M04- Programming Using the Partitioned Global Address Space (PGAS) Model

Tarek El-Ghazawi, GWU
Brad Chamberlain, Cray
Vijay Saraswat, IBM

Overview
A. Introduction to PGAS (~ 45 mts)
B. Introduction to Languages
   A. UPC (~ 60 mts)
   B. X10 (~ 60 mts)
   C. Chapel (~ 60 mts)
C. Comparison of Languages (~45 minutes)
   A. Comparative Heat transfer Example
   B. Comparative Summary of features
   C. Discussion
D. Hands-On (90 mts)
A. Introduction to PGAS

The common architectural landscape

SMP Clusters

Massively Parallel Processor Systems

IBM Blue Gene

Cray XTS
Programming Models

◆ What is a programming model?
  – The logical interface between architecture and applications

◆ Why Programming Models?
  – Decouple applications and architectures
    ◆ Write applications that run effectively across architectures
    ◆ Design new architectures that can effectively support legacy applications

◆ Programming Model Design Considerations
  – Expose modern architectural features to exploit machine power and improve performance
  – Maintain Ease of Use

Examples of Parallel Programming Models

◆ Message Passing
◆ Shared Memory (Global Address Space)
◆ Partitioned Global Address Space (PGAS)
The Message Passing Model

- Concurrent sequential processes
- Explicit communication, two-sided
- Library-based
- Positive:
  - Programmers control data and work distribution.
- Negative:
  - Significant communication overhead for small transactions
  - Excessive buffering
  - Hard to program in
- Example: MPI

Legend:
- Thread/Process
- Address Space
- Memory Access
- Message

The Shared Memory Model

- Concurrent threads with shared space
- Positive:
  - Simple statements
  - Read remote memory via an expression
  - Write remote memory through assignment
- Negative:
  - Manipulating shared data leads to synchronization requirements
  - Does not allow locality exploitation
- Example: OpenMP, Java

Legend:
- Thread/Process
- Address Space
- Memory Access
Hybrid Model(s)
Example: Shared + Message Passing

Example: OpenMP at the node (SMP), and MPI in between

The PGAS Model

Concurrent threads with a partitioned shared space
- A datum may reference data in other partitions
- Global arrays have fragments in multiple partitions

Positive:
- Helps in exploiting locality
- Simple statements as shared memory

Negative:
- Sharing all memory can result in subtle bugs and race conditions
- Examples: This Tutorial! UPC, X10, Chapel, CAF, Titanium
PGAS vs. other programming models/languages

<table>
<thead>
<tr>
<th></th>
<th>UPC, X10, Chapel</th>
<th>MPI</th>
<th>OpenMP</th>
<th>HPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory model</td>
<td>PGAS</td>
<td></td>
<td></td>
<td>Distributed Shared Memory</td>
</tr>
<tr>
<td>Notation</td>
<td>Language</td>
<td>Library</td>
<td>Annotations</td>
<td>Language</td>
</tr>
<tr>
<td>Global arrays?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Global pointers/references?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Locality Exploitation</td>
<td>Yes</td>
<td>Yes, necessarily</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The heterogeneous/accelerated architectural landscape

Cray XT3h: FPGA/Vector-accelerated Opteron
Road Runner: Cell-accelerated Opteron
Example accelerator technologies

From NVidia CUDA Programming Guide

The current architectural landscape

- Substantial architectural innovation is anticipated over the next ten years.
  - Hardware situation remains murky, but programmers need stable interfaces to develop applications.
- Heterogenous accelerator-based systems will exist, raising serious programmability challenges.
  - Programmers must choreograph interactions between heterogenous processors, memory subsystems.
- Multicore systems will dramatically raise the number of cores available to applications.
  - Programmers must understand concurrent structure of their applications.
- Applications seeking to leverage these architectures will need to go beyond data-parallel, globally synchronizing MPI model.
- These changes, while most profound for HPC now, will change the face of commercial computing over time.
Asynchronous PGAS

- Explicit concurrency
- SPMD is a special case
- Asynchronous activities can be started and stopped in a given space partition

Concurrency is made explicit and programmable.

How do we realize APGAS?

- Through an APGAS library in C, Fortran, Java (co-habiting with MPI)
  - Implements PGAS
    - Remote references
    - Global data-structures
  - Implements inter-place messaging
    - Optimizes inlinable asyncs
  - Implements global and/or collective operations
  - Implements intra-place concurrency
    - Atomic operations
    - Algorithmic scheduler
- Builds on XL UPC runtime, GASNet, ARMCI, LAPI, DCMF, DaCS, Cilk runtime, Chapel runtime

- Through languages
  - Asynchronous Co-Array Fortran
    - extension of CAF with asyncs
  - Asynchronous UPC (AUPC)
    - Proper extension of UPC with asyncs
  - X10 (already asynchronous)
    - Extension of sequential Java
    - Chapel (already synchronous)

- Language runtimes share common APGAS runtime
- Libraries reduce cost of adoption, languages offer enhanced productivity benefits
  - Customer gets to choose
Overview

A. Introduction to PGAS (~ 45 mts)

B. Introduction to Languages
   A. UPC (~ 60 mts)
   B. X10 (~ 60 mts)
   C. Chapel (~ 60 mts)

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

D. Hands-On (90 mts)
UPC Overview

1) UPC in a nutshell
   - Memory model
   - Execution model
   - UPC Systems

2) Data Distribution and Pointers
   - Shared vs Private Data
   - Examples of data distribution
   - UPC pointers

3) Workload Sharing
   - upc_forall

4) Advanced topics in UPC
   - Dynamic Memory Allocation
   - Synchronization in UPC
   - UPC Libraries

5) UPC Productivity
   - Code efficiency

Introduction

◆ UPC – Unified Parallel C
◆ Set of specs for a parallel C
  – v1.0 completed February of 2001
  – v1.1.1 in October of 2003
  – v1.2 in May of 2005
◆ Compiler implementations by vendors and others
◆ Consortium of government, academia, and HPC vendors
  including IDA CCS, GWU, UCB, MTU, UMN, ARSC,
  UMCP, U of Florida, ANL, LBNL, LLNL, DoD, DoE, HP,
  Cray, IBM, Sun, Intrepid, Etnus, ...
Introduction cont.

- UPC compilers are now available for most HPC platforms and clusters
  - Some are open source
- A debugger is available and a performance analysis tool is in the works
- Benchmarks, programming examples, and compiler testing suite(s) are available
- Visit www.upcworld.org or upc.gwu.edu for more information

UPC Systems

- Current UPC Compilers
  - Hewlett-Packard
  - Cray
  - IBM
  - Berkeley
  - Intrepid (GCC UPC)
  - MTU
- UPC application development tools
  - Totalview
  - PPW from UF
UPC Home Page

http://www.upc.gwu.edu

UPC textbook now available

- **UPC: Distributed Shared Memory Programming**
  Tarek El-Ghazawi
  William Carlson
  Thomas Sterling
  Katherine Yelick

- Wiley, May, 2005
What is UPC?

- Unified Parallel C
- An explicit parallel extension of ISO C
- A partitioned shared memory parallel programming language

UPC Execution Model

- A number of threads working independently in a **SPMD** fashion
  - MYTHREAD specifies thread index (0..THREADS-1)
  - Number of threads specified at compile-time or run-time
- Synchronization when needed
  - Barriers
  - Locks
  - Memory consistency control
A pointer-to-shared can reference all locations in the shared space, but there is data-thread affinity.

A private pointer may reference addresses in its private space or its local portion of the shared space.

Static and dynamic memory allocations are supported for both shared and private memory.

---

UPC Overview

1) UPC in a nutshell
   - Memory model
   - Execution model
   - UPC Systems

2) Data Distribution and Pointers
   - Shared vs. Private Data
   - Examples of data distribution
   - UPC pointers

3) Workload Sharing
   - upc_forall

4) Advanced topics in UPC
   - Dynamic Memory Allocation
   - Synchronization in UPC
   - UPC Libraries

5) UPC Productivity
   - Code efficiency
A First Example: Vector addition

//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS
shared int v1[N], v2[N], v1plusv2[N];
void main() {
    int i;
    for(i=0; i<N; i++)
        if (MYTHREAD==i%THREADS)
            v1plusv2[i]=v1[i]+v2[i];
}

2nd Example: A More Efficient Implementation

//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS
shared int v1[N], v2[N], v1plusv2[N];
void main() {
    int i;
    for(i=MYTHREAD; i<N; i+=THREADS)
        v1plusv2[i]=v1[i]+v2[i];
}
3rd Example: A More Convenient Implementation with upc_forall

```
// vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS
shared int v1[N], v2[N], v1plusv2[N];
void main()
{
    int i;
    upc_forall(i=0; i<N; i++; i)
    v1plusv2[i]=v1[i]+v2[i];
}
```

Example: UPC Matrix-Vector Multiplication - Default Distribution

```
// vect_mat_mult.c
#include <upc_relaxed.h>

shared int a[THREADS][THREADS] ;
shared int b[THREADS], c[THREADS] ;
void main (void)
{
    int i, j;
    upc_forall( i = 0 ; i < THREADS ; i++ ; i){
        c[i] = 0;
        for ( j= 0 ; j < THREADS ; j++)
            c[i] += a[i][j]*b[j];
    }
}
Data Distribution

Th. 0  Th. 1  Th. 2
Th. 0  Th. 1  Th. 2
Th. 0  Th. 1  Th. 2
C    A    B

A Better Data Distribution

Th. 0  Th. 1  Th. 2
Th. 0  Th. 1  Th. 2
Th. 0  Th. 1  Th. 2
C    A    B
Example: UPC Matrix-Vector Multiplication - The Better Distribution

```
// vect_mat_mult.c
#include <upc_relaxed.h>

shared [THREADS] int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];

void main (void)
{
    int i, j;
    upc_forall( i = 0 ; i < THREADS ; i++; i){
        c[i] = 0;
        for ( j= 0 ; j < THREADS ; j++)
            c[i] += a[i][j]*b[j];
    }
}
```

Examples of Shared and Private Data Layout:

Assume THREADS = 3

Shared int x; /*x will have affinity to thread 0 */
Shared int y[THREADS];
int z;

will result in the layout:

<table>
<thead>
<tr>
<th></th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y[0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Shared and Private Data**

```plaintext
shared int A[4][THREADS];

will result in the following data layout:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0][0]</td>
<td>A[0][1]</td>
<td>A[0][2]</td>
</tr>
</tbody>
</table>
```

**Shared and Private Data**

```plaintext
shared int A[2][2*THREADS];

will result in the following data layout:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>...</th>
<th>Thread (THREADS-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0][0]</td>
<td>A[0][1]</td>
<td>...</td>
<td>A[0][THREADS-1]</td>
</tr>
<tr>
<td>A[0][THREADS]</td>
<td>A[0][THREADS+1]</td>
<td>...</td>
<td>A[0][2*THREADS-1]</td>
</tr>
</tbody>
</table>
```
Blocking of Shared Arrays

- Default block size is 1
- Shared arrays can be distributed on a block per thread basis, round robin with arbitrary block sizes.
- A block size is specified in the declaration as follows:

