Overview

A. Introduction to PGAS (~ 30 mts)

B. Introduction to Languages
   A. UPC (~ 65 mts)
   B. X10 (~ 65 mts)
   C. Chapel (~ 65 mts)

C. Comparison of Languages (~45 minutes)
   A. Comparative Heat transfer Example
   B. Comparative Summary of features
   C. Discussion

D. Hands-On (90 mts)
Comparison of Languages

UPC
2D Heat Conduction Problem

Based on the 2D Partial Differential Equation (1), 2D Heat Conduction problem is similar to a 4-point stencil operation, as seen in (2):

\[ \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{1}{\alpha} \frac{\partial T}{\partial t} \quad (1) \]

\[ T_{i,j}^{t+1} = \frac{1}{4 \cdot \alpha} \left( T_{i-1,j}^{t} + T_{i+1,j}^{t} + T_{i,j-1}^{t} + T_{i,j+1}^{t} \right) \quad (2) \]
Heat Transfer in Pictures

\[ \sum \left( \begin{array}{c}
\text{sum of relevant areas}
\end{array} \right) \div 4 \]

repeat until max change < \( \varepsilon \)
2D Heat Conduction Problem

shared [BLOCKSIZE] double grids[2][N][N];
shared double dTmax_local[THREADS], dTmax_shared;
int x, y, nr_iter = 0, finished = 0;
int dg = 1, sg = 0;
double dTmax, dT, T, epsilon = 0.0001;
do {
    dTmax = 0.0;
    for( y=1; y<N-1; y++ ){
        upc_forall( x=1; x<N-1; x++; &grids[sg][y][x] ){
            T = (grids[sg][y-1][x] + grids[sg][y+1][x] +
                 grids[sg][y][x-1] + grids[sg][y][x+1])
                / 4.0;
            dT = T - grids[sg][y][x];
            grids[dg][y][x] = T;
            if( dTmax < fabs(dT) )
                dTmax = fabs(dT);
        }
    }
}

Affinity field, used for work distribution

4-pt stencil
if( dTmax_shared < epsilon )
    finished = 1;
else{
    /*swapping the source & destination “pointers”*/
    dg = sg;
    sg = !sg;
}
while( !finished );
upc_barrier;

2D Heat Conduction Problem

reduction operation using UPC collectives library
Comparison of Languages

X10
Heat transfer in X10

X10 permits smooth variation between multiple concurrency styles

- “High-level” ZPL-style (operations on global arrays)
  - Chapel “global view” style
  - Expressible, but relies on “compiler magic” for performance

- OpenMP style
  - Chunking within a single place

- MPI-style
  - SPMD computation with explicit all-to-all reduction
  - Uses clocks

- “OpenMP within MPI” style
  - For hierarchical parallelism
  - Fairly easy to derive from ZPL-style program.
Heat Transfer in X10 – ZPL style

class Stencil2D {
    static type Real=Double;
    const n = 6, epsilon = 1.0e-5;

    const BigD = Dist.makeBlock([0..n+1, 0..n+1]),
        D = BigD | [1..n, 1..n],
        LastRow = [0..0, 1..n] to Region;
    val A = Array.make[Real](BigD), Temp = Array.make[Real](BigD);
    {
        A(LastRow) = 1.0D;
    }
    def run() {
        do {
            finish ateach (p in D)
                Temp(p) = A(p.stencil(1)).reduce(Double.+, 0.0)/4;

            val delta = (A(D)-Temp(D)).lift(Math.abs).reduce(Math.max, 0.0);
            A(D) = Temp(D);
        } while (delta > epsilon);
    }
}
Heat Transfer in X10 – ZPL style

◆ Cast in fork-join style rather than SPMD style
  – Compiler needs to transform into SPMD style

◆ Compiler needs to chunk iterations per place
  – Fine grained iteration has too much overhead

◆ Compiler needs to generate code for distributed array operations
  – Create temporary global arrays, hoist them out of loop, etc.

◆ Uses implicit syntax to access remote locations.

Simple to write — tough to implement efficiently
Heat Transfer in X10 – II

```scala
def run() {
  val D_Base = Dist.makeUnique(D.places());
  do {
    finish at each (z in D_Base)
    for (p in D | here)
      Temp(p) = A(p.stencil(1)).reduce(Double.+, 0.0)/4;

    val delta = (A(D) - Temp(D)).lift(Math.abs).reduce(Math.max, 0.0);
    A(D) = Temp(D);
  } while (delta > epsilon);
}
```

- **Flat parallelism**: Assume one activity per place is desired.
- **D.places()** returns ValRail of places in D.
  - **Dist.makeUnique(D.places())** returns a unique distribution (one point per place) over the given ValRail of places
- **D | x** returns sub-region of D at place x.

