PGAS Languages

PGAS -- Parallel Global Address Space

- Data may be stored on a unique processor
- Language allows any processor to read/write any data

Languages in this class:
- ZPL from University of Washington (2005 latest work)
- Fortress by Sun Microsystems (mod of Fortran) (2011 last work)
- X10 from IBM Research (active in 2017)
- Berkeley UPC (Unified Parallel C) (March 17, ver 2.24.2)
- Chapel from Cray Inc, April 6, 2017, ver 1.15.0
- May be others ....

Initial thoughts?
- Snyder’s paper -- model is the message
  - Shared memory trying to be implemented by distributed memory
ZPL

Z level Programming Language

University of Washington, Larry Snyder
http://cs.washington.edu/research/zpl (2005 latest modification)

Poker

- GUI IDE for parallel programs
- CTA machine ... Chip
- Phase ...
  - connection network (x)
  - process code (y)
  - export/import interphase
- Switching between phases (z)
- very simple scripting language
Program at the Z level ...

- Results are another language: ZPL
- Doesn’t have a GUI
- Doesn’t specify computing elements
- Very high level ...
- Parallel array processing language
- Syntax similar to Pascal
- A research language, no wide adoption
first class concept

- region  \( R = [ 1 .. n, 1 .. n ]; \)
- \( \text{InR} = [ 2 .. n-1, 2 .. n-1 ]; \)
- \( \text{TopRow} = [ 1 .. n, 1 ]; \)
- \( \text{LeftC} = [ 1, 1 .. n ]; \)

Direction -- used in regions

- direction west = [ -1, 0 ];
- east = [ 1, 0 ]; NE = [ 1, 1 ]; SW = [ -1, -1 ];

region modifiers

- in: region left = west in R; // left column
- of: region lleft = west of InR;
- at: region lInR = InR at west;
- by: direction step = [ 1, 2 ];
- region strip = R by step;
Parallel Arrays

Region of data

- var a, b, c : [R] double;
- d : [InR] double;

Assignment & ops

- [R] a := b;
- [InR] a := b;
- [R] a := b + c;
- [InR] d := a + b;
- var sum : double;
  - [R] sum := +<< a; // reduction
- var f, g : [1 .. n] integer;
  - [1 .. n] f := +|| g; // prefix sum

reduction/prefix ops: +<< *<< min<< max<< |<< (or) &<< (and) bor<< band<< userfunc<< (assoc & comm)
Program examples

hello.z
  □ basic outline

jacobi.z
  □ config variables
    □ -p, -help, -svarname=value
  □ boundary conditions
  □ Communication: @ operator
    □ @^ operator -- wrap
  □ ResetTimer(), CheckTimer()
  □ write() on a region

jacobi2.z
  □ write() to a file
More operations

Flood

\[ [R] A := >>[1..n,i] A; \]
\[ [R] A := >>[i,1..n] A; \]

summa.z

\[ \text{matrix multiply} \]

region FCol = [1..m, *]; FRow = [* , 1..p];

\[ \text{var} \quad \text{Aflood} : [FCol] \text{ double}; \]
\[ \quad \text{Bflood} : [FRow] \text{ double}; \]

\[ \text{for} \quad i := 1 \text{ to } n \quad \text{do} \]
\[ [FCol] \quad \text{Aflood} := >>[i] A; \quad \text{-- flood A col} \]
\[ [FRow] \quad \text{Bflood} := >>[i] B; \quad \text{-- flood B row} \]
\[ \quad C += (\text{Aflood} \ast \text{Bflood}); \quad \text{-- multiply} \]
\[ \text{end;} \]
More operations (page 2)

shat.z
- Index1, Index2
- Shattered flow control

char.z
- Index1 again
- character regions
- write() of char regions

prefix.z
- Index* again
- Prefix operation on a 2D regionn

masks
- Mask : [R] boolean;
- [R with Mask] ...
- [" with Mask] ...
More operations (page 3)

Remap operator 

[R] A := A#[Index2, Index1];
[R] A#[Index2, Index1] := A;
[cannon.z]

Compiling & Running on cluster

/home/phil/zpl/zpl.devel/zpl -- primary directory
[home/phil/setzplseq.bash
zc file.z => file, file_real, and file_dbg (MPI version)
Chapel -- Cascade High Productivity Language