```c
shared [block-size] type array[N];
```

- e.g.: `shared [4] int a[16];`

---

Blocking of Shared Arrays

- Block size and THREADS determine affinity
- The term affinity means in which thread’s local shared-memory space, a shared data item will reside
- Element $i$ of a blocked array has affinity to thread:

$$\left\lfloor \frac{i}{\text{blocksize}} \right\rfloor \mod \text{THREADS}$$
Shared and Private Data

- Shared objects placed in memory based on affinity
- Affinity can be also defined based on the ability of a thread to refer to an object by a private pointer
- All non-array shared qualified objects, i.e. shared scalars, have affinity to thread 0
- Threads access shared and private data

Assume THREADS = 4


will result in the following data layout:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[3][1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[3][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Special Operators

- `upc.localsizeof(type-name or expression);` returns the size of the local portion of a shared object
- `upc.blocksizeof(type-name or expression);` returns the blocking factor associated with the argument
- `upc.elemsizeof(type-name or expression);` returns the size (in bytes) of the left-most type that is not an array

Usage Example of Special Operators

```c
typedef shared int sharray[10*THREADS];
sharray a;
char i;

- `upc.localsizeof(sharray) \rightarrow 10*sizeof(int)`
- `upc.localsizeof(a) \rightarrow 10 \times sizeof(int)`
- `upc.localsizeof(i) \rightarrow 1`
- `upc.blocksizeof(a) \rightarrow 1`
- `upc.elemsizeof(a) \rightarrow sizeof(int)`
```
String functions in UPC

- UPC provides standard library functions to move data to/from shared memory
- Can be used to move chunks in the shared space or between shared and private spaces

Equivalent of memcpy:
- `upc_memcpy(dst, src, size)`
  - copy from shared to shared
- `upc_memput(dst, src, size)`
  - copy from private to shared
- `upc_memget(dst, src, size)`
  - copy from shared to private

Equivalent of memset:
- `upc_memset(dst, char, size)`
  - initializes shared memory with a character

The shared block must be a contiguous with all of its elements having the same affinity
UPC Pointers

Where does it point to?

<table>
<thead>
<tr>
<th></th>
<th>Private</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>PP</td>
<td>PS</td>
</tr>
<tr>
<td>Shared</td>
<td>SP</td>
<td>SS</td>
</tr>
</tbody>
</table>

Where does it reside?

- How to declare them?
  ```c
  int *p1;    /* private pointer pointing locally */
  shared int *p2;  /* private pointer pointing into the shared space */
  int *shared p3;  /* shared pointer pointing locally */
  shared int *shared p4; /* shared pointer pointing into the shared space */
  ```

- You may find many using “shared pointer” to mean a pointer pointing to a shared object, e.g. equivalent to p2 but could be p4 as well.
What are the common usages?

- `int *p1; /* access to private data or to local shared data */`
- `shared int *p2; /* independent access of threads to data in shared space */`
- `int *shared p3; /* not recommended */`
- `shared int *shared p4; /* common access of all threads to data in the shared space */`
In UPC pointers to shared objects have three fields:
- thread number
- local address of block
- phase (specifies position in the block)

**Example: Cray T3E implementation**

<table>
<thead>
<tr>
<th>Phase</th>
<th>Thread</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>49</td>
<td>38</td>
</tr>
<tr>
<td>48</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**UPC Pointers**

- Pointer arithmetic supports blocked and non-blocked array distributions
- Casting of shared to private pointers is allowed but not vice versa!
- When casting a pointer-to-shared to a private pointer, the thread number of the pointer-to-shared may be lost
- Casting of a pointer-to-shared to a private pointer is well defined only if the pointed to object has affinity with the local thread
Special Functions

- `size_t upc_threadof(shared void *ptr);`  
  returns the thread number that has affinity to the object pointed to by ptr

- `size_t upc_phaseof(shared void *ptr);`  
  returns the index (position within the block) of the object which is pointed to by ptr

- `size_t upcaddrfield(shared void *ptr);`  
  returns the address of the block which is pointed at by the pointer to shared

- `shared void *upc_resetphase(shared void *ptr);`  
  resets the phase to zero

- `size_t upc_affinitysize(size_t ntotal, size_t nbytes, size_t thr);`  
  returns the exact size of the local portion of the data in a shared object with affinity to a given thread

UPC Pointers

**pointer to shared Arithmetic Examples:**

Assume THREADS = 4

```
#define N 16

shared int x[N];
shared int *dp=&x[5], *dp1;

dp1 = dp + 9;
```
Assume THREADS = 4

shared[3] int x[N], *dp=&x[5], *dp1;

dp1 = dp + 9;
**UPC Pointers**

**Example Pointer Castings and Mismatched Assignments:**

- **Pointer Casting**

  ```
  shared  int x[THREADS];
  int *p;
  p = (int *) &x[MYTHREAD];  /* p points to x[MYTHREAD] */
  ```

  - Each of the private pointers will point at the `x` element which has affinity with its thread, i.e. `MYTHREAD`
UPC Pointers

- Mismatched Assignments

Assume THREADS = 4

shared int x[N];

shared[3] int *dp=&x[5], *dp1;

dp1 = dp + 9;

- The last statement assigns to dp1 a value that is 9 positions beyond dp
- The pointer will follow its own blocking and not that of the array

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[16]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dp</td>
<td>dp + 1</td>
<td>dp + 3</td>
<td>dp + 6</td>
</tr>
<tr>
<td>dp + 2</td>
<td>dp + 4</td>
<td>dp + 7</td>
<td></td>
</tr>
<tr>
<td>dp + 9</td>
<td>dp + 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dp1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
 UPC Pointers

◆ Given the declarations
  
  `shared[3] int *p;`
  `shared[5] int *q;`
  
◆ Then
  
  `p=q; /* is acceptable (an implementation may
  require an explicit cast, e.g. p=(*shared [3])q;) */`
  
◆ Pointer p, however, will follow pointer arithmetic for
  blocks of 3, not 5 !!

◆ A pointer cast sets the phase to 0

---

 UPC Overview

1) UPC in a nutshell
   - Memory model
   - Execution model
   - UPC Systems

2) Data Distribution and Pointers
   - Shared vs Private Data
   - Examples of data distribution
   - UPC pointers

3) Workload Sharing
   - upc_forall

4) Advanced topics in UPC
   - Dynamic Memory Allocation
   - Synchronization in UPC
   - UPC Libraries

5) UPC Productivity
   - Code efficiency
Worksharing with upc_forall

◆ Distributes independent iteration across threads in the way you wish—typically used to boost locality exploitation in a convenient way

◆ Simple C-like syntax and semantics
  
  upc_forall(init; test; loop; affinity)
  
  statement
  
  – Affinity could be an integer expression, or a
  
  – Reference to (address of) a shared object

Work Sharing and Exploiting Locality via upc_forall()

◆ Example 1: explicit affinity using shared references
  
  shared int a[100], b[100], c[100];
  int i;
  upc_forall (i=0; i<100; i++; &a[i])
  
  a[i] = b[i] * c[i];

◆ Example 2: implicit affinity with integer expressions and distribution in a round-robin fashion
  
  shared int a[100], b[100], c[100];
  int i;
  upc_forall (i=0; i<100; i++; i)
  
  a[i] = b[i] * c[i];

Note: Examples 1 and 2 result in the same distribution
Work Sharing: upc forall()

- Example 3: Implicitly with distribution by chunks
  
  ```c
  shared int a[100], b[100], c[100];
  int i;
  upc forall (i=0; i<100; i++; (i*THREADS)/100)
      a[i] = b[i] * c[i];
  ```

- Assuming 4 threads, the following results

<table>
<thead>
<tr>
<th>i</th>
<th>i*THREADS</th>
<th>i*THREADS/100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..24</td>
<td>0..96</td>
<td>0</td>
</tr>
<tr>
<td>25..49</td>
<td>100..196</td>
<td>1</td>
</tr>
<tr>
<td>50..74</td>
<td>200..296</td>
<td>2</td>
</tr>
<tr>
<td>75..99</td>
<td>300..396</td>
<td>3</td>
</tr>
</tbody>
</table>

UPC Overview

1) UPC in a nutshell
   - Memory model
   - Execution model
   - UPC Systems

2) Data Distribution and Pointers
   - Shared vs Private Data
   - Examples of data distribution
   - UPC pointers

3) Workload Sharing
   - upc forall

4) Advanced topics in UPC
   - Dynamic Memory Allocation
   - Synchronization in UPC
   - UPC Libraries

5) UPC Productivity
   - Code efficiency
Dynamic Memory Allocation in UPC

- Dynamic memory allocation of shared memory is available in UPC
- Functions can be collective or not
- A collective function has to be called by every thread and will return the same value to all of them
- As a convention, the name of a collective function typically includes “all”

Collective Global Memory Allocation

```c
shared void *upc_all_alloc
    (size_t nblocks, size_t nbytes);
```

- `nbblocks`: number of blocks
- `nbytes`: block size
- This function has the same result as `upc_global_alloc`. But this is a collective function, which is expected to be called by all threads
- All the threads will get the same pointer
- Equivalent to:
  ```c
  shared [nbytes] char[nblocks * nbytes]
  ```
```c
shared [N] int *ptr;
ptr = (shared [N] int *) upc_all_alloc( THREADS, N*sizeof( int ) );
```

---

**Collective Global Memory Allocation**

<table>
<thead>
<tr>
<th>Thread₀</th>
<th>Thread₁</th>
<th>Thread\textsubscript{THREADS-1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>[N]</td>
<td>[N]</td>
<td>...</td>
</tr>
</tbody>
</table>

- **SHARED**
- **PRIVATE**

```
shared void *upc_global_alloc
   (size_t nblocks, size_t nbytes);

nblocks : number of blocks
nbytes : block size
```

- **Non collective, expected to be called by one thread**
- **The calling thread allocates a contiguous memory region in the shared space**
- **Space allocated per calling thread is equivalent to :**
  - `shared [nbytes] char[nblocks * nbytes]`
- **If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer**
shared [N] int *ptr;
ptr =
    (shared [N] int *)
    upc_global_alloc( THREADS, N*sizeof( int ));

shared [N] int *shared
myptr[THREADS];
myptr[MYTHREAD] =
    (shared [N] int *)
    upc_global_alloc( THREADS, N*sizeof( int ));
Local-Shared Memory Allocation

```c
shared void *upc_alloc (size_t nbytes);
```

- **nbytes**: block size

- Non collective, expected to be called by one thread
- The calling thread allocates a contiguous memory region in the local-shared space of the calling thread
- Space allocated per calling thread is equivalent to:
  - `shared [] char[nbytes]`
- If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer
Local-Shared Memory Allocation

```
shared [] int *ptr;
ptr = (shared [] int *)upc_alloc(N*sizeof( int ));
```

Memory Space Clean-up

```
void upc_free(shared void *ptr);
```

- The `upc_free` function frees the dynamically allocated shared memory pointed to by `ptr`
- `upc_free` is not collective
Given two integer matrices A(NxP) and B(PxM), we want to compute C = A x B.

Entries $c_{ij}$ in C are computed by the formula:

$$c_{ij} = \sum_{l=1}^{P} a_{il} \times b_{lj}$$

---

Example: Matrix Multiplication in UPC

Doing it in C

```c
#include <stdlib.h>

#define N 4
#define P 4
#define M 4
int a[N][P] = {1,2,3,4,5,6,7,8,9,10,11,12,14,14,15,16}, c[N][M];
int b[P][M] = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};

void main (void) {
    int i, j, l;
    for (i = 0 ; i<N ; i++) {
        for (j=0 ; j<M ; j++) {
            c[i][j] = 0;
            for (l = 0 ; l<P ; l++) c[i][j] += a[i][l]*b[l][j];
        }
    }
}
```
Domain Decomposition for UPC

Exploiting locality in matrix multiplication

- A $(N \times P)$ is decomposed row-wise into blocks of size $(N \times P) / \text{THREADS}$ as shown below:

- $B(P \times M)$ is decomposed column-wise into $M / \text{THREADS}$ blocks as shown below:

*Note:* $N$ and $M$ are assumed to be multiples of $\text{THREADS}$

UPC Matrix Multiplication Code

