logical-expression do statements end;
repeat statements until logical-expression;
return {expression};
begin statements end;
```

Text in braces is optional; text in italics must be replaced by program constructs of the indicted kind.

Array Computation

- Operators applied element-wise on corresponding elements of the arrays.
  [R] TW:=(TW & NN=2) | (NN=3);
- Operators different than the ones in C.

- Lesson: High-level operators suited for parallelism.

Operators

<table>
<tr><th>Datatype</th><th>Operators</th></tr>
<tr><td>Numeric</td><td>+ (unary), – (unary), +, –, *, /, ^, % (modulus)</td></tr>
<tr><td>Logical</td><td>!, &, |</td></tr>
<tr><td>Relational</td><td>=, !=, <, >, <=, >=</td></tr>
<tr><td>Bit-wise</td><td>bnot(a), band(a,b), bor(a,b), bxor(a,b),
bsl(s,a) (shift a's bits s places left, fill with 0s),
bsr(s,a) (shift a's bits right s places, fill with 0s)</td></tr>
</table>

Exponentiation (^) is optimized to multiplication for small powers, for example, 2, but generally compiles to a call on C's `pow()` function.
The operator assignments recognized are: `+=, -=, *=, /=, %=, &=, |=`

@-translation

- Shift indices on operations – otherwise very boring operations only.
  - direction left=[-1]; right=[1];
    declares directions for references
  - [2..n-1] A:=(A+A@left+A@right)/3;
    translates the indices according to the directions.
  - Example:
    direction nw=[-1,-1]; no=[-1,0]; ne[-1,1]; …
    TW@nw+TW@no+TW@ne+TW@we... gives the number of neighbors relative to TW current element.

Reduce

- op<<A with an associative & commutative operator.
  - [2..n-1] total=+<<A;
  - [R] biggest := max<<B;
  - [R] span:=(max<<A)-(min<<A)+1;

- Lesson: Provide useful high-level operators in a way that can be exploited for parallelism.

Conway's Game of Life

- Start with an initial configuration = generation 0.
- Rules between every generation:
  - An organism survives if it has 2 or 3 neighbors.
  - An organism is born at a free position if it has 3 neighbors.
  - All other organisms die.
- Coding: The world array TW, use @-translation to read neighbors.

Conway's Game of Life

```
program Life;
config const n : integer = 50;

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

var
  TW:[BigR]     boolean = 0;          -- The World
  NN:[R]        integer;              -- Number of Neighbors

direction
  nw=[-1, -1]; no=[-1, 0]; ne=[-1, 1];
  we=[ 0, -1];             ea=[ 0, 1];
  sw=[ 1, -1]; so=[ 1, 0]; se=[ 1, 1];

procedure Life();
begin
  --Initialize the world
[R] repeat
  NN:=TW@nw+TW@no+TW@ne+
      TW@we+      TW@ea+
      TW@sw+TW@so+TW@se;
  TW:=(TW & NN = 2) |(NN = 3);
until !(|<< TW);
end;
```

Arrays declared logically but the compiler does not have to really create them.

No race condition problem.

Lessons

- Simple problem, simple program.
- Concise & clear.
  - Manipulate entire arrays at the same time.
  - Regions and directions.
  - Implicit parallelism comes from array operations.

Distinguishing Features
compared to other array languages

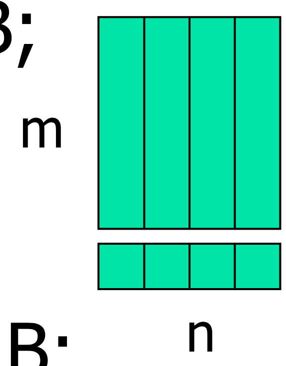
- Regions and @ operator.
  - Restrictions to enforce programming discipline & distinguish expensive operations.
    - No transpose possible with only regions & @.
    - Cost distinction between transpose & copy.
  - Note: typos in transpose example.
- Removal of very general operators with non defined costs.
- Restriction on ranks of arrays.

Manipulating Arrays of Different Ranks

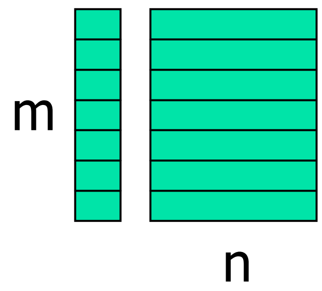
- Regions define dimensions, number of elements, the indices, and the allocation.
- Operators between arrays of the same ranks.
  - Use larger rank if mismatch (with collapsed dimensions).
  - Replicate elements – flood operator. Elements are logically replicated but not necessarily really copied.

Partial Reduce

- Partial reduce on some dimensions.
  - with regions.
  - Example: [1,1..n] C:=+<< [1..m, 1..n] B;



  - Example: [1..m, 1] D:=*<< [1..m, 1..n] B;



  - Example: [1,1,1..n] G:= max << [1,1..m,1..n] (min<<[1..p,1..m,1..n F);
  - Lesson: high-level parallelizable operators.

Flooding

- Way to expand dimensions.
- Inverse of partial reduce.
  - [1..m,1..n] B:=>>[1,1..n] C;
  - [1..m,1..n] C:=>>[1..m,1] D;
  - Fills the missing dimension by copies.
- Principle:
  - Element-wise operators need the same dimensions.
  - Logical copies.

Matrix Multiplication

- Usual sequential language:

