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, ie 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
2 Comment forms
  -- to end-of-line
  Paired /* */
Assignment is :=
Statements end in ;
Hybrid I/O
Basic data types
New concepts --
  config
  region
  [...] notation
  complex operators

```
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 μ and σ

```
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);
        writeln("Mean: ",mu,"S.D. :", sigma);
     end;
```

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

$$\mu = \frac{\sum_i Sample_i}{n} \qquad \sigma = \sqrt{\frac{\sum_i (Sample_i - \mu)^2}{n}}$$

One Slide of Standard Stuff ...

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

Unary Operators: `+, -, !`

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

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

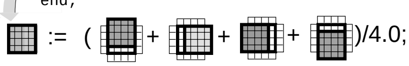
Bit Operators: bnot(),band(),bor(),bxor(),bsl(),bsr()

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

Contol 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;
         eps : float = 0.00001;
region     R = [1..n, 1..n];
var  A, Temp : [R] float;
         err : float;
direction  N = [-1, 0];  S = [ 1, 0];
           E = [ 0, 1];  W = [ 0,-1];
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;
```



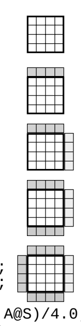
Jacobi Iteration, The Region

```
program Jacobi;
config var n : integer = 512;
         eps : float = 0.00001;
region     R = [1..n, 1..n];
var  A, Temp : [R] float;
         err : float;
direction  N = [-1, 0];  S = [ 1, 0];
           E = [ 0, 1];  W = [ 0,-1];
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;
         eps : float = 0.00001;
region     R = [1..n, 1..n];
var  A, Temp : [R] float;
         err : float;
direction  N = [-1, 0];  S = [ 1, 0];
           E = [ 0, 1];  W = [ 0,-1];
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;
         eps : float = 0.00001;
region     R = [1..n, 1..n];
var  A, Temp : [R] float;
         err : float;
direction  N = [-1, 0];  S = [ 1, 0];
           E = [ 0, 1];  W = [ 0,-1];
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"

  `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"

  `abs(Temp - A)`

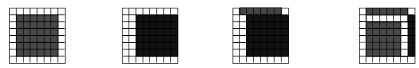
  The (scalar) function is applied to each element of the array
- Programmer-written scalar functions can be promoted, too

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 R = [1..8,1..8];
region C = [2..7,2..7];
var X,Y  : [R] integer;
```

```
E =[ 0,1]
N =[-1,0]
NE=[-1,1]
```



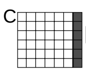
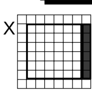
`[C] X:=` `[C] Y@E:=` `[N of C] Y:=` `[C] Y:=X@NE`

Defining Regions Using `of`

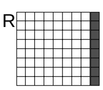
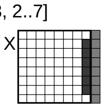
```
E =[ 0,1]
N =[-1,0]
NE=[-1,1]
```

`of` defines a region adjacent to the given region in the given direction

```
region R = [1..8,1..8];
region C = [2..7,2..7];
var X,Y  : [R] integer;
```

C  `[E of C]` X  [E of C] X :=

[E of C] defines the region [8, 2..7]

R  `[E of R]` X  [E of R] X :=

[E of R] defines the region [9, 1..8]

Border Extend On Defining Region Only

Region Calculus

- ZPL's region operators induce a "region calculus"
- Let a dense r-dimensional region be speicifed by its upper and lower limit pairs: $<l_1,u_1>,<l_2,u_2>\ldots <l_r,u_r>$

When $d = (d_1, d_2, \ldots, d_r)$ and $R = <l_1,u_1>,<l_2,u_2>\ldots <l_r,u_r>$, then

`R at d` $= <l_1+d_1,u_1+d_1> <l_2+d_2,u_2+d_2>\ldots<l_r+d_r,u_r+d_r>$

`d of R` satisfies ...

$$<l_i',u_i'> = \begin{cases} <u_i+1,u_i+d_i> & \text{if } d_i > 0 \\ <l_i,u_i> & \text{if } d_i = 0 \\ <l_1+d_i,l_i-1> & \text{if } d_i < 0 \end{cases}$$

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

Regions In Computation

- The region *r* prefixing a statement gives the indices over which all computation on rank *r* arrays is applied

  `[Rr]  ... Ar + Br ...`
- Regions are *scoped*, i.e. a region on an inner statement "over-rides" a region on outer stmt

```
[1..n] begin ...
       [2..n-1] ... A + B ...
       end;
```

