CS 470
Spring 2017

Mike Lam, Professor

Parallel Languages

Graphics and content taken from the following:
http://dl.acm.org/citation.cfm?id=2716320
http://chapel.cray.com/papers/BriefOverviewChapel.pdf
http://arxiv.org/pdf/1411.1607v4.pdf

Parallel languages

- Writing efficient parallel code is hard
- We've covered two generic paradigms ...
  - Shared-memory
  - Distributed message-passing
- … and three specific technologies (but all in C!)
  - Pthreads
  - OpenMP
  - MPI
- Can we make parallelism easier by changing our language?
  - Similarly: Can we improve programmer *productivity*?

Productivity

- Economic definition: $Productivity = \frac{Output}{Input}$

- What does this mean for parallel programming?
  - How do you measure *input*?
    - Bad idea: size of programming team
    - "The Mythical Man Month" by Frederick Brooks
  - How do you measure *output*?
    - Bad idea: lines of code

Productivity vs. Performance

- General idea: Produce **better** code **faster**
  - **Better** can mean a variety of things: speed, robustness, etc.
  - **Faster** generally means time/personnel investment
- Problem: productivity often trades off with performance
  - E.g., Python vs. C or Matlab vs. Fortran
  - E.g., garbage collection or thread management

Why?

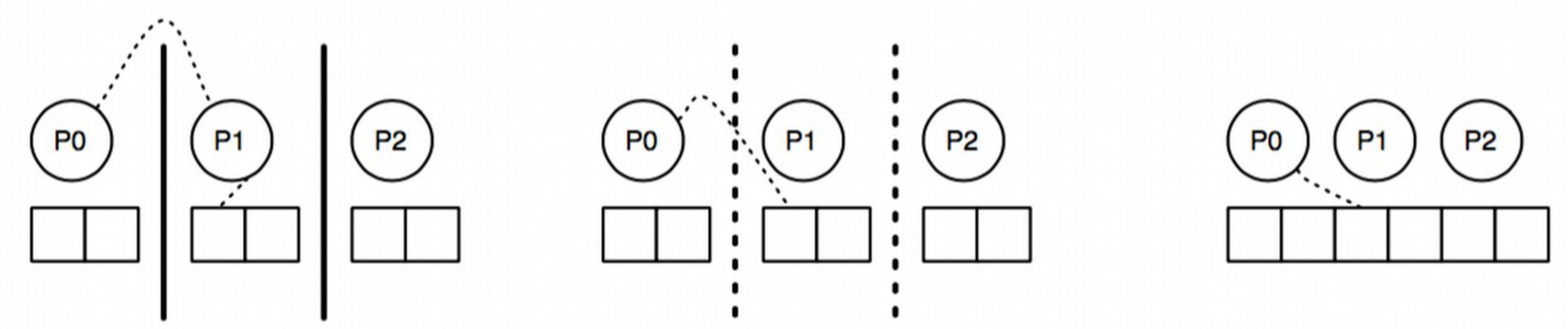
Complexity

- Core issue: handling **complexity**
- Tradeoff: developer effort vs. system effort
  - Hiding complexity from the developer increases the complexity of the system
  - Higher burden on compiler and runtime systems
  - Implicit features cause unpredictable interactions
  - More middleware increases chance of interference and software regressions
  - In distributed systems: locality matters **a lot**

Holy Grail

PGAS

- Partitioned Global Address Space (PGAS)
  - Hybrid of distributed message-passing and shared-memory
  - Programmer sees one global address space
  - Compiler/runtime must sort out the communication
  - Issue: Non-Uniform Memory Access (NUMA) effects



(a) Message-passing (b) Partitioned-memory (PGAS) (c) Shared-memory

Parallel Languages (Mostly PGAS)

- Erlang [Ericsson, 1986], Haskell [1990], and Clojure [2007]
  - Functional languages; most include explicit or implicit parallelism
- High Performance Fortran (HPF) [1993]
  - Designed by committee
- Academic languages
  - ZPL [UW, 1994]
  - Cilk [MIT, 1994] and Cilk Plus [Intel, 2010]
  - Titanium [UC Berkeley, 1998]
- Coarray Fortran (CAF) [1998]
  - Now officially part of the Fortran 2008 standard
- Unified Parallel C (UPC) [1999]
- HPCS languages [starting 2002]
- Julia [2012]

High-Performance Fortran

- Motivation: higher abstractions for parallelism
  - Predefined data distributions and parallel loops
  - Optional **directives** for parallelism (similar to OpenMP)
- Development based on Fortran 90
  - Proposed 1991 w/ intense design efforts in early 1990s
  - Wide variety of influences on the design committee
  - Standardized in 1993 and presented at Supercomputing '93

```
REAL A(1000,1000), B(1000,1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(I,J) WITH A(I,J)
DO J = 2, N
  DO I = 2, N
    A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
          + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
```

Listing 8: Simple relaxation loop in HPF.