```c
#include <upc_relaxed.h>
#define N 4
#define P 4
#define M 4

shared [N*P / THREADS] int a[N][P];
shared [N*M / THREADS] int c[N][M];
/* a and c are blocked shared matrices, initialization is not
 currently implemented */
shared[M/THREADS] int b[P][M];
void main (void) {
    int i, j, l; // private variables
    upc_forall(i = 0; i<N; i++;
        &c[i][0]) {
        for (j=0; j<M; j++) {
            c[i][j] = 0;
            for (l=0; l<P; l++)
                c[i][j] += a[i][l]*b[l][j];
        }
    }
}
```
UPC Matrix Multiplication Code with Privatization

```c
#include <upc_relaxed.h>
#define N 4
#define P 4
#define M 4

shared [N*P /THREADS] int a[N][P]; // N, P and M divisible by THREADS
shared [N*M /THREADS] int c[N][M];
shared [M/THREADS] int b[P][M];
int *a_priv, *c_priv;

void main (void) {
  int i, j , l; // private variables
  upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
    a_priv = (int *)a[i]; c_priv = (int *)c[i];
    for (j=0 ; j<M ;j++)
      c_priv[j] = 0;
    for (l= 0 ; l<P ; l++)
      c_priv[j] += a_priv[l]*b[l][j];
  }
}
```

UPC Matrix Multiplication Code with block copy

```c
#include <upc_relaxed.h>
shared [N*P /THREADS] int a[N][P];
shared [N*M /THREADS] int c[N][M];
/* a and c are blocked shared matrices, initialization is not currently implemented */
shared [M/THREADS] int b[P][M];
int b_local[P][M];

void main (void) {
  int i, j, l; // private variables
  for( i=0; i<P; i++ )
    for( j=0; j<THREADS; j++ )
      upc_memget(&b_local[i][j*(M/THREADS)],
                 &b[i][j*(M/THREADS)], (M/THREADS)*sizeof(int));
  upc_forall(i = 0 ; i<N ; i++) {
    for (j=0 ; j<M ;j++)
      c[i][j] = 0;
    for (l= 0 ; l<P ; l++) c[i][j] +=a[i][l]*b_local[l][j];
  }
}
```
# UPC Matrix Multiplication Code with Privatization and Block Copy

```c
#include <upc_relaxed.h>

shared [N*P/THREADS] int a[N][P]; // N, P and M divisible by
 // THREADS
shared [N*M/THREADS] int c[N][M];
shared [M/THREADS] int b[P][M];
int *a_priv, *c_priv, b_local[P][M];

void main (void) {
    int i, priv_i, j, l; // private variables
    for (i=0; i<P; i++)
        for (j=0; j<THREADS; j++)
            upc_memget(&b_local[i][j*(M/THREADS)],
                &b[i][j*(M/THREADS)], (M/THREADS)*sizeof(int));
    for (i=0; i<N; i++)
        for (j=0; j<M; j++)
            c_priv[j] = 0;
        for (l=0; l<P; l++)
            c_priv[j] += a_priv[l]*b_local[l][j];
}
```

---

Matrix Multiplication with dynamic memory

```c
#include <upc_relaxed.h>

shared [N*P/THREADS] int *a;
shared [N*M/THREADS] int *c;
shared [M/THREADS] int *b;

void main (void) {
    int i, j, l; // private variables
    a = upc_all_alloc(THREADS,(N*P/THREADS))
        *upc_elmsizeof(*a);
    c = upc_all_alloc(THREADS,(N*M/THREADS))
        *upc_elmsizeof(*c);
    b = upc_all_alloc(P*THREADS,(M/THREADS))
        *upc_elmsizeof(*b));

    upc_forall (i = 0; i<N; i++)
        for (j=0; j<M; j++)
            c[i*M+j] = 0;
        for (l=0; l<P; l++)
            c[i*M+j] += a[i*P+l]*b[l*M+j];
}
```
Example: RandomAccess

Description of the problem:
Let $T$ be a table of size $2^n$ and let $S$ be a table of size $2^m$ filled with random 64-bit integers.

Let $\{A_i\}$ be a stream of 64-bit integers of length $2^{n+2}$ generated by the primitive polynomial over $\text{GF}(2)$, $X_{63}+1$.

For each $a_i$:
$$T[\text{LSB}_{n-1\ldots0}(a_i)] = T[\text{LSB}_{n-1\ldots0}(a_i)] \quad \text{XOR} \quad S[\text{MSB}_{m-1\ldots0}(a_i)]$$

2 Sets of typical problem sizes:
(a) $m=9$, $n=8, 9, \text{max}$
(b) $m$ such as $2^m$ is half of the size of the cache
    $n$ such as $2^n$ is equal to half of the total memory

Objects:
- Stable = substitution table of size $2^m$ 64-bit integers
- Table = data set array of size $2^n$ 64-bit integers

Sequence of 64-bit random numbers as described by Eq 1 of length $\text{UPDATES} = 2^{n+2}$

Operation: Update of $T$
$$T = T \text{ XOR } S$$
RandomAccess – UPC

```c
u64Int Stable[STSIZE]; // private
shared u64Int *Table; // shared

// Table[] allocated dynamically at run-time by:
Table = (shared u64Int *)
    upc_all_alloc(TableSize,sizeof(u64Int));

// STSIZE=2^m where m=9,
// TableSize=2^n where n=4 in this example
...
Table[ran&(TableSize-1)] ^= Stable[ran>>B_SHR];
```

RandomAccess: Computational Core

```c
void RandomAccessUpdate(u64Int TableSize) {
    s64Int i;
    u64Int ran[128], ind;
    int j;
    /* Initialize main table */
    upc_forall ( i=0; i<TableSize; i++; i )
        Table[i] = i;
    upc_barrier;
    for (j=0; j<128; j++)
        ran[j] = starts ((NUPDATE/128) * j);
    for (i=0; i<NUPDATE/128; i++){
        upc_forall ( j=0; j<128; j++; j ){
            ran[j] = (ran[j] << 1) ^ ((s64Int) ran[j] < 0 ? POLY : 0);
            Table[ran[j] & (TableSize-1)] ^= Stable[ran[j] >> (64-LSTSIZE)];
        }
    }
}
```
Expressing Memory Operations Efficiently: Large Histogram (HPCC RA)

MPI/2

- MPI Initialize
- Memory Allocation
- Create memory windows
- Read Input Index
  - Update ITables
  - Decide rank
  - Decide offset
  - MPI_Get
  - Update
  - MPI_Put

UPC

- Memory Allocation
- Read Input Index
- Update Table
- Read from Table[]
- Update
- Write to Table[]

End

Compact Code

<table>
<thead>
<tr>
<th>Random Access (#lines)</th>
<th>%Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>UPC</td>
<td>9.72%</td>
</tr>
<tr>
<td>MPI/2</td>
<td>45.83%</td>
</tr>
<tr>
<td>MPI/1</td>
<td>138.89%</td>
</tr>
</tbody>
</table>

*Study done with HPCC 0.5alpha compliant code*
### Less Conceptual Complexity

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>Function Calls</td>
<td>Keywords with rank and np</td>
<td>MPI Types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>9</td>
<td>35</td>
<td>5</td>
<td>6</td>
<td>81</td>
<td>18</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Notes</td>
<td>11 If 5 For</td>
<td>1 malloc 1 window create 4 for collective operation 4 one-sided 1 fence 3 barriers (1 implicit) mpi_init mpi_finalize mpi_comm_rank mpi_comm_size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>Function Calls</td>
<td>Keywords</td>
<td>UPC Constructs &amp; UPC Types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Notes</td>
<td>3 forall 4 if 1 for</td>
<td>2 shared 1 upc_all_alloc</td>
<td>5 barriers 2 global_exit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>Function Calls</td>
<td>Keywords</td>
<td>UPC Constructs &amp; UPC Types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Notes</td>
<td>1 forall 4 if 1 for</td>
<td>2 shared 1 upc_all_alloc</td>
<td>5 barriers 2 global_exit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>Function Calls</td>
<td>Keywords</td>
<td>UPC Constructs</td>
<td>UPC Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Notes</td>
<td>3 forall 4 if 1 for</td>
<td>2 shared 1 upc_all_alloc</td>
<td>5 barriers 2 global_exit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Synchronization

- **Explicit synchronization with the following mechanisms:**
  - Barriers
  - Locks
  - Memory Consistency Control
  - Fence
Synchronization - Barriers

- No implicit synchronization among the threads
- UPC provides the following barrier synchronization constructs:
  - Barriers (Blocking)
    - `upc_barrier` expr<opt>
  - Split-Phase Barriers (Non-blocking)
    - `upc_notify` expr<opt>
    - `upc_wait` expr<opt>
    - Note: `upc_notify` is not blocking `upc_wait` is

Synchronization - Locks

- In UPC, shared data can be protected against multiple writers:
  - `void upc_lock(upc_lock_t *l)`
  - `int upc_lock_attempt(upc_lock_t *l)` //returns 1 on success and 0 on failure
  - `void upc_unlock(upc_lock_t *l)`

- Locks are allocated dynamically, and can be freed
- Locks are properly initialized after they are allocated
Memory Consistency Models

- Has to do with ordering of shared operations, and when a change of a shared object by a thread becomes visible to others

- Consistency can be strict or relaxed

- Under the relaxed consistency model, the shared operations can be reordered by the compiler / runtime system

- The strict consistency model enforces sequential ordering of shared operations. (No operation on shared can begin before the previous ones are done, and changes become visible immediately)

Memory Consistency- Fence

- UPC provides a fence construct
  - Equivalent to a null strict reference, and has the syntax
    - upc_fence;
    - UPC ensures that all shared references are issued before the upc_fence is completed
Memory Consistency Example