- Regions can be *dynamic*, i.e. bounds are evaluated on each execution of the statement

  `[i..j] ... A + B ...`

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`

Reduce	Scan
`+<<`	`+||`
`*<<`	`*||`
`max<<`	`max||`
`min<<`	`min||`
`&<<`	`&||`
`|<<`	`|||`

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

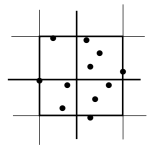
Reduce and scan apply only over applicable region

`[1..i] firsti := +<<A; -- sum first i 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:

```
[R] begin
      rightedge  := max<< X;
      topedge    := max<< Y;
      leftedge   := min<< X;
      bottomedge := min<< Y;
    end;
```



Bounding Box With `point` Type

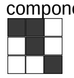
- Rather than using arrays of integers, define a type

```
type point = record
        x : integer;  -- x coordinate
        y : integer;  -- y coordinate
     end;
var  Pts : [1..n] point; -- Points in plane
...
      rightedge  := max<< Pts.x;
      topedge    := max<< Pts.y;
      leftedge   := min<< Pts.x;
      bottomedge := min<< Pts.y;
...
```

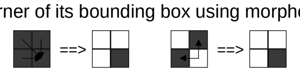

8-way Connected Components

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

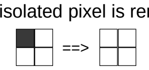
- Assume an array of binary pixels
- Define connectedness 8-ways



- Reduce each component to the lower right corner of its bounding box using morphology:

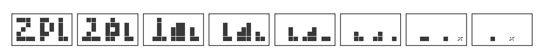


- When an isolated pixel is removed, count it



ZPL Connected Components

```
...
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;
  ...
```

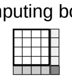


Support for Boundaries

`of` automatically extends arrays to have borders

Borders seamlessly participate in computation

`wrap` and `reflect` assist in computing boundaries



Compare boundary code from SPEC92 benchmark swm

Fortran 90

```
C
C     PERIODIC CONTINUA
C
      uold(m + 1,:n) = uold(1,:n)
      vold(m + 1,:n) = vold(1,:n)
      pold(m + 1,:n) = pold(1,:n)
      u(m + 1,:n) = u(1,:n)
      v(m + 1,:n) = v(1,:n)
      p(m + 1,:n) = p(1,:n)
      uold(:m,n + 1) = uold(:m,1)
      vold(:m,n + 1) = vold(:m,1)
      pold(:m,n + 1) = pold(:m,1)
      u(:m,n + 1) = u(:m,1)
      v(:m,n + 1) = v(:m,1)
      p(:m,n + 1) = p(:m,1)
      uold(m + 1,n + 1) = uold(1,1)
      vold(m + 1,n + 1) = vold(1,1)
      pold(m + 1,n + 1) = pold(1,1)
      u(m + 1,n + 1) = u(1,1)
      v(m + 1,n + 1) = v(1,1)
      p(m + 1,n + 1) = p(1,1)
```

ZPL

```
/* Periodic Continuation */

[e  of I] wrap U,Uold,V,Vold,P,Pold;
[s  of I] wrap U,Uold,V,Vold,P,Pold;
[se of I] wrap U,Uold,V,Vold,P,Pold;
```


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

        ^
        |
            b13
        b12 b23
    b11 b22 b33
    b21 b32 b43
    b31 b42
    b41
```

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

```
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
     ^
     |   b13    c11 c12 c13   a11 a12 a13 a14
    b12 b23     c21 c22 c23   a22 a23 a24 a21
b11 b22 b33     c31 c32 c33   a33 a34 a31 a32
b21 b32 b43     c41 c42 c43   a44 a41 a42 a43
b31 b42
b41             b11 b22 b33
                b21 b32 b43
                b31 b42 b13
                b41 b12 b23
```


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 a12 a13 a14        a11 a12 a13 a14
a21 a22 a23 a24        a22 a23 a24 a21
a31 a32 a33 a34        a33 a34 a31 a32
a41 a42 a43 a44        a44 a41 a42 a43
```

```
for i := 2 to m do
[right of Lop] wrap A;   --Move col 1 to r border
   [i..m,1..n] A := A@right;--Shift last i rows left
end;
```