Explicit Loop Chunking
def run() {
    val D_Base = Dist.makeUnique(D.places());
    val blocks = DistUtil.block(D, P);
    do {
        finish at each (z in D_Base)
        foreach (q in 1..P)
            for (p in blocks(here, q))
                Temp(p) = A(p.stencil(1)).reduce(Double.+, 0.0)/4;

        val delta = (A(D) - Temp(D)).lift(Math.abs).reduce(Math.max, 0.0);
        A(D) = Temp(D);
    } while (delta > epsilon);
}
Heat Transfer in X10 – IV

```python
def run() {
    finish async {
        val c = clock.make();
        val D_Base = Dist.makeUnique(D.places());
        val diff = Array.make[Real](D_Base),
            scratch = Array.make[Real](D_Base);
        ateach (z in D_Base) clocked(c) do {
            diff(z) = 0.0D;
            for (p in D | here) {
                Temp(p) = A(p.stencil(1)).reduce(Double.+, 0.0)/4;
                diff(z) = Math.max(diff(z), Math.abs(A(p) - Temp(p)));
            }
            next;
            A(D | here) = Temp(D | here);
            reduceMax(z, diff, scratch);
        } while (diff(z) > epsilon);
    }
}
```

- **reduceMax()** performs an all-to-all max reduction.

SPMD with all-to-all reduction == MPI style
def run() {
    finish async {
        val c = clock.make();
        val D_Base = Dist.makeUnique(D.places());
        val diff = Array.make[Real](D_Base),
        scratch = Array.make[Real](D_Base);
        aeach (z in D_Base) clocked(c)
        foreach (q in 1..P) clocked(c)
            do {
                if (q==1) diff(z) = 0.0D;
                var myDiff: Real = 0.0D;
                for (p in blocks(here,q)) {
                    Temp(p) = A(p.stencil(1)).reduce(Double.+, 0.0)/4;
                    myDiff = Math.max(myDiff, Math.abs(A(p) - Temp(p)));
                }
                atomic diff(z) = Math.max(myDiff, diff(z));
                next;
                A(blocks(here,q)) = Temp(blocks(here,q));
                if (q==1) reduceMax(z, diff, scratch);
                next;
        } while (diff(z) > epsilon);
    }
}
Heat Transfer in X10 – VI

- All previous versions permit fine-grained remote access
  - Used to access boundary elements

- Much more efficient to transfer boundary elements in bulk between clock phases.

- May be done by allocating extra “ghost” boundary at each place
  - API extension: `Dist.makeBlock(D, P, f)`
    - D: distribution, P: processor grid, f: region→region transformer

- `reduceMax()` phase overlapped with ghost distribution phase
Comparison of Languages

Chapel
Heat Transfer in Chapel
Heat Transfer in Pictures

\[ \sum \left( \begin{array}{c} \begin{array}{ccc} & \text{+} & \\ \text{+} & & \text{+} \\ & \\ \end{array} \end{array} \right) \div 4 \]

repeat until max change < \( \varepsilon \)
Heat Transfer in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [(i,j) in D] 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);

writeln(A);
```
Heat Transfer in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
do {
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4.0;
    const delta = max reduce abs(A(D) - Temp(D));
    A[D] = Temp[D];
} while (delta > epsilon);
writeln(A);
```

**Declare program parameters**

- `config` ⇒ can be set on executable command-line
  ```prompt>```
  ```jacobi --n=10000 --epsilon=0.0001```

- `const` ⇒ can’t change values after initialization

Note that no types are given; inferred from initializer
- `n` ⇒ `integer` (current default, 32 bits)
- `epsilon` ⇒ `floating-point` (current default, 64 bits)
Heat Transfer in Chapel

```
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);
```

Declare domains (first class index sets)

- **domain(2)** ⇒ 2D arithmetic domain, indices are integer 2-tuples
- **subdomain(P)** ⇒ a domain of the same type as \( P \) whose indices are guaranteed to be a subset of \( P \)'s

```

0 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 0
0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4

BigD D LastRow
```

- **exterior** ⇒ one of several built-in domain generators
Heat Transfer in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
```

Declare arrays

- `var` can be modified throughout its lifetime
- `: T` declares variable to be of type `T`
- `:[D] T` array of size `D` with elements of type `T`
- `(no initializer)` values initialized to default value (0.0 for reals)
Heat Transfer in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
```

Set Explicit Boundary Condition

- indexing by domain ⇒ slicing mechanism
- array expressions ⇒ parallel evaluation
Heat Transfer in Chapel

**Compute 5-point stencil**

\[(i,j) \text{ in } D \] \Rightarrow \text{parallel forall expression over } D \text{'s indices, binding them to new variables } i \text{ and } j