```c
strict shared int flag_ready = 0;
shared int result0, result1;
if (MYTHREAD==0)
    { results0 = expression1;
      flag_ready=1; //if not strict, it could be
      // switched with the above statement }
else if (MYTHREAD==1)
    { while(!flag_ready); //Same note
      result1=expression2+results0;
    }
```  
- We could have used a barrier between the first and second statement in the if and the else code blocks. Expensive!! Affects all operations at all threads.
- We could have used a fence in the same places. Affects shared references at all threads!
- The above works as an example of point to point synchronization.

UPC Libraries

- UPC Collectives
- UPC-IO
Overview UPC Collectives

◆ A collective function performs an operation in which all threads participate

◆ Recall that UPC includes the collectives:
  - upc_barrier, upc_notify, upc_wait, upc_all_alloc, upc_all_lock_alloc

◆ Collectives library include functions for bulk data movement and computation.
  - upc_all_broadcast, upc_all_exchange, upc_all_prefix_reduce, etc.

Overview of UPC-IO

◆ Most UPC-IO functions are collective
  - Function entry/exit includes implicit synchronization
  - Single return values for specific functions

◆ API provided through extension libraries

◆ UPC-IO data operations support:
  - shared or private buffers
  - Blocking (upc_all_fread_shared(), …)
  - Non-blocking (async) operations (upc_all_fread_shared_async(), …)

◆ Supports List-IO Access
◆ Several reference implementations by GWU
File Accessing and File Pointers

List I/O Access using Explicit Offsets

- With Local Buffers
- With Shared Buffers

File I/O with File Pointers

- With Local Buffers
- With Shared Buffers
  - With Individual FP
  - With Common FP

All Read/Write operations have blocking and asynchronous (non-blocking) variants

UPC Overview

1) UPC in a nutshell
   - Memory model
   - Execution model
   - UPC Systems

2) Data Distribution and Pointers
   - Shared vs Private Data
   - Examples of data distribution
   - UPC pointers

3) Workload Sharing
   - upc_forall

4) Advanced topics in UPC
   - Dynamic Memory Allocation
   - Synchronization in UPC
   - UPC Libraries

5) UPC Productivity
   - Code efficiency
Reduced Coding Effort is Not Limited to Random Access– NPB Examples

<table>
<thead>
<tr>
<th></th>
<th>SEQ1</th>
<th>UPC</th>
<th>SEQ2</th>
<th>MPI</th>
<th>UPC Effort (%)</th>
<th>MPI Effort (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPB-CG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#lines</td>
<td>665</td>
<td>710</td>
<td>506</td>
<td>1046</td>
<td>6.77</td>
<td>106.72</td>
</tr>
<tr>
<td>#chars</td>
<td>16145</td>
<td>17200</td>
<td>16485</td>
<td>37501</td>
<td>6.53</td>
<td>127.49</td>
</tr>
<tr>
<td>NPB-EP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#lines</td>
<td>127</td>
<td>183</td>
<td>130</td>
<td>181</td>
<td>44.09</td>
<td>36.23</td>
</tr>
<tr>
<td>#chars</td>
<td>2868</td>
<td>4117</td>
<td>474F</td>
<td>6567</td>
<td>43.55</td>
<td>38.52</td>
</tr>
<tr>
<td>NPB-FT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#lines</td>
<td>575</td>
<td>1018</td>
<td>665</td>
<td>1278</td>
<td>77.04</td>
<td>92.18</td>
</tr>
<tr>
<td>#chars</td>
<td>13090</td>
<td>21672</td>
<td>22188</td>
<td>44348</td>
<td>65.56</td>
<td>99.87</td>
</tr>
<tr>
<td>NPB-IS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#lines</td>
<td>353</td>
<td>528</td>
<td>353</td>
<td>627</td>
<td>49.58</td>
<td>77.62</td>
</tr>
<tr>
<td>#chars</td>
<td>7273</td>
<td>13114</td>
<td>7273</td>
<td>13324</td>
<td>80.31</td>
<td>83.20</td>
</tr>
<tr>
<td>NPB-MG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#lines</td>
<td>610</td>
<td>866</td>
<td>885</td>
<td>1613</td>
<td>41.97</td>
<td>82.26</td>
</tr>
<tr>
<td>#chars</td>
<td>14830</td>
<td>21990</td>
<td>27129</td>
<td>50497</td>
<td>48.28</td>
<td>86.14</td>
</tr>
</tbody>
</table>

\[
UPC_{\text{effort}} = \frac{\#UPC - \#SEQ1}{\#SEQ1} \quad \quad MPI_{\text{effort}} = \frac{\#MPI - \#SEQ2}{\#SEQ2}
\]

SEQ1 is C
SEQ2 is from NAS, all FORTRAN except for IS
Overview

A. Introduction to PGAS (~ 45 mts)

B. Introduction to Languages
   A. UPC (~ 60 mts)
   B. X10 (~ 60 mts)
   C. Chapel (~ 60 mts)

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

D. Hands-On (90 mts)

The X10 Programming Language*
http://x10-lang.org

Vijay Saraswat*

* Winner: 2007 HPCC Award for “Most Productive Research Implementation”

* With thanks to Christoph von Praun, Vivek Sarkar, Nate Nystrom, Igor Peshansky for contributions to slides.

* Please see http://x10-lang.org for most up-to-date version of these slides.
Acknowledgements

X10 Core Team (IBM)
- Shivali Agarwal, Ganesh Bikshandi, Dave Grove, Sreedhar Kodali, Bruce Lucas, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat, Pradeep Varma, Sayantan Sur, Olivier Tarjdieu, Krishna Venkat, Jose Castanos, Ankur Narang

X10 Tools
- Philippe Charles, Robert Fuhrer, Matt Kaplan

Emeritus
- Kemal Ebioglu, Christian Grothoff, Vincent Cave, Lex Spoon, Christoph von Praun, Rajkishore Barik, Chris Donawa, Allan Kielstra, Tong Wen

Research colleagues
- Vivek Sarkar, Rice U
- Satish Chandra, Guojing Cong, Calin Cascaval (IBM)
- Ras Bodik, Guang Gao, Radha Jagadeesan, Jens Palsberg, Rodric Rabbah, Jan Vitek
- Vinod Tippuraju, Jarek Nieplocha (PNNL)
- Kathy Yelick, Dan Bonachea (Berkeley)
- Several others at IBM

Selected Recent Publications
4. “Deadlock-free scheduling of X10 Computations with bounded resources”, SPAA 2007

Tutorials
- Graduate course on X10 at U Pisa (07/07)
- Graduate course at Waseda U (Tokyo, 04/08)

Acknowledgements (contd.)

X10 is an open source project (Eclipse Public License).

Reference implementation in Java, runs on any Java 5 VM:
- Windows/Intel, Linux/Intel
- AIX/PPC, Linux/PPC
- Runs on multiprocessors

X10Flash project — cluster implementation of X10 under development at IBM
- Translation of X10 to Common PGAS Runtime

Website: http://x10-lang.org

Website contains
- Language specification
- Tutorial material
- Presentations
- Download instructions
- Copies of some papers
- Pointers to mailing list

This material is based upon work supported in part by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.
The common architectural landscape

SMP Clusters

Massively Parallel Processor Systems

IBM Blue Gene

Cray XT5

The heterogeneous/accelerated architectural landscape

Cray XT5h: FPGA/Vector-accelerated Opteron

Road Runner: Cell-accelerated Opteron
Example accelerator technologies

From NVidia CUDA Programming Guide

Asynchronous PGAS

- Places may have different computation properties
  - Global SPMD no longer appropriate.
- Use Asynchrony
  - Simple explicitly concurrent model for the user: async (p) S runs statement S “in parallel” at place p
  - Controlled through finish, and local (conditional) atomic

Concurrency is made explicit and programmable.
What is X10?

- X10 is a new language developed in the IBM PERCS project as part of the DARPA program on High Productivity Computing Systems (HPCS)
- X10 is an instance of the APGAS framework in the Java family
- X10
  1. Is more productive than current models,
  2. Can support high levels of abstraction
  3. Can exploit multiple levels of parallelism and non-uniform data access
  4. Is suitable for multiple architectures, and multiple workloads.

X10 is not an “HPC” Language

How is X10 realized?

- **X10**
  - Interprocedural Analysis
  - PIR Analysis & Optimization
  - Classfile Transformations

- **Front End**
  - AST

- **PIR**
  - Parallel Intermediate Representation from IBM and Rice
  - In collaboration with Vivek Sarkar’s group at Rice

- **Annotated Classfiles**

- **XVM**
  - Portable Managed Runtime

- **Planned X10 Tool-chain**

- **Current X10 Tool-chain**

- **Partitioned Code**
  - WCode for xlc
  - Partitioned Code
  - {C/C++/Fortran (restricted code regions for targeting accelerators & high-end computing)}
  - Platform-specific static compiler

- **Partitioned Code**
  - Sequential C, Fortran, Java
PGAS runtime structure

w/ IBM
UPC group

XL UPC
front-end

CAF
front-end

X10
compiler

UPC runtime API

CAF runtime API

X10 API

array
deref
locks
startup
index
assign
deref
update
shutdown

BlueGene
messaging

Myrinet
GM / MX

TCP/IP

sockets

LAPI/
GSM

IB verbs

dummy
(SMP)

memory
allocation,
locks

value
caching

shared
variable
directory

collective
API

startup
shutdown

X10 RoadMap

The

JVM Implementation

v1.7 impl

Initial debugger release

v2.0 impl

v1.7 impl Rel 1

v2.0 impl

v1.7 impl Rel 2

v2.0 impl

Port to PERCS h/w

Advanced
debugger features

Design of X10 Implementation for PERCS

Multi-process debugger

SSCA#2, UTS

Other Apps (TBD)

v1.7 spec

v2.0 spec

06/08

12/08

06/09

12/09

6/10

12/10

External milestones

Internal milestones

Libraries, APIs, Tools, user trials

Trial at Rice U

X10DT

enhancements

Concur. refactorings

More refactorings

Language Definition

v1.7 spec

v2.0 spec

JVM Implementation

X10 Flash C/C++ Implementation

v1.7 impl

v2.0 impl

v2.0 impl

v2.0 impl
Quick Language Review

Language goals

- **Simple**
  - Start with a well-accepted programming model, build on strong technical foundations, add few core constructs

- **Safe**
  - Eliminate possibility of errors by design, and through static checking

- **Powerful**
  - Permit easy expression of high-level idioms
  - And Permit expression of high-performance programs

- **Scalable**
  - Support high-end computing with millions of concurrent tasks

- **Universal**
  - Present one core programming model to abstract from the current plethora of architectures.
Overview of Features

- A lot of sequential features of Java inherited unchanged
  - Classes (w/ single inheritance)
  - Interfaces, (w/ multiple inheritance)
  - Instance and static fields
  - Constructors, (static) initializers
  - Overloaded, over-rideable methods
  - Garbage collection

- Value classes
- Closures
- Points, Regions, Distributions, Arrays

- Substantial extensions to the type system
  - Dependent types
  - Generic types
  - Function types
  - Type definitions, inference

- Concurrency
  - Fine-grained concurrency:
    - async (p,l) S
  - Atomicity
    - atomic (s)
  - Ordering
    - L: finish S
  - Data-dependent synchronization
    - when (c) S

Value and reference classes

- Value classes
  - All fields of a value class are final
  - A variable of value class type is never null
  - “primitive” types are value classes: Boolean, Int, Char, Double, ...
  - Instances of value classes may be freely copied from place to place

- Reference classes
  - May have mutable fields
  - May be null
  - Only references to instances may be communicated between places (RemoteRefs)
Points and Regions

A point is an element of an n-dimensional Cartesian space \( (n\geq 1) \) with integer-valued coordinates e.g., \([5]\), \([1, 2]\), ...

A point variable can hold values of different ranks e.g.,
- \( \text{var } p: \text{Point} = [1]; p = [2,3]; ... \)

Operations
- \( p1\text{.rank} \)
  - returns rank of point \( p1 \)
- \( p1(i) \)
  - returns element \((i \mod p1\text{.rank})\) if \( i < 0 \) or \( i \geq p1\text{.rank} \)
- \( p1 < p2, p1 \leq p2, p1 > p2, p1 \geq p2 \)
  - returns true iff \( p1 \) is lexicographically \(<, \leq, >, \geq\) \( p2 \)
  - only defined when \( p1\text{.rank} \) and \( p1\text{.rank} \) are equal

Regions are collections of points of the same dimension
- Rectangular regions have a simple representation, e.g. \([1..10, 3..40]\)
- Rich algebra over regions is provided

Regions are collections of points of the same dimension
- Rectangular regions have a simple representation, e.g. \([1..10, 3..40]\)
- Rich algebra over regions is provided

Distributions and Arrays

- Distributions specify mapping of points in a region to places
  - E.g. Dist.makeBlock(R)
  - E.g. Dist.unique()

- Arrays are defined over a distribution and a base type
  - A:Array[T]
  - A:Array[T](d)

- Arrays are created through initializers
  - Array.make[T](d, init)

- Arrays may be immutable

- Arrays operations
  - A.rank ::= # dimensions in array
  - A.region ::= index region (domain) of array
  - A.dist ::= distribution of array A
  - A(p) ::= element at point p, where p belongs to A.region
  - A(R) ::= restriction of array onto region R
    - Useful for extracting subarrays
## Generic classes