Four Steps of Skewing A

```
for i := 2 to m do
 [right of Lop] wrap A;       --Move col 1 to r border
    [i..m,1..n] A := A@right; --Shift last i rows left
end;
```

```
a11 a12 a13 a14 | -     a11 a12 a13 a14 | a11
a21 a22 a23 a24 | -     a22 a23 a24 a21 | a21
a31 a32 a33 a34 | -     a32 a33 a34 a31 | a31
a41 a42 a43 a44 | -     a42 a43 a44 a41 | a41
     Initial                 i=2 step

a11 a12 a13 a14 | a11   a11 a12 a13 a14 | a11
a22 a23 a24 a21 | a22   a22 a23 a24 a21 | a22
a33 a34 a31 a32 | a32   a33 a34 a31 a32 | a33
a43 a44 a41 a42 | a42   a44 a41 a42 a43 | a43
     i=3 step                i=4 step
```


Cannon's Algorithm

Skew A, Skew B, Multiply, Accumulate, Rotate

```
           for i := 2 to m do-- Skew A
[right of Lop] wrap A;       -- Move col 1 to border
  [i..m, 1..n] A := A@right; -- Shift last i rows left
           end;
           for i := 2 to p do-- Skew B
[below of Rop] wrap B;       -- Move 1st row below last
  [1..n, i..p] B := B@below; -- Shift last i cols up
           end;
         [Res] C := 0.0;     -- Initialize C
           for i := 1 to n do-- For A&B's common dimension
         [Res] C := C + A*B ;-- Form product and accumulate
[right of Lop] wrap A;       -- Send first col right
         [Lop] A := A@right; -- Shift array left
[below of Rop] wrap B;       -- Send top row down
         [Rop] B := B@below; -- Shift array up
           end;
```


Indexi

- ZPL doesn't need subscripts, but it is still useful to have indices.
- `Index`*`i`* is a (compiler created) constant array giving the value of the ith subscript

  `[1..50] A := 2*Index1;-- A=even nums 2 to 100`

  `Index1` in this instance is `1  2  3  4  5  ...  50`
- The "i" must be a number of a legal dimension

```
[1..n,1..n] Ident := Index1=Index2;--1s on diag
[1..2,1..2] Ident := 1  1  =  1  2      1  0
                     2  2     1  2      0  1
```

- `Index`*`i`* arrays are logical, they use no storage
- It is not legal to assign to `Index`*`i`*

Control-flow Chacteristics

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

```
fact := 1;
for i = 2 to n do fact *= i; end;   -- n!
```

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

```
Fact  := 1;
for I := 2 to N do Fact *= i; end;  -- N!
```

  `N = 3 1 4 1 5` implies `Fact = 6 1 24 1 120`
- 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 D != 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

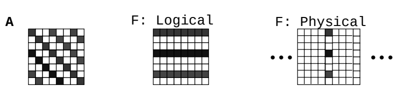
```
-- Take squaroot, preserve sign
if X>=0 then Y := sqrt(X);
        else Y := - sqrt(-X);
end;
```

- 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"

```
ZPL      [1..n,*] F := >>[1..n,1] A;
MATLAB            F  = A(:,ones(1,size(A,2)))
F-90              F  = SPREAD(A[:,1],DIM=2,N)
```



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

```
[1..n,1..n] Col := >>[1..n,k] A;
```

Replicate the kth column

```
[1..n,1..n] Row := >>[k,1..n] A;
```

Replicate the kth row

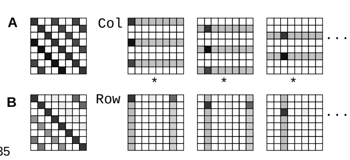
- ZPL recognizes flooded regions ([1..n,*]) and flooded arrays, i.e. arrays defined over flooded regions

Matrix Product

1 Hall of Fame

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

```
[1..n,1..n] C := 0.0;         -- Initialize C
[1..n,1..n] for k := 1 to n do
       [,*]   Col := >>[,k] A;-- Flood kth col of A
       [*,]   Row := >>[k,] B;-- Flood kth row of B
              C := C+Col*Row; -- Accumulate product
            end;
```



Invariant: *On k^{th} iteration the k^{th} term in the dot-product of row i and column j is accumulated in position i,j.*

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

```
region R = [1..n];
var Data,Result:[R] array [1..64,1..64] of float;
      ...
    Result := indexed_matrix_fcn(Data);
```

- The elements of the array are evaluated concurrently, though the computation on each element is sequential

Array/i-array gives 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 F(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;
   var sum : integer;
   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
      -- Imput 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