**Note:** since \( (i,j) \in D \) and \( D \subseteq BigD \) and \( Temp: [BigD] \)

\Rightarrow \text{no bounds check required for } Temp(i,j) \\
\text{with compiler analysis, same can be proven for } A \text{'s accesses}

\[
\sum \left( \begin{array}{c}
\text{orange} \\
\text{yellow} \\
\text{green} \\
\text{blue}
\end{array} \right) \div 4
\]

\[
[(i,j) \text{ in } D] \quad Temp(i,j) = \frac{A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)}{4};
\]

```
const delta = max reduce abs(A[D] - Temp[D]);
A[D] = Temp[D];
} while (delta > epsilon);
```

```
writeln(A);
```
Chapel (25)

Heat Transfer in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],

Compute maximum change

`op reduce` ⇒ collapse aggregate expression to scalar using `op`

_Promotion:_ `abs()` and – are scalar operators, automatically promoted to work with array operands

```
do {
    [(i,j) in D] 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);
```

writeln(A);
```
Heat Transfer in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    [(i,j) in D] 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);

writeln(A);
```

Copy data back & Repeat until done

uses slicing and whole array assignment
standard do...while loop construct
Heat Transfer in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [(i,j) in D]
        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);

writeln(A);
```

Write array to console

If written to a file, parallel I/O would be used
Heat Transfer in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1] distributed Block,
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
```

With this change, same code runs in a distributed manner
Domain distribution maps indices to locales
⇒ decomposition of arrays & default location of iterations over locales
Subdomains inherit parent domain’s distribution
Heat Transfer in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1] distributed Block,
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    [(i,j) in D] 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);

writeln(A);
```
Heat Transfer in Chapel (Variations)
Heat Transfer in Chapel (double buffered version)

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1] distributed Block,
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A : [1..2] [BigD] real;
A[..][LastRow] = 1.0;

var src = 1, dst = 2;

do {
    [(i,j) in D] A(dst)(i,j) = (A(src)(i-1,j) + A(src)(i+1,j)
    + A(src)(i,j-1) + A(src)(i,j+1)) / 4;

    const delta = max reduce abs(A[src] - A[dst]);
    src <=> dst;
} while (delta > epsilon);

writeln(A);
```

Chapel (31)
Heat Transfer in Chapel (named direction version)

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1] distributed Block,
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

const north = (-1,0), south = (1,0), east = (0,1), west = (0,-1);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [ind in D] Temp(ind) = (A(ind + north) + A(ind + south)
        + A(ind + east) + A(ind + west)) / 4;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Heat Transfer in Chapel (named direction version)
Heat Transfer in Chapel (array of offsets version)

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1] distributed Block,
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

param offset : [1..4] (int, int) = ((-1,0), (1,0), (0,1), (0,-1));

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [ind in D] Temp(ind) = (+ reduce [off in offset] A(ind + off))
        / offset.numElements;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Heat Transfer in Chapel (sparse offsets version)

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1] distributed Block,
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

param stencilSpace: domain(2) = [-1..1, -1..1],
    offSet: sparse subdomain(stencilSpace)
        = ((-1,0), (1,0), (0,1), (0,-1));
var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [ind in D] Temp(ind) = (+ reduce [off in offSet] A(ind + off))
        / offSet.numIndices;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Chapel (34)
Heat Transfer in Chapel (UPC-ish version)

```chapel
config const N = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..#N, 0..#N] distributed Block,
    D: subdomain(BigD) = D.expand(-1);

var grids : [0..1] [BigD] real;
var sg = 0, dg = 1;

do {
    [(x,y) in D] grids(dst)(x,y) = (grids(src)(x-1,y)
        + grids(src)(x+1,y)
        + grids(src)(x,y-1)
        + grids(src)(x,y+1)) / 4;

    const dTmax = max reduce abs(grids(src) - grids(dst));
    src <=> dst;
} while (dTmax > epsilon);

writeln(A);
```
Comparison of Languages

Comparative Feature Matrix
# Features Matrix

<table>
<thead>
<tr>
<th>Feature</th>
<th>UPC</th>
<th>X10</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory model</strong></td>
<td></td>
<td>PGAS</td>
<td></td>
</tr>
<tr>
<td><strong>Programming/Execution model</strong></td>
<td>SPMD</td>
<td>Multithreaded</td>
<td>Global-view / Multithreaded</td>
</tr>
<tr>
<td><strong>Base Language</strong></td>
<td>C</td>
<td>Java</td>
<td>N/A (influences include C, Modula, Java, Perl, CLU, ZPL, MTA, Scala, …)</td>
</tr>
<tr>
<td><strong>Nested Parallelism</strong></td>
<td>Not supported</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>Incremental Parallelization of code</strong></td>
<td>Indirectly supported</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td><strong>Locality Awareness</strong></td>
<td>Yes (Blocking and affinity)</td>
<td>Yes</td>
<td>Yes (affinity of code and data to locales; distributed data aggregates)</td>
</tr>
<tr>
<td><strong>Dynamic Parallelism</strong></td>
<td>Still in research</td>
<td>Yes – Asynchronous PGAS</td>
<td>Yes – Asynchronous PGAS</td>
</tr>
</tbody>
</table>
## Features Matrix