```java
generic class Rail[T] [length: int]
  implements Indexable[int, T],
  Settable[int, T]
{
  private native def this(n): Rail[T][length==n];
  public native def get(i): T;
  public native def apply(i): T;
  public native def set(v, i): void;
}
```

- Classes and interfaces may have type parameters
- class `Rail[T]`
  - Defines a type constructor `Rail`
  - and a family of types `Rail[int], Rail[String], Rail[Object], Rail[C], ...`

- `Rail[C]`: as if `Rail` class is copied and `C` substituted for `T`

- Can instantiate on any type, including primitives (e.g., `int`)

## Dependent Types

- Classes have properties
  - public final instance fields
    - class `Region(rank: int, zeroBased: boolean, rect: boolean)` { ... }
  - Can constrain properties with a boolean expression
    - `Region{rank==3}`
      - type of all regions with rank 3
    - `Array[int]{region==R}`
      - type of all arrays defined over region `R`
      - `R` must be a constant or a final variable in scope at the type

- Dependent types are checked statically.
- Runtime casts are also permitted
  - Requires run-time constraint checking/solving

- Dependent type system is extensible

- See OOPSLA 08 paper.
Function Types

- (T1, T2, ..., Tn) => U
  - type of functions that take arguments Ti and returns U
- If f: (T) => U and x: T
- then invoke with f(x): U
- Function types can be used as an interface
  - Define apply method with the appropriate signature:

Closures

- First-class functions
  - (x: T): U => e
  - used in array initializers:
    - Array.make[int][ 0..4, (p: point) => p(0)*p(0) ]
      - the array [ 0, 1, 4, 9, 16 ]

Operators

- int.+, boolean.&, ...
- sum =
  a.reduce(int.+(int,int), 0)

Type inference

- Field, local variable types inferred from initializer type
  - val x = 1;  /* x has type int{self==1} */
  - val y = 1..2;  /* y has type Region{rank==1} */

- Method return types inferred from method body
  - def m() { ... return true ... return false ... }
    /* m has return type boolean */

- Loop index types inferred from region
  - R: Region{rank==2}
  - for (p in R) { ... /* p has type Point{rank==2} */ }

- Proposed:
  - Inference of place types for asyncs (cf PPoPP 08 paper)
async

Stmt ::= async(p,l) Stmt

◆ Creates a new child activity that executes statement S
cf Cilk’s spawn
◆ Returns immediately
◆ S may reference final variables in enclosing blocks
◆ Activities cannot be named
◆ Activity cannot be aborted or cancelled

```java
void run() {
    if (r < 2) return;
    final Fib f1 = new Fib(r-1),
                f2 = new Fib(r-2);
    finish {
        async f1.run();
        f2.run();
    }
    result = f1.r + f2.r;
}
```

finish

Stmt ::= finish Stmt

L:finish S
◆ Execute S, but wait until all (transitively) spawned asyncs have terminated.
cf Cilk’s sync
Rooted exception model
◆ Trap all exceptions thrown by spawned activities.
◆ Throw an (aggregate) exception if any spawned async terminates abruptly.
◆ implicit finish at main activity

finish is useful for expressing “synchronous” operations on (local or) remote data.
atomic

- Atomic blocks are conceptually executed in a single step while other activities are suspended: isolation and atomicity.

- An atomic block ...
  - must be nonblocking
  - must not create concurrent activities (sequential)
  - must not access remote data (local)

    Stmt ::= atomic Statement
    MethodModifier ::= atomic

when

- when (E) S
  - Activity suspends until a state in which the guard E is true.
  - In that state, S is executed atomically and in isolation.

- Guard E
  - boolean expression
  - must be nonblocking
  - must not create concurrent activities (sequential)
  - must not access remote data (local)
  - must not have side-effects (const)

- await (E)
  - syntactic shortcut for when (E);

    class OneBuffer {
      var datum:Object = null;
      var filled:Boolean = false;

      def send(v:Object) {
        when ( ! filled ) {
          datum = v;
          filled = true;
        }
      }

      def receive():Object {
        when ( filled ) {
          val v = datum;
          datum = null;
          filled = false;
          return v;
        }
      }
    }

    Stmt ::= WhenStmt
    WhenStmt ::= when ( Expr ) Stmt | WhenStmt or (Expr) Stmt
**Clocks: Motivation**

- Activity coordination using **finish** is accomplished by checking for activity termination.
- But in many cases activities have a producer-consumer relationship and a "barrier"-like coordination is needed without waiting for activity termination.
  - The activities involved may be in the same place or in different places.
- **Design clocks to offer determinate and deadlock-free coordination between a dynamically varying number of activities.**

![Diagram of activity phases and clocks]

**Clocks – Main operations**

```javascript
var c = Clock.make();

- Allocate a clock, register current activity with it. Phase 0 of c starts.

- Create async activities registered on clocks c1, c2, ...

- Nonblocking operation that signals completion of work by current activity for this phase of clock c

- Barrier --- suspend until all clocks that the current activity is registered with can advance. **c.resume()** is first performed for each such clock, if needed.

- Next can be viewed like a “finish” of all computations under way in the current phase of the clock.
Fundamental X10 Property

Programs written using async, finish, atomic, clock **cannot** deadlock

X10 Programming Idioms
**Random Access (LOC=79)**

Core algorithm:

```java
static RAUpdate(logLocalTableSize:Int, Table: Array[Long]{rail}){
    finish ateach((p) in UNIQUE) {
        val localTableSize=1<<logLocalTableSize,
        TableSize=localTableSize*NUM_PLACES,
        mask=TableSize-1,
        NumUpdates=4*localTableSize;
        var ran:Long =HPCC_starts(p*NumUpdates);
        for (var i:Long=0; i<NumUpdates; i++) {
            val temp=ran;
            val index = (temp & mask) to Int;
            async
            atomic Table(index) ^= temp;
            ran = (ran << 1)^((long) ran < 0 ? POLY: 0);
        }
    }
}
```

**SPMD + Remote atomic operations (X10Flash Implementation)**

**Performance**

- **Random Access performance on BG/W**
- **Random Access on Power5 SMP Cluster**

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>UPC</th>
<th>X10</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.58</td>
<td>0.58</td>
<td>0.001342</td>
</tr>
<tr>
<td>2</td>
<td>1.15</td>
<td>1.04</td>
<td>0.007755</td>
</tr>
<tr>
<td>4</td>
<td>2.28</td>
<td>1.31</td>
<td>0.031426</td>
</tr>
<tr>
<td>8</td>
<td>4.49</td>
<td>2.51</td>
<td>0.004144</td>
</tr>
<tr>
<td>16</td>
<td>8.83</td>
<td>4.51</td>
<td>0.052395</td>
</tr>
<tr>
<td>32</td>
<td>14.8</td>
<td></td>
<td>0.055678</td>
</tr>
<tr>
<td>64</td>
<td>28.3</td>
<td></td>
<td>0.094022</td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td>0.143919</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td>0.368779</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td>0.398779</td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td></td>
<td>0.398779</td>
</tr>
</tbody>
</table>
Depth-first search

class V(index:Int) {
  var parent:V;
  var neighbors: Rail[V];
  def this(i:int):V(i){property(i);}
  public void compute(): void {
    for (int k=0; k < neighbors.length; k++) {
      val v = neighbors[k];
      atomic v.parent=(v.parent==null?this:v.parent);
      if (v.parent==this)
        async clocked (c) {next; v.compute();} }
  }
  public computeTree(): void {finish compute();}
  ...
}

Adaptive stealing -- Code

def compute(w:Worker):Void throws StealAbort {
  w.popAndReturnFrame();
  val newList:List = null, newLength:Int = 0;
  var oldList:V = this, par:V = parent, batchSize:Int=0;
  do {
    val v = oldList, edges = v.neighbors;
    oldList = v.next;
    for (var k:Int = 0; k < edges.length; ++k) {
      val e = edges[k];
      if (e != null && e.level == 0 &&
        UPDATER.compareAndSet(e,0,1)) {
        e.parent = par; e.next = newList; newList=e;
        if (batchSize=0) {
          val s=w.getLocalQueueSize();
          batchSize=(s <1)? 1:(s>=LOG_MAX? 1<<LOG_MAX: 1<<s);
        }
      }
    }
  } while (oldList != null);
Torus on moxie: DFS vs BFS

KGraph on Moxie: BFS vs DFS
FT Code (LOC=137)

Key routine: global transpose

```java
finish ateach((p) in UNIQUE) {
    val numLocalRows = SQRTN/NUM_PLACES;
    int rowStartA = p*numLocalRows; // local transpose
    val block = [0..numLocalRows-1,0..numLocalRows-1];
    for ((k) in 0..NUM_PLACES-1) { //for each block
        int colStartA = k*numLocalRows;
        ... transpose locally ...
        for ((i) in 0..numLocalRows-1) {
            val srcIdx = 2*((rowStartA + i)*SQRTN+colStartA),
            destIdx = 2*(SQRTN * (colStartA + i) + rowStartA);
            async (UNIQUE(k))
            Runtime.arrayCopy(Y, srcIdx, Z, destIdx, 2*numLocalRows);
        }
    }
}
```

Communication/computation overlap across multiple nodes (X10Flash implementation)

FT Performance

32 nodes (16-way Power 5+, 1.9GHz, 64GB)

BG/W – X10 uses g++. Now running for more racks.
(UPC == 51 GF/s for one rack.)
Core algorithm:

```java
void run() {
    finish foreach (point [pi,pj] : [0:px-1,0:py-1]) {
        val startY=0, startX = new Rail.make[Int](ny);
        val myBlocks=A[pord(pi,pj)].z;
        while(startY < ny) {
            bvar done: boolean =false;
            for (var j:Int=startY; j < min(startY+LOOK_AHEAD, ny) & & !done; ++j) {
                for (var i:Int=startX(j); i < nx; ++i) {
                    val b = myBlocks[lord(i,j)];
                    if (b.ready) {
                        if (i==startX[j]) startX[j]++;
                    } else done |= b.step(startY, startX);
                }
                if (startX(startY)==nx) { startY++;
            }
        }
    }
}
```

Communication/computation overlap across multiple nodes (X10Flash implementation)

Example of steps

```java
def step(val startY:Int, startX:Rail[Int]):Boolean {
    visitCount++;
    if (count==maxCount) {
    } else {
        val IBuddy=getBlock(I, count);
        if (!IBuddy.ready) return false;
        val JBuddy=getBlock(count,J);
        if (!JBuddy.ready) return false;
        mulsub(IBuddy, JBuddy);
        count++;
        return true;
    }
}
```

Call BLAS for DGEMM.

```
stepIltJ \rightarrow \text{wait; backsolve}
stepIeqJ \rightarrow \text{wait; control panel LU factorization}
stepIgtJ \rightarrow \text{wait; compute lower, participate in LU factorization}
```

Communication/computation overlap across multiple nodes (X10Flash implementation)
**LU Performance (SMP only)**

**IBM J9 JVM (64-bit)**

Additional work presented at http://x10-lang.org
## Overview

A. Introduction to PGAS (~ 45 mts)

B. Introduction to Languages
   - A. UPC (~ 60 mts)
   - B. X10 (~ 60 mts)
   - C. Chapel (~ 60 mts)

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

D. Hands-On (90 mts)

---

# Chapel

the Cascade High Productivity Language

Brad Chamberlain  
Cray Inc.