```
for(i=0; i<m; i++)
  for(j=0; j<p; j++) {
    C[i,j]=0;
    for(k=0; k<n; k++)
      C[i,j] += A[i,k]*B[k,j];
  }
```

C[m*p]=A[m*n]*B[n*p]

- Simple but not suited for *parallel* product.

Matrix Multiplication

- Considering parallel element-wise multiplications, we can flood the input matrices, do the multiplications, and accumulate.

$$C_{1,1} = A_{1,1} * B_{1,1} + A_{1,2} * B_{2,1} + A_{1,3} * B_{3,1}$$
$$C_{2,1} = A_{2,1} * B_{1,1} + A_{2,2} * B_{2,1} + A_{2,3} * B_{3,1}$$
$$C_{3,1} = A_{3,1} * B_{1,1} + A_{3,2} * B_{2,1} + A_{3,3} * B_{3,1}$$

$$C_{1,1} = A_{1,1} * B_{1,1} + A_{1,2} * B_{2,1} + A_{1,3} * B_{3,1}$$
$$C_{1,2} = A_{1,1} * B_{1,2} + A_{1,2} * B_{2,2} + A_{1,3} * B_{3,2}$$
$$C_{1,3} = A_{1,1} * B_{1,3} + A_{1,3} * B_{2,2} + A_{1,3} * B_{3,3}$$

ZPL Matrix Multiplication

```
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;
    k     :              integer;

procedure MM();
[1..m, 1..p] begin
                 C:=0;
                 for k:=1 to n do
    [1..m, *]      Col:=>> [1..m, k] A;
    [*, 1..p]      Row:=>> [k, 1..p] B;
                   C+=Col*Row;
                 end;
             end;
```

Reordering Data

- Explicit communication cost.
- Index arrays
  - predefined arrays Index1, Index2, …
    (indices on i dimension flooded on the others)
  - Use: [1..n,1..n] Diag:=Index1=Index2;
- Remap operator (#)
  - gather: B=A#[P]; -- pick elements of A in order defined by indices in P
  - scatter: C#[P]=A; -- reverse
  - Ex: [1..n, 1..m] Btransp:=B#[Index2,Index1];
  - Lesson: higher-order operators available

Parallel Execution of ZPL

- Based on the array language features.
- The compiler generates loop nests, adds communication, reduce, …
- Optimizations
  - combine loop nests – reduce memory
  - combine communication – reduce interaction
  - overlap communication & computation
  - efficient flood arrays
  - efficient index arrays
- Lesson: Force to think using certain language constructs that exhibit parallelism. The compiler does the rest.

Performance Model

ZPL's performance model specifications for worst-case behavior; the actual performance is influenced by *n*, *P*, process arrangement, and compiler optimizations, in addition to the physical features of the computer.

Syntactic Cue	Example	Parallelism (*P*)	Communication Cost	Remarks
`[R]` *array ops*	`[R] ... A+B ...`	full; work/*P*	–	
`@` *array transl.*	`... A@east ...`	–	1 point-to-point	xmit "surface" only
`<<` *reduction*	`... +<<A ...`	work/*P* + log *P*	2log *P* point-to-point	fan-in/out trees
`<<` *partial red*	`... +<<[ ] A ...`	work/*P* + log *P*	log *P* point-to-point	
`||` *scan*	`... +|| ...`	work/*P* + log *P*	2log *P* point-to-point	parallel prefix trees
`>>` *flood*	`... >>[ ] A...`	–	multicast in dimension	data not replicated
`#` *remap*	`... A# [I1,I2] ...`	–	2 all-to-all, potentially	general data reorg.

Cost model with the language.
Easy to identify costs.

Communication Cost

- @: λ delay
- Local computation
- Reduce: 2λ log *P*

```
procedure Life();
begin
  --Initialize the world
[R] repeat
  NN:=TW@nw+TW@no+TW@ne+
      TW@we+      TW@ea+
      TW@sw+TW@so+TW@se;
  TW:=(TW & NN = 2) |(NN = 3);
until !(|<< TW);
end;
```

Communication Cost

- SUMMA:
```
[1..m, 1..p] begin
  C:=0;
  for k:=1 to n do
    C+=(>>[1..m,k] A) * (>>[k,1..p] B);
  end;
end;
```
- C=0: perfectly parallel
  (√p*√p grid) flood: λlog P/2

Other Language

- NESL – functional language
  - has a complexity model – work & depth
  - main feature: apply-to-each operation.

- Lessons
  - High-level (restricted) constructs
  - Force to use these constructs and exhibit parallelism
  - Cost/complexity model to reason about performance