<table>
<thead>
<tr>
<th></th>
<th>UPC</th>
<th>X10</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Implicit/Explicit</strong></td>
<td><strong>Both</strong></td>
<td><strong>Both</strong></td>
<td><strong>Implicit; User can assert locality of a code block (checked at compile-/runtime)</strong></td>
</tr>
<tr>
<td><strong>Communications</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Collective Operations</strong></td>
<td>No explicit collective operations but remote string functions are provided</td>
<td>Yes (possibly nonblocking, initiated by single activity)</td>
<td><strong>Reductions, scans, whole-array operations</strong></td>
</tr>
<tr>
<td><strong>Work Sharing</strong></td>
<td>Different affinity values in upc_forall</td>
<td>Work-stealing supported on a single node.</td>
<td>Currently, must be explicitly done by the user; future versions will support a work-sharing mode</td>
</tr>
<tr>
<td><strong>Data Distribution</strong></td>
<td>Block, round-robin</td>
<td>Standard distributions, users may define more.</td>
<td>Library of standard distributions + ability for advanced users to define their own</td>
</tr>
<tr>
<td><strong>Memory Consistency</strong></td>
<td>Strict and relaxed allowed on block statements or variable by variable basis</td>
<td>Under development. (See theory in PPoPP 07)</td>
<td><strong>Strict with respect to sync/single variables; relaxed otherwise</strong></td>
</tr>
</tbody>
</table>
## Features Matrix

<table>
<thead>
<tr>
<th>Dynamic Memory Allocation</th>
<th>UPC</th>
<th>X10</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private or shared with or without blocking</td>
<td>Supports objects and arrays.</td>
<td>No pointers -- all dynamic allocations are through objects &amp; array resizing</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synchronization</th>
<th>UPC</th>
<th>X10</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barriers, split phase barrier, locks, and memory consistency control</td>
<td>Conditional atomic blocks, dynamic barriers (clocks)</td>
<td>Synchronization and single variables; transactional memory-style atomic blocks</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type Conversion</th>
<th>UPC</th>
<th>X10</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td>C rules Casting of shared pointers to private pointers</td>
<td>Coercions, conversions supported as in OO languages</td>
<td>C#-style rules plus explicit conversions</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pointers To Shared Space</th>
<th>UPC</th>
<th>X10</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>global-view distributed arrays</th>
<th>UPC</th>
<th>X10</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes, but 1D only</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## Partial Construct Comparison

<table>
<thead>
<tr>
<th>Constructs</th>
<th>UPC</th>
<th>X10</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel loops</td>
<td>upc_forall</td>
<td>foreach, ateach</td>
<td>forall, coforall</td>
</tr>
<tr>
<td>Concurrency spawn</td>
<td>N/A</td>
<td>async, future,</td>
<td>begin, cobegin,</td>
</tr>
<tr>
<td>Termination detection</td>
<td>N/A</td>
<td>finish</td>
<td>sync</td>
</tr>
<tr>
<td>Distribution construct</td>
<td>affinity in upc_forall, blocksize in work distribution</td>
<td>places, regions, distributions</td>
<td>locales, domains, distributions</td>
</tr>
<tr>
<td>Atomicity control</td>
<td>N/A</td>
<td>Basic atomic blocks</td>
<td>TM-based atomic blocks</td>
</tr>
<tr>
<td>Data-flow synchronization</td>
<td>N/A</td>
<td>Conditional atomic blocks</td>
<td>single variables</td>
</tr>
<tr>
<td>Barriers</td>
<td>upc_barrier</td>
<td>clocks</td>
<td>sync variables</td>
</tr>
</tbody>
</table>
You might consider using UPC if...

- you prefer C-based languages
- the SPMD programming/execution model fits your algorithm
- 1D block-cyclic/cyclic global arrays fit your algorithm
- you need to do production work today
You might consider using X10 if...

- you prefer Java-style languages
- you require dynamic/nested parallelism than SPMD
- you require multidimensional global arrays
- you're able to work with an emerging technology
You might consider using Chapel if...

- you're not particularly tied to any base language
- you require dynamic/nested parallelism than SPMD
- you require multidimensional global arrays
- you're able to work with an emerging technology
Discussion
Overview

A. Introduction to PGAS (~ 30 mts)
B. Introduction to Languages
   A. UPC (~ 65 mts)
   B. X10 (~ 65 mts)
   C. Chapel (~ 65 mts)
C. Comparison of Languages (~45 minutes)
   A. Comparative Heat transfer Example
   B. Comparative Summary of features
   C. Discussion
D. Hands-On (90 mts)
D. Hands-On