SC08: Tutorial M04 – 11/17/08
Chapel

_Chapel_: a new parallel language being developed by Cray Inc.

Themes:
- **general parallel programming**
  - data-, task-, and nested parallelism
  - express general levels of software parallelism
  - target general levels of hardware parallelism
- **global-view abstractions**
- **multiresolution design**
- **control of locality**
- **reduce gap between mainstream & parallel languages**

---

Chapel’s Setting: HPCS

**HPCS**: High Productivity Computing Systems (DARPA et al.)
- **Goal**: Raise HEC user productivity by $10 \times$ for the year 2010
- **Productivity** = Performance
  - + Programmability
  - + Portability
  - + Robustness

- **Phase II**: Cray, IBM, Sun (July 2003 – June 2006)
  - Evaluated the entire system architecture’s impact on productivity…
    - processors, memory, network, I/O, OS, runtime, compilers, tools, …
  - …and new languages:
    - Cray: Chapel
    - IBM: X10
    - Sun: Fortress

- **Phase III**: Cray, IBM (July 2006 – 2010)
  - Implement the systems and technologies resulting from phase II
  - (Sun also continues work on Fortress, without HPCS funding)
Chapel and Productivity

Chapel’s Productivity Goals:

- vastly improve programmability over current languages/models
  - writing parallel codes
  - reading, modifying, porting, tuning, maintaining, evolving them
- support performance at least as good as MPI
  - competitive with MPI on generic clusters
  - better than MPI on more capable architectures
- improve portability compared to current languages/models
  - as ubiquitous as MPI, but with fewer architectural assumptions
  - more portable than OpenMP, UPC, CAF, …
- improve code robustness via improved semantics and concepts
  - eliminate common error cases altogether
  - better abstractions to help avoid other errors

Outline

✓ Chapel Context

➢ Terminology: Global-view & Multiresolution Prog. Models

❓ Language Overview

❓ Status, Future Work, Collaborations
Parallel Programming Model Taxonomy

programming model: the mental model a programmer uses when coding using a language, library, or other notation

fragmented models: those in which the programmer writes code from the point-of-view of a single processor/thread

global-view models: those in which the programmer can write code that describes the computation as a whole

Global-view vs. Fragmented

Problem: “Apply 3-pt stencil to vector”

```plaintext
( [ ] ) + [ ] / 2 = [ ]
```

```plaintext
[ ] [ ] [ ] [ ]
```

```
[ ] [ ] [ ] [ ]
```

```
[ ] [ ] [ ] [ ]
```
Global-view vs. Fragmented

**Problem:** “Apply 3-pt stencil to vector”

<table>
<thead>
<tr>
<th>Global-view</th>
<th>Fragmented</th>
</tr>
</thead>
</table>
| \[
\begin{align*}
&= \text{vector } a_0 + \text{vector } a_1 + \text{vector } a_2 \\
&= \frac{1}{2} (\text{vector } a_0 + \text{vector } a_1 + \text{vector } a_2)
\end{align*}
\] | \[
\begin{align*}
&= \text{vector } a_0 + \text{vector } a_1 + \text{vector } a_2 \\
&= \frac{1}{2} (\text{vector } a_0 + \text{vector } a_1 + \text{vector } a_2)
\end{align*}
\] |

Parallel Programming Model Taxonomy

**programming model:** the mental model a programmer uses when coding using a language, library, or other notation

**fragmented models:** those in which the programmer writes code from the point-of-view of a single processor/thread

**SPMD models:** Single-Program, Multiple Data -- a common fragmented model in which the user writes one program & runs multiple copies of it, parameterized by a unique ID

**global-view models:** those in which the programmer can write code that describes the computation as a whole
Global-view vs. SPMD Code

**Problem:** “Apply 3-pt stencil to vector”

**Global-view**

```chapel
def main() {
    var n: int = 1000;
    var a, b: [1..n] real;
    forall i in 2..n-1 {
        b(i) = (a(i-1) + a(i+1))/2;
    }
}
```

**SPMD**

```chapel
def main() {
    var n: int = 1000;
    var locN: int = n/numProcs;
    var a, b: [0..locN+1] real;
    if (iHaveRightNeighbor) {
        send(right, a(locN));
        recv(right, a(locN+1));
    } else {
        innerHi = locN-1;
    }
    if (iHaveLeftNeighbor) {
        send(left, a(1));
        recv(left, a(0));
    } else {
        innerLo = 2;
    }
    forall i in innerLo..innerHi {
        b(i) = (a(i-1) + a(i+1))/2;
    }
}
```

Assumes `numProcs` divides `n`; a more general version would require additional effort.
MPI SPMD pseudo-code

Problem: “Apply 3-pt stencil to vector”

SPMD (pseudocode + MPI)

```plaintext
var n: int = 1000, locN: int = n/numProcs;
var a, b: [0..locN+1] real;
var innerLo: int = 1, innerHi: int = locN;
var numProcs, myPE: int;
var retval: int;
var status: MPI_Status;

MPI_Comm_size( MPI_COMM_WORLD, &numProcs);
MPI_Comm_rank( MPI_COMM_WORLD, &myPE);
if (myPE < numProcs-1) {
    retval = MPI_Send( &(a(locN)), 1, MPI_FLOAT, myPE+1, 0, MPI_COMM_WORLD);
    if (retval != MPI_SUCCESS) { handleError(retval); }
    retval = MPI_Recv( &(a(locN+1)), 1, MPI_FLOAT, myPE+1, 1, MPI_COMM_WORLD, &status);
    if (retval != MPI_SUCCESS) { handleErrorWithStatus(retval, status); }
} else
    innerHi = locN-1;

if (myPE > 0) {
    retval = MPI_Send( &(a(1)), 1, MPI_FLOAT, myPE-1, 1, MPI_COMM_WORLD);
    if (retval != MPI_SUCCESS) { handleError(retval); }
    retval = MPI_Recv( &(a(0)), 1, MPI_FLOAT, myPE-1, 0, MPI_COMM_WORLD, &status);
    if (retval != MPI_SUCCESS) { handleErrorWithStatus(retval, status); }
} else
    innerLo = 2;
forall i in (innerLo..innerHi) {
    b(i) = (a(i-1) + a(i+1))/2;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays

rprj3 stencil from NAS MG

```
= w0 + w1 + w2 + w3
```

Brad Chamberlain, Cray Inc.
NAS MG \textit{rprj3} stencil in Fortran + MPI

\begin{verbatim}
if( axis .eq.  1 )then
  buff_id = 2 + dir
endif

if( dir .eq. -1 )then
  buff_len = buff_len + 1
  buff(buff_len,buff_id ) = u( i1,1,i3)
endif
endif
endif

endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endi
Summarizing Fragmented/SPMD Models

- **Advantages:**
  - fairly straightforward model of execution
  - relatively easy to implement
  - reasonable performance on commodity architectures
  - portable/ubiquitous
  - lots of important scientific work has been accomplished with them

- **Disadvantages:**
  - blunt means of expressing parallelism: cooperating executables
  - fails to abstract away architecture / implementing mechanisms
  - obfuscates algorithms with many low-level details
    - error-prone
    - brittle code: difficult to read, maintain, modify, experiment
    - “MPI: the assembly language of parallel computing”

---

Current HPC Programming Notations

- **communication libraries:**
  - MPI, MPI-2
  - SHMEM, ARMCI, GASNet

- **shared memory models:**
  - OpenMP, pthreads

- **PGAS languages:**
  - Co-Array Fortran
  - UPC
  - Titanium

- **HPCS languages:**
  - Chapel
  - X10 (IBM)
  - Fortress (Sun)
Parallel Programming Models: Two Camps

- Expose Implementing Mechanisms
  - Target Machine
  - MPI
  - OpenMP
  - pthreads
  - ZPL
  - HPF
  - Higher-Level Abstractions

"Why is everything so painful?"

"Why do my hands feel tied?"

Multiresolution Language Design

**Our Approach:** Permit the language to be utilized at multiple levels, as required by the problem/programmer
- provide high-level features and automation for convenience
- provide the ability to drop down to lower, more manual levels
- use appropriate separation of concerns to keep these layers clean

**Language Concepts**
- Data Parallelism
- Distributions
- Task Parallelism
- Base Language
- Locality Control

**Task Scheduling**
- Stealable Tasks
- Suspendable Tasks
- Run to Completion
- Thread per Task

**Memory Management**
- Garbage Collection
- Region-based
- Alloc/Free

Brad Chamberlain, Cray Inc.
Outline

✓ Chapel Context
✓ Terminology: Global-view & Multiresolution Prog. Models

➢ Language Overview
  • Base Language
  • Parallel Features
    • task parallel
    • data parallel
  • Locality Features

☐ Status, Future Work, Collaborations

Base Language: Design

▪ Block-structured, imperative programming
▪ Intentionally not an extension to an existing language
▪ Instead, select attractive features from others:
  - **ZPL, HPF**: data parallelism, index sets, distributed arrays
    (see also APL, NESL, Fortran90)
  - **Cray MTA C/fortran**: task parallelism, lightweight synchronization
  - **CLU**: iterators (see also Ruby, Python, C#)
  - **ML**: latent types (see also Scala, Matlab, Perl, Python, C#)
  - **Java, C#**: OOP, type safety
  - **C++**: generic programming/templates (without adopting its syntax)
  - **C, Modula, Ada**: syntax
Base Language: Standard Stuff

- Lexical structure and syntax based largely on C/C++
  
  ```chapel
  { a = b + c; foo(); } // no surprises here
  ```

- Reasonably standard in terms of:
  - scalar types
  - constants, variables
  - operators, expressions, statements, functions

- Support for object-oriented programming
  - value- and reference-based classes
  - no strong requirement to use OOP

- Modules for namespace management

- Generic functions and classes

Base Language: Departures

- **Syntax:** declaration syntax differs from C/C++
  
  ```chapel
  var <varName> [: <definition>] [= <init>];
  def <fnName>(<argList>)[: <returnType>] { … }
  ```

- **Types**
  - support for complex, imaginary, string types
  - sizes more explicit than in C/C++ (e.g., `int(32)`, `complex(128)`)
  - richer array support than C/C++, Java, even Fortran
  - no pointers (apart from class references)

- **Operators**
  - casts via `:'` (e.g., `3.14: int(32)`)
  - exponentiation via `'*'` (e.g., `2**n`)

- **Statements:** for loop differs from C/C++
  
  ```chapel
  for <indices> in <iterationSpace> { … } 
  ```

  *e.g.,* for `i in 1..n { … }`

- **Functions:** argument-passing semantics
Base Language: My Favorite Departures

- Rich compile-time language
  - parameter values (compile-time constants)
  - folded conditionals, unrolled for loops, expanded tuples
  - type and parameter functions – evaluated at compile-time
- Latent types:
  - ability to omit type specifications for convenience or reuse
  - type specifications can be omitted from...
    - variables (inferred from initializers)
    - class members (inferred from constructors)
    - function arguments (inferred from callsite)
    - function return types (inferred from return statements)
- Configuration variables (and parameters)
  \[\text{config const } n = 100; \quad \text{// override with --n=100000}\]

Tuples
Iterators...

Base Language: Motivation for Iterators

<table>
<thead>
<tr>
<th>Given a program with a bunch of similar loops...</th>
<th>Consider the effort to convert them from RMO to CMO...</th>
<th>Or to tile the loops...</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;m; i++) { for (j=0; j&lt;n; j++) { _A(i,j)... } } } } }</td>
<td>for (j=0; j&lt;n; j++) { for (i=0; i&lt;m; i++) { _A(i,j)... } } } } }</td>
<td>for (jj=0; jj&lt;n; jj+=blocksize) { for (ii=0; ii&lt;m; ii+=blocksize) { for (j=jj; j&lt;\min(m,jj+blocksize-1) { for (i=ii; i&lt;\min(n,ii+blocksize-1) { _A(i,j)... } } } } }</td>
</tr>
<tr>
<td>... for (i=0; i&lt;m; i++) { for (j=0; j&lt;n; j++) { _A(i,j)... } } } } }</td>
<td>... for (j=0; j&lt;n; j++) { for (i=0; i&lt;m; i++) { _A(i,j)... } } } } }</td>
<td>... for (jj=0; jj&lt;n; jj+=blocksize) { for (ii=0; ii&lt;m; ii+=blocksize) { for (j=jj; j&lt;\min(m,jj+blocksize-1) { for (i=ii; i&lt;\min(n,ii+blocksize-1) { _A(i,j)... } } } } }</td>
</tr>
</tbody>
</table>

Brad Chamberlain, Cray Inc.
Base Language: Motivation for Iterators

Given a program with a bunch of similar loops...

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    ...A(i,j)...
  }
}
```