For the full story, see "The Rise and Fall of High Performance Fortran: An Historical Object Lesson"
`http://dl.acm.org/citation.cfm?id=1238851`

High-Performance Fortran

- Issues
  - Immature compilers and no reference implementation
  - Poor support for non-standard data distributions
  - Poor code performance; difficult to optimize and tune
  - Slow uptake among the HPC community
- Legacy
  - Effort in 1995-1996 to fix problems with HPF 2.0 standard
  - Eventually dropped in popularity and was largely abandoned
  - Some ideas still had a profound influence on later efforts

ZPL ("Z-level Programming Language")

- Array programming language (1994)
  - All parallelism is implicit
  - Regular data structures with grid alignments
  - Explicit regions and directions

TW = "the world"
NN = "num of neighbors"

```
program Life;
   config const n : integer
   region R = [1..n, 1..n];
   direction nw = [-1, -1]; no = [-1, 0]; ne = [-1, 1];
             w  = [ 0, -1];                e  = [ 0, 1];
             sw = [ 1, -1]; so = [ 1, 0]; se = [ 1, 1];
var  TW : [R] boolean;
     NN : [R] sbyte;
procedure Life();
begin -- Initialize the world
   [R] repeat
          NN := TW@^nw + TW@^no + TW@^ne
              + TW@^w  +          TW^@e
              + TW@^sw + TW@^so + TW@^se;
          TW := (TW & NN = 2) | (NN = 3)
       until !(|<< TW);
end;
```

Listing 9: Conway's Game of Life in ZPL.

field of cells

@^ "wrap-at" operator
(shifts array/matrix)

region specifier

<< reduction operator
(applied here on boolean OR)

Directly influenced the Chapel language

Co-Array Fortran (CAF) [1998]

Extension to Fortran

```
INTEGER n
...
n = 5
```

(a) Allocate private integer.

```
INTEGER n[*]
...
n[p] = 5
```

(b) Allocate shared integer by creating a co-array.

Fig. 7: Both code fragments allocate one integer n for each place.

```
! global_sum
INTEGER :: x(n)[*]                      ! array with a co-array
INTEGER :: local_temp(n)                ! array without a co-array
INTEGER :: me, mypartner                ! indices of places
INTEGER :: n, bit, i, iterations        ! other variables

iterations = log2_images()
bit = 1
me = this_image(x)
DO i = 1,iterations
  mypartner = xor(me, bit)
  bit = shiftl(bit,1)
  CALL sync_all()                       ! barrier
  local_temp(:) = x(:)[mypartner]
  CALL sync_all()                       ! barrier
  x(:) = x(:) + local_temp(:)
ENDDO
```

butterfly reduction pattern

remote memory access

Listing 1: Sum reduction of arrays in CAF.

at end, all values of x are identical

CAF was added to the Fortran standard in 2008

Unified Parallel C (UPC) [1999]

Extension to C

blocking factor

shared/global arrays

threads only execute iterations where affinity is local

parallel for-loop

SPMD and remote data accesses

explicit thread ID check

```
shared [N*N/THREADS] uint8_t orig[N][N], edge[N][N];
int Sobel() {
  int i,j,d1,d2;
  double magnitude;
  //              init    cond     step    affinity
  upc_forall(i=1;  i<N-1; i++;   &edge[i][0]) {
    for(j=1; j<N-1; j++) {
      d1  =  (int) orig[i-1][j+1] - orig[i-1][j-1];
      d1 += ((int) orig[i  ][j+1] - orig[i  ][j-1]) << 1;
      d1 +=  (int) orig[i+1][j+1] - orig[i+1][j-1];
      d2  =  (int) orig[i-1][j-1] - orig[i+1][j-1];
      d2 += ((int) orig[i-1][j  ] - orig[i+1][j  ]) << 1;
      d2 +=  (int) orig[i-1][j+1] - orig[i+1][j+1];
      magnitude = sqrt(d1*d1+d2*d2);
      edge[i][j] = magnitude>255 ? 255 : (uint8_t)magnitude;
    }
  }
  if (MYTHREAD == 0)
    printf("DONE\n");

  return 0;
}
```

Listing 3: Parallel edge detection using Sobel operators in UPC.