- Done by Cray in Seattle, now at version 1.15.0.
- Brad Chamberlain -- Ph.D. at UW on ZPL
- DARPA-funded HPCS (High Productivity Computing systems)
- Old paper: on Linux lab machines as ~phil/public/cs515/final-chamberlain.pdf
- Web chapel.cray.com
  - Open source -- get your own copy at the web site
- Primary goal: Parallel language and efficient
- Secondary goal: Have a language people actually want to use

Basic ideas of Chapel/ZPL …. global view language or PGAS

- MPI -- fragmented view
  - SPMD
    - Data spread across processors
    - Communication is explicit
- Global view
  - Single program
  - Data in single program
  - Compiler figures out communication
Distributed -- 3 point stencil

var n: int = 1000;
var locN: int = n/numTasks;
var A, B: [0..locN+1] float;
var myItLo: int = 1;
var myItHi: int = locN;
if (iHaveLeftNeighbor) then
    send(left, (A[1])
else
    myItLo = 2;
if (iHaveRightNeighbor) {  
    send(right, A(locN));
    recv(right,A(locN+1));
} else
    myItHi = locN-1;
if (iHaveLeftNeighbor) then
    recv(left,A[0]);
for i := myItLo to myItHi do
    B[i] = (A[i-1] + A[i+1]) / 2
Global view 3 point stencil

```plaintext
var n: int = 1000;
var A, B: [1..n] float;
forall i in 2..n-1 do
  B[i] = (A[i-1] + A[i+1]) / 2;
```

Points of interest:
- forall i in ...
- No worries about numTasks
- No sequential dependencies
- Allows compiler and runtime system to partition A and B
- Allows for algorithm/implementation separation
- Does not show underlying system, shared vs. non-shared
- Portability -- recompile and reuse

Many global view languages appear to be "array processing languages"
Chapel’s goals

- Be a global view language
- Have constructs that allow for large amounts of parallelism
- Support both shared-memory and distributed-memory models
  - Add GPU support in the future
- Support for "data parallelism" and "task parallelism"
  - Beware ... these can conflict!
- Block-imperative programming style
- Allow object orient programming -- in parallel!
- Be a "useful" language, not a research language.
- Be efficient in implementation
- Run on a wide variety of current parallel machines
  - small n SMP machines, small clusters
  - All of the top 500 machines!
- No GPUs yet. (as of April 2017)
writeln ("Hello CSCI 415/515 students!");

□ Longer version
    module Hello {
        config const message = "Hello CSCI 415/515 students!";
        proc main() {
            writeln (message);
        }
    }

□ Parallel Hello World
    config const n = 1000;
    forall i in 1 .. n do writeln ("Hello from iteration ", i);

Sample programs
    □ /home/phil/chapel/examples - directory
      □ primers has interesting programs
Basic Chapel

Block structured language similar to Ada, Pascal, Modula, C ...
Basic types and variables

    var <name> [ : <definition> ] [ = <initializer> ];

Examples:
- examples/primers/variables.chpl
- examples/hello6-taskpar-dist.chpl
- examples/hello5-taskpar.chpl
  - here.id entry
- Notice: -nl 2 does nothing here!

Locales: a unit of a parallel architecture, e.g. cluster node.
- predefined variables:
  - const numLocales: int = ....;
  - const Locales: [1 .. numLocalse] Locale = ...;
- examples/hello6-taskpar-dist.chpl
  - here.id entry
- examples/hello5-taskpar.chpl
  - here.numPEs();
Other Elements

Basic types
- bool, int, uint, real, imag, complex, string
- enum id { enum-constand-list }; (C style with name=value)

Structured types
- record types, union types, class types
- tuple (lightweight record, all fields of same type)
- domains (sets of indices ... e.g. region)
- arrays, index types ... more later.

Flow Control
- if/then/else
- while, do while
- for, forall, coforall
  □ Be careful! coforall can cause slowdowns!
- select X { when ... do ... when ... do ... }
- type select
- label, break, continue
- use statement (for records)
- begin (start an asynchronous task), cobegin (parallel statements)
Domains, Arrays and Array processing

Simple array work: (examples/primer/arrays.chpl)

- var A: [1..n] real; // no "array" key word
  - A[1] = 1.1;
  - A(2) = 2.2; // parens also work
  - A[2..4] = 3.3; // slicing a sub array
- writeln(A); // writes entire array
- writeln(A[2..4]); // or A(2..4)