Consider the effort to convert them from RMO to CMO...

```
for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    ...A(i,j)...
  }
}
```

Or to tile the loops...

```
for (jj=0; jj<n; jj+=blocksize) {
  for (ii=0; ii<m; ii+=blocksize) {
    for (j=jj; j<min(m,jj+blocksize-1) {
      for (i=ii; i<min(n,ii+blocksize-1) {
        ...A(i,j)...
      }
    }
  }
}
```

Or to change the iteration order over the tiles...

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    ...A(i,j)...
  }
}
```

Or to make them into fragmented loops for an MPI program...

```
for (jj=0; jj<n; jj+=blocksize) {
  for (ii=0; ii<m; ii+=blocksize) {
    for (j=jj; j<min(m,jj+blocksize-1) {
      for (i=ii; i<min(n,ii+blocksize-1) {
        ...A(i,j)...
      }
    }
  }
}
```

We wouldn't program straight-line code this way, so why are we so tolerant of our lack of loop abstractions?

Base Language: Iterators

- like functions, but *yield* a number of elements one-by-one:
  ```
  iterator RMO() { 
    for i in 1..m do 
      for j in 1..n do
        yield (i,j);
  }
  ```
  ```
  iterator tiled(blocksize) {
    for ii in 1..m by blocksize do 
      for jj in 1..n by blocksize do 
        for i in ii..min(n, ii+blocksize-1) do 
          for j in jj..min(m, jj+blocksize-1) {
            yield (i,j);
          }
    }
  }
  ```

- iterators are used to drive loops:
  ```
  for ij in RMO() {
    ...A(ij)...
  }
  ```
  ```
  for ij in tiled(blocksize) {
    ...A(ij)...
  }
  ```

- as with functions...
  ```
  ...one iterator can be redefined to change the behavior of many loops
  ```
  ```
  ...a single invocation can be altered, or its arguments can be changed
  ```

- not necessarily any more expensive than in-line loops
Task Parallelism: Task Creation

`begin`: creates a task for future evaluation

```chapel
begin DoThisTask();
WhileContinuing();
TheOriginalThread();
```

`sync`: waits on all begins created within a dynamic scope

```chapel
sync {
    begin recursiveTreeSearch(root);
}
```

Task Parallelism: Task Coordination

`sync variables`: store full/empty state along with value

```chapel
var result$: sync real; // result is initially empty
sync {
    begin _ = result$;    // block until full, leave empty
    begin result$ = _;    // block until empty, leave full
}
result$.readFF(); // read when full, leave full;
// other variations also supported
```

`single-assignment variables`: writable once only

```chapel
var result$: single real = begin f(); // result initially empty
... // do some other things
total += result$; // block until f() has completed
```

`atomic sections`: support transactions against memory

```chapel
atomic {
    newnode.next = insertpt;
    newnode.prev = insertpt.prev;
    insertpt.prev.next = newnode;
    insertpt.prev = newnode;
}
```
Task Parallelism: Structured Tasks

cobegin: creates a task per component statement:

```chapel
computePivot(lo, hi, data);

cobegin {
    Quicksort(lo, pivot, data);
    Quicksort(pivot, hi, data);
} // implicit join here
```

coforall: creates a task per loop iteration

```chapel
coforall e in Edges {
    exploreEdge(e);
} // implicit join here
```

Producer/Consumer example

```chapel
var buff$: [0..buffersize-1] sync int;

cobegin {
    producer();
    consumer();
}

def producer() {
    var i = 0;
    for ... {
        i = (i+1) % buffersize;
        buff$(i) = ...;
    }
}

def consumer() {
    var i = 0;
    while ... {
        i = (i+1) % buffersize;
        ...buff$(i)...;
    }
```
Domains

*domain*: a first-class index set

```chapel
var m = 4, n = 8;
var D: domain(2) = [1..m, 1..n];
```

```chapel
var m = 4, n = 8;
var D: domain(2) = [1..m, 1..n];
var Inner: subdomain(D) = [2..m-1, 2..n-1];
```
Domains: Some Uses

- Declaring arrays:
  \[
  \text{var A, B: [D] real;}
  \]

- Iteration (sequential or parallel):
  \[
  \text{for ij in Inner { ... }}
  \]
  or:
  \[
  \text{forall ij in Inner { ... }}
  \]
  or:
  
- Array Slicing:
  \[
  A[\text{Inner}] = B[\text{Inner}];
  \]

- Array reallocation:
  \[
  D = [1..2*m, 1..2*n];
  \]

---

Data Parallelism: Domains

*domains*: first-class index sets, whose indices can be...

...integer tuples...

...dense... ...strided... ...sparse...

...graphs... ...associative... ...or arbitrary values.
Data Parallelism: Domain Declarations

```chapel
var DnsDom: domain(2) = [1..10, 0..24],
StrDom: subdomain(DnsDom) = DnsDom by (2,4),
SpsDom: subdomain(DnsDom) = genIndices();
```

```
var GrphDom: domain(opaque),
NameDom: domain(string) = readNames();
```

---

Data Parallelism: Domains and Arrays

Domains are used to declare arrays...

```chapel
var DnsArr: [DnsDom] complex,
SpsArr: [SpsDom] real;
...
```

---

Brad Chamberlain, Cray Inc.
Data Parallelism: Domain Iteration

...to iterate over index spaces...

forall \( ij \) in StrDom {
  DnsArr(ij) += SpsArr(ij);
}

Data Parallelism: Array Slicing

...to slice arrays...

DnsArr[StrDom] += SpsArr[StrDom];
Data Parallelism: Array Reallocation

...and to reallocate arrays

```chapel
StrDom = DnsDom by (2,2);
SpsDom += genEquator();
```

Locality: Locales

**locale:** architectural unit of locality
- has capacity for processing and storage
- threads within a locale have ~uniform access to local memory
- memory within other locales is accessible, but at a price
- *e.g.*, a multicore processor or SMP node could be a locale
Locality: Locales

- user specifies # locales on executable command-line
  
  ```prompt> myChapelProg -nl=8```

- Chapel programs have built-in locale variables:
  
  ```config const numLocales: int;
  const LocaleSpace = [0..numLocales-1],
  Locales: [LocaleSpace] locale;
  0 1 2 3 4 5 6 7```

- Programmers can create their own locale views:
  
  ```var CompGrid = Locales.reshape([1..GridRows, 1..GridCols]);
  var TaskALocs = Locales[..numTaskALocs];
  var TaskBLocs = Locales[numTaskALocs+1..];```

 Locality: Task Placement

**on clauses:** indicate where tasks should execute

Either in a data-driven manner...

```computePivot(lo, hi, data);
cobegin {
  on A(lo) do Quicksort(lo, pivot, data);
  on A(pivot) do Quicksort(pivot, hi, data);
}
```

...or by naming locales explicitly

```cobegin {
  on TaskALocs do computeTaskA(...);
  on TaskBLocs do computeTaskB(...);
  on Locales(0) do computeTaskC(...);
  0 1 computeTaskA()
  2 3 4 5 6 7 computeTaskB()
  0 computeTaskC()
  0 1```
Locality: Domain Distribution

Domains may be distributed across locales

```chapel
var D: domain(2) distributed Block on CompGrid = ...;
```

A distribution implies...
- Ownership of the domain’s indices (and its arrays’ elements)
- The default work ownership for operations on the domains/arrays

Chapel provides...
- A standard library of distributions (Block, Recursive Bisection, …)
- The means for advanced users to author their own distributions

Locality: Domain Distributions

A distribution must implement...
- The mapping from indices to locales
- The per-locale representation of domain indices and array elements
- The compiler’s target interface for lowering global-view operations