UPC is still used, with multiple distributions

DARPA HPCS Program

- High Productivity Computing Systems (HPCS)
- Launched in 2002 with five teams (later narrowed to three)
  - Cray, HP, IBM, SGI, Sun
- Language efforts
  - X10 [IBM, 2004]
    - Based on Java runtime environment
  - Fortress [Sun, 2008]
    - Unique idea: "typesetting" code
    - Discontinued in 2012 due to type system complications
  - Chapel [Cray, 2009]
    - "Cascade High Productivity Language"

X10

Asynchronous PGAS

```
val initializer = (i:Point) => {
  val r = new Random();
  var local_result:double = 0.0D;
  for (c in 1..N) {
    val x = r.nextDouble();
    val y = r.nextDouble();
    if ((x*x + y*y) <= 1.0)
      local_result++;
  }
  local_result
};
val result_array = DistArray.make[Double](Dist.makeUnique(), initializer);
val sum_reducer = (x:Double, y:Double) => { x + y };
val pi = 4 * result_array.reduce(sum_reducer, 0.0) / (N * Place.MAX_PLACES);
```

Listing 6: Estimating π using Monte Carlo method in X10.

X10 is still used, but seems to have lost momentum

Fortress

Hybrid async PGAS and implicit parallelism

```
spawn x.region do
    f(x)
end
```

Computes f(x) wherever x is currently stored

$\Sigma \quad \Pi$

Valid operators

```
var a : RR64 = 0.0
var b : RR64 = 0.0
var c : RR64 = 0.0

DELTA = b^2 - 4 a c
x_1 = (-b - SQRT DELTA)/(2 a)
x_2 = (-b + SQRT DELTA)/(2 a)
```

(a) Small example program in Fortress without unicode characters.

$$\mathtt{var}\ a : \mathbb{R}64 = 0.0$$
$$\mathtt{var}\ b : \mathbb{R}64 = 0.0$$
$$\mathtt{var}\ c : \mathbb{R}64 = 0.0$$
$$\Delta = b^2 - 4\,a\,c$$
$$\mathrm{x}_1 = \frac{-b - \sqrt{\Delta}}{2\,a}$$
$$\mathrm{x}_2 = \frac{-b + \sqrt{\Delta}}{2\,a}$$

(b) Small example program in Fortress that supports unicode characters.

Officially discontinued in 2012; source code is still available

Chapel

- New language designed for parallel computation
  - Heavily influenced by ZPL and High-Performance Fortran
- Design is based on user requirements
  - Recent graduates: "a language similar to Python, Matlab, Java, etc."
  - HPC veterans: "a language that gives me complete control"
  - Scientists: "a language that lets me focus on the science"
- Chapel stated goals:
  - *"A language that lets scientists **express** what they want …*
  - *… without taking away the **control** that veterans want …*
  - *… in a package that's as **attractive** as recent graduates want."*

Chapel themes

- Open source compiler (Apache license)
  - Uses Pthreads for local concurrency
  - Uses GASNet library for distributed communication
- Multi-resolution parallelism
  - Multiple levels of abstraction (task and data parallelism)
  - Higher levels build on lower levels
  - Developers can mix-and-match as desired
- Locality control
  - PGAS memory model; developers control data locales
- Reduced gap between HPC and mainstream
  - Type inference, generic programming, optional OOP

Chapel examples

```
var done: bool = true;     // 'done' is a boolean variable, initialized to 'true'

proc abs(x: int): int {    // a procedure to compute the absolute value of 'x'
  if (x < 0) then
    return -x;
  else
    return x;
}

var Hist: [-3..3] int,                // a 1D array of integers
    Mat: [0..#n, 0..#n] complex,      // a 2D array of complexes
    Tri: [i in 1..n] [1..i] real;     // a "triangular" skyline array

var count = 0;             // '0' is an integer, so 'count' is too
const area = 2*r;          // if 'r' is an int/real/complex, 'area' will be too
var len = computeLen();    // 'len' is whatever type computeLen() returns
config const n = 10;       // can be overridden by "--n=X" on the command line

for i in 1..n do           // print 1, 2, 3, ..., n
  writeln(i);

for elem in Mat do         // increment all elements in Mat
  elem += 1;
```