- var B: [1..n, 1..n] real; // multi-dimensional
  - forall (i,j) in {1..n, 1..n} do
    - B(i,j) = i + j/10.0;
  - forall b in B do b += 1; // Do something to every element
  - forall (i,j) in B.domain do B(i,j) -= 1; // another way

printArr(B); // exact code

def printArr(X: [?D] real) {
  writeln("within printArr, D is: ", D, "\n");
  forall (i,j) in D do
    X(i,j) = -X(i,j);
    writeln("after negating X within printArr, X is:\n", X, "\n");
}

Arrays, ... (page 2)

- var ProbSace: domain(2) = {1..n, 1..n};
- var C, D, E: [ProbSpace] bool;
- for (i,j) in ProbSpace do
  - C(i,j) = (i+j) %3 == 0;
- for ij in ProbSpace do  // ij is a tuple!
  - D(ij) = ij(1) == ij(2);
- E = C; // array assignment
- same as:
  - forall (e,c) in (E,C) do e = c;
- E = 1.0; // promotion of scalar to array
- D[2..n-1, 2..n-1] = C[1..n-2, 3..n]; // sub array assignment
- D[2.., ..] = C[..n-1, ..]; // less specification!
- A = B[n/2, ..]; // slicing and assignment
- const offset = (1,1); // tuple (no type given)
- [ij in ProbSpace[2..n-1, 2..n-1]] F(ij) = B(ij + offset); // ZPL style!
- [b in B] b = -b; // All the array!
- const Z: [ProbSpace] complex = [(i,j) in ProbSpace] i + j*1.0i;
var VarDom = {1..n};
var VarArr: [VarDom] real = [i in VarDom] i;
writeln("Initially, VarArr = ", VarArr, "\n");

// Now, if we reassign VarDom, VarArr will be reallocated with the
// old values preserved and the new values initialized to the element
// type’s default value.
VarDom = {1..2*n};
writeln("After doubling VarDom, VarArr = ", VarArr, "\n");

// As mentioned before, this reallocation preserves values according
// to index, so if we extend the lower bound of the domain, the
// non-zero values will still logically be associated with indices
// 1..n:
VarDom = {-n+1..2*n};
writeln("After lowering VarDom’s lower bound, VarArr = ", VarArr, "\n");

// If the domain shrinks, values will be thrown away
VarDom = {2..n-1};
writeln("After shrinking VarDom, VarArr = ", VarArr, "\n");
var Y: [ProbSpace] [1..3] real;

forall ((i,j), k) in [ProbSpace, 1..3] do
  Y(i,j)(k) = i*10 + j + k/10.0;

writeln("Y is:\n", Y);

View example  examples/programs/jacobi.chpl

- reduce operator
- promotion of abs function over array

Indefinite Domains ...

- var People: domain(string):
- var Age: [People] int;
- People += "John";
- Age("John") = 62;
Chapel Parallel considerations

Data Distributions -- not done for you like ZPL
- examples/primers/distributions.chpl

- Data distributions are standard modules
- BlockDist, CyclicDist, BlockCycDist, ReplicatedDist
- DimensionalDist2D, ReplicatedDim, BlockCycDim
  - see ../chapel/modules/dists
- Block distributions ... (examples/primers/distributions.chpl)

config const n = 8;
const Space = {1..n, 1..n};
const BlockSpace = Space dmapped Block(boundingBox=Space);
var BA: [BlockSpace] int;
forall ba in BA do
  ba = here.id;

const CyclicSpace = Space dmapped Cyclic(startIdx=Space.low);
var CA: [CyclicSpace] int;
forall ca in CA do
  ca = here.id;
Locales and the on clause
for L in Locales {
    on L {
        -- Variables and code
    }
}

□ Issue -- Not parallel!
    □ for statement is sequential
□ two issues:
    □ Location -- solved by locale
□ Parallelism:
    □ data parallelism -- forall
    □ task parallelism -- coforall
    □ async: begin/cobegin
Debugging chapel

- writeln()s -- as expected...
- GDB?
  - Quite difficult ... gdb on multiple machines
  - Variable names changed: name_chpl or name_chpl...
  - <project!> write a front end to gdb for chapel
- VisualDebug module and the tool chplvis
  - Add code to your program (calls to the module)
  - Run a normal, generates data files
  - Run the chplvis program to inspect run
- Primer examples:
  - Simple version: chplvis1.chpl
  - Tags: chplvis2.chpl
  - Repeated tags: chplvis3.chpl
  - Task tracking: chplvis4.chpl
- Other examples:
  - examples/primers/iterators.chpl
  - examples/primers/parIters.chpl
  - tree.chpl -- iterators and procedures ...