```
"steve"
"mary"
"wayne"
"david"
"john"
"pete"
"peg"
```
Locality: Domain Distributions

A distribution must implement…
…the mapping from indices to locales
…the per-locale representation of domain indices and array elements
…the compiler’s target interface for lowering global-view operations

Locality: Distributions Overview

**Distributions**: “recipes for distributed arrays”

- Intuitively, distributions support the lowering…
  …from: the user’s global view operations on a distributed array
  …to: the fragmented implementation for a distributed memory machine

- Users can implement custom distributions:
  - written using task parallel features, on clauses, domains/arrays
  - must implement standard interface:
    - allocation/reallocation of domain indices and array elements
    - mapping functions (e.g., index-to-locale, index-to-value)
    - iterators: parallel/serial × global/local
    - optionally, communication idioms

- Chapel provides a standard library of distributions…
  …written using the same mechanism as user-defined distributions
  …tuned for different platforms to maximize performance
Other Features

- zippered and tensor flavors of iteration and promotion
- subdomains and index types to help reason about indices
- reductions and scans (standard or user-defined operators)

Outline

- Chapel Context
- Global-view Programming Models
- Language Overview
- Status, Future Work, Collaborations
Chapel Work

- Chapel Team’s Focus:
  - specify Chapel syntax and semantics
  - implement open-source prototype compiler for Chapel
  - perform code studies of benchmarks, apps, and libraries in Chapel
  - do community outreach to inform and learn from users/researchers
  - support users of code releases
  - refine language based on all these activities

Prototype Compiler Development

- Development Strategy:
  - start by developing and nurturing within Cray under HPCS
  - initial releases to small sets of “friendly” users for past few years
  - public release scheduled for SC08
  - turn over to community when it’s ready to stand on its own

- Compilation approach:
  - source-to-source compiler for portability (Chapel-to-C)
  - link against runtime libraries to hide machine details
    - threading layer currently implemented using pthreads
    - communication currently implemented using Berkeley’s GASNet
Compiling Chapel

Chapel Source Code → Chapel Compiler → Chapel Executable

Chapel Standard Modules

Chapel Compiler Architecture

Chapel Source Code → Chapel-to-C Compiler → Generated C Code

Chapel Standard Modules

Internal Modules (written in Chapel)

Runtime Support Libraries (in C)

1-sided Messaging, Threading Libraries

Chapel-to-C Compiler

Generated C Code

Standard C Compiler & Linker

Chapel Executable
Implementation Status

- **Base language:** stable (a few gaps and bugs remain)
- **Task parallel:** stable, multithreaded
- **Data parallel:**
  - stable serial reference implementation
  - initial support for multi-threaded implementation
- **Locality:**
  - stable locale types and arrays
  - stable task parallelism across multiple locales
  - initial support for distributed arrays across multiple locales
- **Performance:**
  - has received much attention in designing the language
  - yet very little implementation effort thus far

Chapel and Research

- Chapel contains a number of research challenges
  - the broadest: “solve the parallel programming problem”
- We intentionally bit off more than an academic project would
  - due to our emphasis on general parallel programming
  - due to the belief that adoption requires a broad feature set
  - to create a platform for broad community involvement
- Most Chapel features are taken from previous work
  - though we mix and match heavily which brings new challenges
- Others represent research of interest to us/the community
Some Research Challenges

- **Near-term:**
  - user-defined distributions
  - zippered parallel iteration
  - index/subdomain optimizations
  - heterogeneous locale types
  - language interoperability

- **Medium-term:**
  - memory management policies/mechanisms
  - task scheduling policies
  - performance tuning for multicore processors
  - unstructured/graph-based codes
  - compiling/optimizing atomic sections (STM)
  - parallel I/O

- **Longer-term:**
  - checkpoint/resiliency mechanisms
  - mapping to accelerator technologies (GP-GPUs, FPGAs?)
  - hierarchical locales

---

Chapel and Community

- **Our philosophy:**
  - Help the community understand what we are doing
  - Make our code available to the community
  - Encourage external collaborations

- **Goals:**
  - to get feedback that will help make the language more useful
  - to support collaborative research efforts
  - to accelerate the implementation
  - to aid with adoption
Current Collaborations

ORNL (David Bernholdt et al.): Chapel code studies – Fock matrix computations, MADNESS, Sweep3D, … (HIPS ’08)

PNNL (Jarek Nieplocha et al.): ARMCI port of comm. layer

UIUC (Vikram Adve and Rob Bocchino): Software Transactional Memory (STM) over distributed memory (PPoPP ’08)

EPCC (Michele Weiland, Thom Haddow): performance study of single-locale task parallelism

CMU (Franz Franchetti): Chapel as portable parallel back-end language for SPIRAL

Possible Collaboration Areas

- any of the previously-mentioned research topics…
- task parallel concepts
  - implementation using alternate threading packages
  - work-stealing task implementation
- application/benchmark studies
- different back-ends (LLVM? MS CLR?)
- visualizations, algorithm animations
- library support
- tools
  - correctness debugging
  - performance debugging
  - IDE support
- runtime compilation
- (your ideas here…)

Brad Chamberlain, Cray Inc.
Next Steps

- Continue to improve performance
- Continue to add missing features
- Expand the set of codes that we are currently studying
- Expand the set of architectures that we are targeting
- Support the public release
- Continue to support collaborations and seek out new ones

Summary

*Chapel strives to solve the Parallel Programming Problem*

through its support for…

…general parallel programming
…global-view abstractions
…control over locality
…multiresolution features
…modern language concepts and themes
Chapel Team

- **Current Team**
  - Brad Chamberlain
  - Steve Deitz
  - Samuel Figueroa
  - David Iten

- **Interns**
  - Robert Bocchino ('06 – UIUC)
  - James Dinan ('07 – Ohio State)
  - Mackale Joyner ('05 – Rice)
  - Andy Stone ('08 – Colorado St)

- **Alumni**
  - David Callahan
  - Roxana Diaconescu
  - Shannon Hoffswell
  - Mary Beth Hribar
  - Mark James
  - John Plevyak
  - Wayne Wong
  - Hans Zima

For More Information

chapel_info@cray.com

http://chapel.cs.washington.edu

Questions?

SC08: Tutorial M04 – 11/17/08
Overview
A. Introduction to PGAS (~ 45 mts)
B. Introduction to Languages
   A. UPC (~ 60 mts)
   B. X10 (~ 60 mts)
   C. Chapel (~ 60 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):

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

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

Because of the time steps, typically, two grids are used.

Heat Transfer in Pictures

- \[ \sum \left( \begin{array}{c} 1.0 \end{array} \right) \div 4 \]

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

shared [BLOCKSIZE] double grids[2][N][N];
shared double dTmax_local[THREADS];
int i, 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);
        }
    }
    if( dTmax < epsilon )
        finished = 1;
    else{
        /*swapping the source & destination “pointers”*/
        dg = sg;
        sg = 1-sg;
    }
    nr_iter++;
} while( !finished );
upc_barrier;
dTmax_local[MYTHREAD]=dTmax;
upc_barrier;
dTmax = dTmax_local[0];
for( i=1; i<THREADS; i++ )
    if( dTmax < dTmax_local[i] )
        dTmax = dTmax_local[i];
upc_barrier;

Reduction operation

Work distribution, according to the defined BLOCKSIZE of grids[2][N][N]
HERE, generic expression, working for any BLOCKSIZE

4-point Stencil

if( dTmax < epsilon )
    finished = 1;
else{
/*swapping the source & destination “pointers”*/
dg = sg;
sg = 1-sg;
}
r
iter++;
} while( !finished );
upc_barrier;
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.
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), Tmp : Array[Real](BigD);
  { A(LastRow) = 1.0D; }
  def run() {
    do {
      finish ateach (p in D)
      Temp(p) = A(p.stencil(1)).reduce(Double.sum)/4
      val delta = (A(D) - Temp(D)).abs().reduce(Double.max)
      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
def run() {
    do {
        finish ateach (z in D.places())
        for (p in D(z))
            Temp(p) = A(p.stencil(1)).reduce(Double.sum)/4

        val delta = Math.abs(A(D) - Temp(D)).reduce(Double.max)
        A(D) = Temp(D);
    } while (delta > epsilon);
}

\(\star\) Flat parallelism: Assume one activity per place is desired.

\(\star\) D.places() returns ValRail of places in D.

\(\star\) D(z) returns sub-region of D at place z.

---

def run() {
    val blocks = Dist.util.block(D, P);
    do {
        finish ateach (z in D.places())
        foreach (q in 1..P)
            for (p in blocks(z,q))
                Temp(p) = A(p.stencil(1)).reduce(Double.sum)/4

        val delta = Math.abs(A(D) - Temp(D)).reduce(Double.max)
        A(D) = Temp(D);
    } while (delta > epsilon);
}

\(\star\) Hierarchical parallelism: \(P\) activities at place \(z\).

\(\star\) Easy to change above code so \(P\) can vary with \(z\).

\(\star\) Dist.util.block(D,P)(z,q) is the region allocated to the \(q\)'th activity in the \(z\)'th place. (Block-block division.)

---

Explicit Loop Chunking

Explicit Loop Chunking with Hierarchical Parallelism
def run() {
    finish async {
        val c = clock.make();
        val D_Base = Dist.unique(D.places);
        val diff = Array.make[Real](D_Base),
                    scratch = Array.make[Real](D_Base);
        aeach (z in D.places()) clocked(c)
            do {
                diff(z)=0.0D;
                for (p in D(z)) {
                    val tmp = A(p);
                    A(p) = A(p.stencil(1)).reduce(Double.sum)/4;
                    diff(z)=Math.max(diff(z), Math.abs(tmp, A(p)));
                }
            next;
            reduceMax(z, diff, scratch);
            while (diff(z) > epsilon);
    }}

    ◆ reduceMax performs an all-to-all max reduction.
    ◆ Temp array is internalized.

    SPMD with all-to-all reduction == MPI style

---

def run() {
    finish async {
        val c = clock.make();
        val D_Base = Dist.unique(D.places);
        val diff = Array.make[Real](D_Base),
                    scratch = Array.make[Real](D_Base);
        aeach (z in D.places()) clocked(c)
            do {
                diff(z)=0.0D;
                foreach (q in 1..P) clocked(c)
                    do {
                        if (q==1) diff(z)=0.0D;
                        var myDiff:Double=0.0D;
                        for (p in blocks(z,q)) {
                            val tmp = A(p);
                            A(p) = A(p.stencil(1)).reduce(Double.sum)/4;
                            myDiff=Math.max(myDiff, Math.abs(tmp, A(p)));
                        }
                        atomic diff(z)= Math.max(myDiff, diff(z));
                        next;
                        if (q==1) reduceMax(z, diff, scratch); next;
                    } while (diff(z) > epsilon);
    }}

    “OpenMP within MPI style”
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 to region transformer.

- `reduceMax` phase overlapped with ghost distribution phase. (few extra lines.)

Comparison of Languages

Chapel
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);

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

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;
var delta = max reduce abs(A(D) - Temp(D));
A(D) = Temp(D);
} while (delta > epsilon);
writeln(A);

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 exterior ⇒ one of several built-in domain generators

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

cfg 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);

Set Explicit Boundary Condition
indexing by domain ⇒ slicing mechanism
array expressions ⇒ parallel evaluation

Compute 5-point stencil

\[
\sum\left(\begin{array}{c}
A(i-1,j) + A(i+1,j) \\
A(i,j-1) + A(i,j+1)
\end{array}\right) / 4
\]

Note: since \((i,j) \in D\) and \(D \subseteq BigD\) and \(Temp: [BigD]\)
⇒ no bounds check required for \(Temp(i,j)\)
with compiler analysis, same can be proven for A’s accesses

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

**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

```chapel
Compute maximum change
```

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

```chapel
Copy data back & Repeat until done
```
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);
```

Heat Transfer in Chapel

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

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);

Comparison of Languages

Comparative Feature Matrix
### Features Matrix

<table>
<thead>
<tr>
<th></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 Communications</strong></td>
<td>Both</td>
<td>Both</td>
<td>Implicit; User can assert locality of a code block (checked at compile-/runtime)</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>Reductions, scans, whole-array operations</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 Model Control</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>Strict with respect to sync/single variables; relaxed otherwise</td>
</tr>
</tbody>
</table>
## Features Matrix

<table>
<thead>
<tr>
<th>Feature</th>
<th>UPC</th>
<th>X10</th>
<th>Chapel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Memory Allocation</td>
<td>Private or shared with or without blocking</td>
<td>Supports objects and arrays.</td>
<td>No pointers -- all dynamic allocations are through allocating new objects &amp; resizing arrays</td>
</tr>
<tr>
<td>Synchronization</td>
<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>
</tr>
<tr>
<td>Type Conversion</td>
<td>C rules Casting of shared pointers to private pointers</td>
<td>Coercions, conversions supported as in OO languages</td>
<td>C#-style rules</td>
</tr>
<tr>
<td>Pointers To Shared Space</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>global-view distributed arrays</td>
<td>Yes, but 1D only</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>Concurrency spawn</td>
<td>upc_forall</td>
<td>async, future, foreach, ateach</td>
<td>begin, cobegin, forall, coforall</td>
</tr>
<tr>
<td>Termination detection</td>
<td>finish</td>
<td>sync</td>
<td>N/A</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 variable</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 richer/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 richer/nested parallelism than SPMD
- you require multidimensional global arrays
- you’re able to work with an emerging technology

Discussion
Overview

A. Introduction to PGAS (~ 45 mts)

B. Introduction to Languages
   A. UPC (~ 60 mts)
   B. X10 (~ 60 mts)
   C. Chapel (~ 60 mts)

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

D. Hands-On (90 mts)
Backup

Heat Transfer in Chapel (Backup 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);
```

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 (array of offsets version)

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)

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

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 {
    for [(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);