Chapel examples

domain definition

```
const BigD = {0..n+1, 0..n+1} dmapped Block(boundingBox=[0..n+1, 0..n+1]),
            D: subdomain(BigD) = {1..n, 1..n};
var A, Temp: [BigD] real;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;
  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);
```

implicit data parallelism

average neighbors' values

Listing 4: Jacobi iteration example in Chapel (data parallel).

arbitrary domain array parameter

```
proc quickSort(arr: [?D],
               thresh = log2(here.numCores()), depth = 0,
               low: int = D.low, high: int = D.high) {
  if high - low < 8 {
    bubbleSort(arr, low, high);
  } else {
    const pivotVal = findPivot(arr, low, high);
    const pivotLoc = partition(arr, low, high, pivotVal);
    serial(depth >= thresh) do cobegin {
      quickSort(arr, thresh, depth+1, low, pivotLoc-1);
      quickSort(arr, thresh, depth+1, pivotLoc+1, high);
} } }
```

explicit task parallelism

Listing 5: Parallel Quicksort example in Chapel (task parallel).

Execution models

- Fully SPMD
  - Fixed number of threads spawn at launch and diverge based on thread index checks (similar to MPI)
- Asynchronous PGAS
  - Single main thread; worker threads spawn automatically in marked parallel regions (similar to OpenMP)
- Fully Implicit
  - Threads spawned dynamically by runtime system as appropriate; no explicit parallel regions

Topologies and data access

- Topologies
  - Flat (indexed)
  - Rectangular / hypercube / torus / mesh
  - Hierarchical
- Access cost function
  - Two-level (local vs. remote)
  - Multi-level
- Data distribution
  - Implicit vs. explicit
  - Regular vs. irregular (domain uniformity)
- Remote data accesses
  - Implicit vs. explicit
  - Local vs. global

PGAS Language Summary

Language	Parallel Execution	Topology	Data Distribution	Distributed Data	Remote Access	Array Indexing
Retrospective PGAS languages						
HPF	Implicit	User defined mesh	Explicit	Regular	Implicit	Global
ZPL	Implicit	User defined mesh	Implicit	Regular	Explicit	Global
GA	SPMD	Flat ordered set	Explicit	Regular	Explicit	Global
Original PGAS languages						
CAF	SPMD	User defined mesh	Implicit	Regular	Explicit	Local
Titanium	SPMD	Flat ordered set	Explicit	Irregular	Expl. + Impl.	not applicable
UPC	SPMD	Flat ordered set	Explicit	Reg. + Irreg.	Implicit	Global
HPCS PGAS languages						
Chapel	APGAS + Impl.	User defined mesh	Explicit	Reg. + Irreg.	Expl. + Impl.	Global
X10	APGAS	Flat ordered set	Explicit	Reg. + Irreg.	Explicit	Global
Fortress	APGAS + Impl.	Hierarchical	Explicit	Reg. + Irreg.	Expl. + Impl.	Global

lower ≈ newer

Lessons learned??

For more details and full paper:
`http://dl.acm.org/citation.cfm?id=2716320`

Julia

- New dynamic language for numeric computing
  - Combines ideas from Python, Matlab, R, and Fortran
  - Mantra: "*vectorize when it feels right*"
  - Core is implemented in C/C++, JIT-compiled to native machine code
  - Includes a REPL
  - IJulia browser-based graphical notebook interface
- Goal: never make the developer resort to using two languages
  - Similar philosophy in Chapel community

```
function mandelbrot(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    return maxiter
end
```

Calculate Mandelbrot function

```
nheads = @parallel (+) for i=1:100000000
  int(randbool())
end
```

Simulate coin tosses in parallel

Python for HPC

- Primary strength: writeability
  - Easy-to-learn
  - Low overhead and boilerplate
- Secondary strength: libraries & frameworks
  - NumPy (supports large, multi-dimensional matrices)
  - SciPy (scientific computing library that uses NumPy)
  - SageMath (open source Mathematica/Matlab alternative)
  - IPython (interactive parallel computing)
  - Many others!

Holy Grail impossible?

Challenge: design your own parallel language!

What would it look like?